Fast Gamma, Beta, Error, Zeta, Lambert function & Bernoulli numbers for Arbitrary Precision.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a continuation of a series of papers dealing with the practical aspects of implementing an arbitrary-precision math package. The paper describes the gamma, beta, error, zeta, and Lambert W_0 functions in arbitrary precision. Some of the mentioned functions rely on factorial, binomial coefficients, Bernoulli numbers, etc. These low-level functions are also described in the sections leading up to the gamma, log gamma, beta, error, zeta, and Lambert W0 functions. We show various methods and how to compute them with arbitrary precision.

Introduction:

Before diving into implementation details, we briefly summarize the mathematical background and typical use cases of the special functions discussed in this paper. This overview clarifies how the functions relate to one another and where they are used in practice.
We start with simple functions such as factorials, falling and rising factorials, and binomial coefficients, and continue with Bernoulli numbers and various optimization techniques to improve performance.
We then turn our attention to the Gamma, Log Gamma, Beta, Error, Zeta, and Lambert W_0 functions.
Historically, these functions trace back to Euler (Gamma), Bernoulli (polynomials and numbers), Lambert (transcendental equations), and Riemann (analytic number theory). Most of them were formalized during the 18th and 19th centuries and remain central in modern numerical analysis.

As usual, we will show the actual C++ source code for the computation using the author's arbitrary-precision Math library (see [1]).

This paper is part of a series of arbitrary-precision papers that describe methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root and inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision

5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from an arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision
12. Fast Computation of Stirling's numbers in arbitrary precision. HVE Fast Computation of Stirling numbers in arbitrary precision
13. Fast Prime computation in arbitrary precision. HVE Fast Prime Computation in arbitrary precision
14. Fast PRNG in arbitrary precision. HVE Fast PRNG in arbitrary precision
15. Fast Fibonacci sequence in arbitrary precision. HVE Fast Fibonacci in arbitrary precision
16. Fast arbitrary precision Multiplication. HVE Fast arbitrary precision integer multiplication
17. Fast Factorial and Binomial computation with arbitrary precision. HVE Fast factorial and binomial for arbitrary precision.docx

Notation and Definitions:

Throughout this paper, we use the following notation and naming conventions:

Function	Symbol	Description
Bernoulli number	B_n	Sequence of rational constants used in many series expansions and formulas, such as the Euler–Maclaurin summation and the Stirling series for Γ(x)
Bernoulli Polynomial	$B(x)_n$	Polynomial generalization of B_n
Gamma	Γ(x)	Extension of the factorial function
Log Gamma	ln Γ(x)	Natural logarithm of
Beta	B(x,y)	Related to Gamma: B(x,y)=Γ(x)Γ(y)/Γ(x+y)
Error	erf(x), erfc(x)	Probability integral and its complement
Zeta	ζ(s)	Riemann zeta function
Lambert W	$W_0(x)$	Principal branch of the Lambert W function

Changes Log

16-Feb-2026 Fixed typos and bugs in the code for the Riemann Zeta function.

1-Nov-2025 Revised minor part of the documents and updated the Gamma functions section, plus added a new section on the recommended methods of computing the Log Gamma function.

Fast Gamma, Beta, Error, Zeta, Lambert function & Bernoulli numbers for Arbitrary Precision

Contents

Abstract: 1
Introduction: 1
  Notation and Definitions: 2
Changes Log 2
The Arbitrary precision library 6
    Internal format for float_precision variables 7
    Normalized numbers 7
Factorials 8
    Source factorial via recursion 8
    Source factorial looping 8
    Source factorial loop-based max 9
  Factorial using binary splitting 10
    Source factorial binary splitting 10
  Performance of Factorials 10
  Recommendation for Factorials 11
Falling Factorials 11
    Source falling factorial 12
    Source falling factorial via threading 13
    Source falling factorial hybrid implementation 13
Binomial coefficients 14
    Source for binomial 15
Bernoulli numbers 15
  Bernoulli using recursion 16
    Source Bernoulli using recursion 16
  Bernoulli numbers using an explicit formula 17
    Source Bernoulli using explicit formula 17
  Fast Calculation of Bernoulli Numbers 18
    Example of Bernoulli number computation 19
    Source of Fast Bernoulli calculation 20
  Bernoulli performance 22
  Recommendation of the Bernoulli method 22
Bernoulli Polynomials 23
    Source for Bernoulli polynomials 23
Gamma function 24
    Algorithm for Gamma computation 26
    Source for the Half Integer calculation 26
  Lanczos-Spouge method 27
    Source of Lanczos-Spouge method. 28
  Stirling asymptotic series method 29
    Source of the Stirling asymptotic method 30
    Optimizing the Stirling Gamma method 32
    Higher Performance source of the Stirling asymptotic method 32
  Integration by parts method 34
    Source Integration by parts 35
  Gamma Performance 36

Recommendation for the Gamma function 37
Log Gamma function 37
Why compute log gamma 37
A short note on the C and C++ interfaces 38
Three practical ways to compute ln(|Γ(x)|) 38
1) Take the logarithm of Γ directly 38
2) Stirling series for log gamma 39
Source Stirling log gamma functions 39
3) Lanczos–Spouge for ln Γ(x) (minimal changes from Γ(x)) 42
For the reflection formula in log form with negative arguments, we convert the Euler reflection to logarithms: 42
Source for the Lanczos-Spouge log gamma. 43
Performance of ln(|Γ(x)|) 44
Recommendation for the ln(|Γ(x)|) function 45
The Beta function 46
Source Beta function 46
The Error function 46
Performance of the error function 48
Source erf with the concurrent series method 49
Source erfc 50
Recommendation for the Error function 50
Lambert W function 51
A Suitable starting point for Lambert W Iteration. 51
Newton's quadratic method 52
Halley's cubic method 52
Boyd's quadratic method 53
Initial performance of the Lambert W function 53
Source Newton method with dynamic precision 53
Source Halley methods with dynamic precision 54
Source Boyd method with dynamic precision 55
Performance of Lambert W function 56
Recommendation for Lambert W function 57
Riemann Zeta function 57
Source for zeta function computation. 60
Reference 63
Appendix 64
Cross-reference clarity 64

The Arbitrary precision library

If you are already familiar with the arbitrary precision library, you can skip this section.

To understand the C++ code and text, we need to highlight a few features of the arbitrary precision library, where the class name is *float_precision*. Instead of declaring a variable as a float or double, you replace the type name with *float_precision*. E.g.

```
float_precision f;  // Declare an arbitrary precision float with 20 decimal digits of precision
```

You can add a few parameters to the declaration. The first is the optional initial value, and the second is the optional floating-point precision. The native type of a *float* has a fixed size of 4 bytes and 8 bytes for *double*; however, since this precision can be arbitrary, we can declare the wanted precision as the number of ***decimal digits*** we want to use when dealing with the variable. E.g.

```
float_precision fp(4.5);  // Initialize it to 4.5 with default 20 digits precision
float_precision fp(6.5,10000); // Initialize it to 6.5 with a precision of 10,000 digits
```

The precision of a variable can be dynamic and change throughout the code, which is very handy for manipulating the variable. To change or set the precision, you can call the method .precision(), e.g.

```
f.precision(100000);                // Change the precision to 100,000 digits
f.precision(fp.precision()-10);  // Lower the precision with 10 digits
f.precision(fp.precision()+20); // Increase precision with 20 digits
```

There is another method for manipulating the exponents of the variables. The method is called .exponent() and returns or sets the exponent as a power of two (same as for our regular built-in types *float* and *double*). E.g.

```
f.exponent();            // Return the exponent as 2^e
f.exponent(0)            // Remove the exponent
f.exponen(16)            // Set the exponent to 2^16
```

There is a second way to manipulate the exponent: the class method. .adjustExponent(). This method adds the parameter to the internal variable that holds the exponent of the float_precision variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number by 2.
f.adjustExponent(-1);  // Subtract 1 from the exponent, the same as dividing the number with 2.
```

This allows very fast multiplication or division with a number that is any power of two.

The method .iszero() returns true if the float_precision number is zero, otherwise false.

There is an additional method(), but I will refer the user to the reference manual for the arbitrary-precision math package for details.

All the standard operators and library calls that work with the built-in type float or double will also work with the float_precision type using the same name and calling parameters.

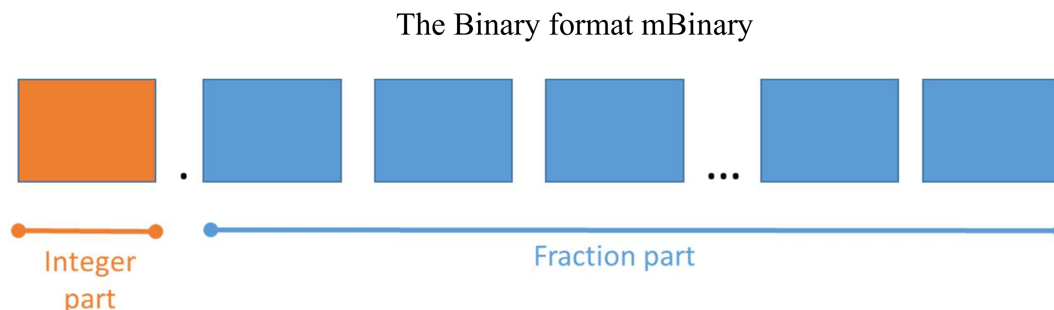
Internal format for float_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mBinary;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integers to store our floating-point precision number.

The method .size() returns the number of internal vector entries needed to hold the number.



- The binary format consist of an unlimited number of 64bit unsigned integer blocks.
- One block in front of the period sign '.' (the integer part of the number)
- Zero or more blocks of fractions after the '.' (the fraction sign of the number)
- The binary number is stored in a STL vector class and defined
  - vector<uintmax_t> mBinary;
- There is always one entry in the mBinary vector.
- Size of vector is always >=1
- A Number is always stored normalized. E.g. the integer part is 1 or zero
- The sign, exponent, precision, rounding mode is stored in separate class fields.

There are other internal class variables, such as the sign, exponent, precision, and rounding mode, but they are not important for understanding the code segments.

Normalized numbers

A float_precision variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details, see [1].

Factorials

Factorials are among the simplest formulas to implement. Its definition is:

$$n! = n \cdot (n-1) \cdot (n-2) \dots 2 \cdot 1 \tag{1}$$

Or using a recursive definition:

$$n! = n \cdot (n-1)! \tag{2}$$

Where 0!=1

Most often, you see it implemented as straightforward recursive code.

Source factorial via recursion

```
uintmax_t factorial(uintmax_t k)
{
	if (k <= 1)
		return 1;
	return k * factorial(k - 1);
}
```

One issue is that you get an overflow pretty quickly. Even when using the above code on a 64-bit system, you get an overflow after 20! and higher.
This is one of the many situations where an arbitrary-precision integer math library comes in handy to save the day.

As we were told back in the computer science class, recursion is good, but unrolling it into a loop-based algorithm is better. The above code, rewritten as a more efficient loop-based implementation, is:

Source factorial looping

```
uintmax_t factorialloop(uintmax_t k)
{
	uintmax_t res = 1;
	for (; k > 1; --k)
		res *= k;
	return res;
}
```

To overcome the limitation of a maximum of 19! In a 64-bit system, you need to switch to integer arbitrary precision arithmetic.
The loop-based source above looks like this in the author's own arbitrary precision math package, where the type for an integer in arbitrary precision is called *int_precision*.

```
int_precision factorialloop(const int_precision& kip)
{
	uintmax_t k = (uintmax_t)kip;
	int_precision res = int_precision(1);
```

```
	for (; k > 1; --k)
		res *= int_precision(k);
	return res;
}
```

Switching to arbitrary precision allows you to compute "unlimited" factorials; however, the above algorithm is not particularly fast, since arbitrary precision slows down computation. One improvement could be to perform some intermediate computations using 64-bit arithmetic. E.g., we could group two consecutive numbers, add them to an arbitrary-precision result, and, to limit any overflow, rearrange the factorial by starting with the top n multiplied by 1, then (n-1)2, etc.

$$n! = n \cdot 1 \cdot (n-1) \cdot 2 \ldots \left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right) \tag{3}$$

This has the benefit that the biggest number will be less than $(\frac{n}{2})^2$ Limiting the possibility of overflow. The arbitrary precision implementation will be as follows:

```
int_precision factorialloopbalance(const int_precision& kip)
{
	uintmax_t l = 1, k=(uintmax_t)kip;
	int_precision res = int_precision(1);
	for (; k > l; --k,++l)
		res *= int_precision(k*l);
	if(k==l)
		res *= int_precision(k);
	return res;
}
```

This provides a nice speedup, since the pairwise multiplication of k and l is performed using 64-bit arithmetic. Continue along with that idea; you can generalize it and multiply in pairs many times, depending on the range of numbers you are multiplying.

Source factorial loop-based max

```
int_precision factorial(const int_precision& kip)
{
	const uintmax_t UINTMAX_T_MAX= ~((uintmax_t)0);
	uintmax_t m=1, kk, prod, k=(uintmax_t)kip;
	int_precision res = int_precision(1);

	if (k <= 1)
		return res;
	for (kk = 1; k>m; --k, ++m)
	{
		prod = k * m;		// Never overflow
		if (kk < UINTMAX_T_MAX / prod)
			kk *= prod;
		else
		{
			res *= int_precision(kk);
			kk = prod;
		}
	}
	res *= int_precision(kk);
	if (k == m)
		res *= int_precision(k);
	return res;
}
```

The performance improvement from the simple loop-based version to the more advanced version above is 6-7x. In arbitrary-precision arithmetic, you always want to prioritize speed over simplicity.

Factorial using binary splitting

There is one method that needs to be explained, and that is the computation using the binary splitting method. Generally, with the balance loop-based approach, you are fine. Still, if you are dealing with very large factorials, the binary splitting method will perform better than any of the other methods. The binary splitting method recursively splits the computation into two halves, calls itself on each half, and then multiplies the two halves. In essence, it does the same number of multiplications as the other method, but does so in a more efficient way, particularly when using arbitrary-precision integer arithmetic. The binary splitting method calculates the division of two factorials. $\frac{n!}{m!}$. To find n! You set m=0. The algorithm is as follows and is called with two integer parameters n and m.

```
FactorialBinarySplitting(n,m)
        Mid=(n+m)/2
        High = FactorialBinarySplitting(n,mid)
        Low = FactorialBinarySplitting(mid,m)
        Return high * low
```

n! is computed with the call FactorialBinarySplitting(n,0);

Source factorial binary splitting

```
int_precision factorial_binary_splitting(const int_precision& a, const int_precision& b)
{
        const int_precision c1(1), c2(2);
        const int_precision diff = a - b;

        switch (static_cast<uintmax_t>(diff))
        {// Base cases
        case 0: return c1; break;
        case 1: return a; break;
        case 2: return a * (a - c1); break;
        case 3: return a * (a - c1) * (a - c2); break;
        }
        int_precision m = (a + b) / c2;
        return factorial_binary_splitting(a, m) * factorial_binary_splitting(m, b);
}
```

Performance of Factorials

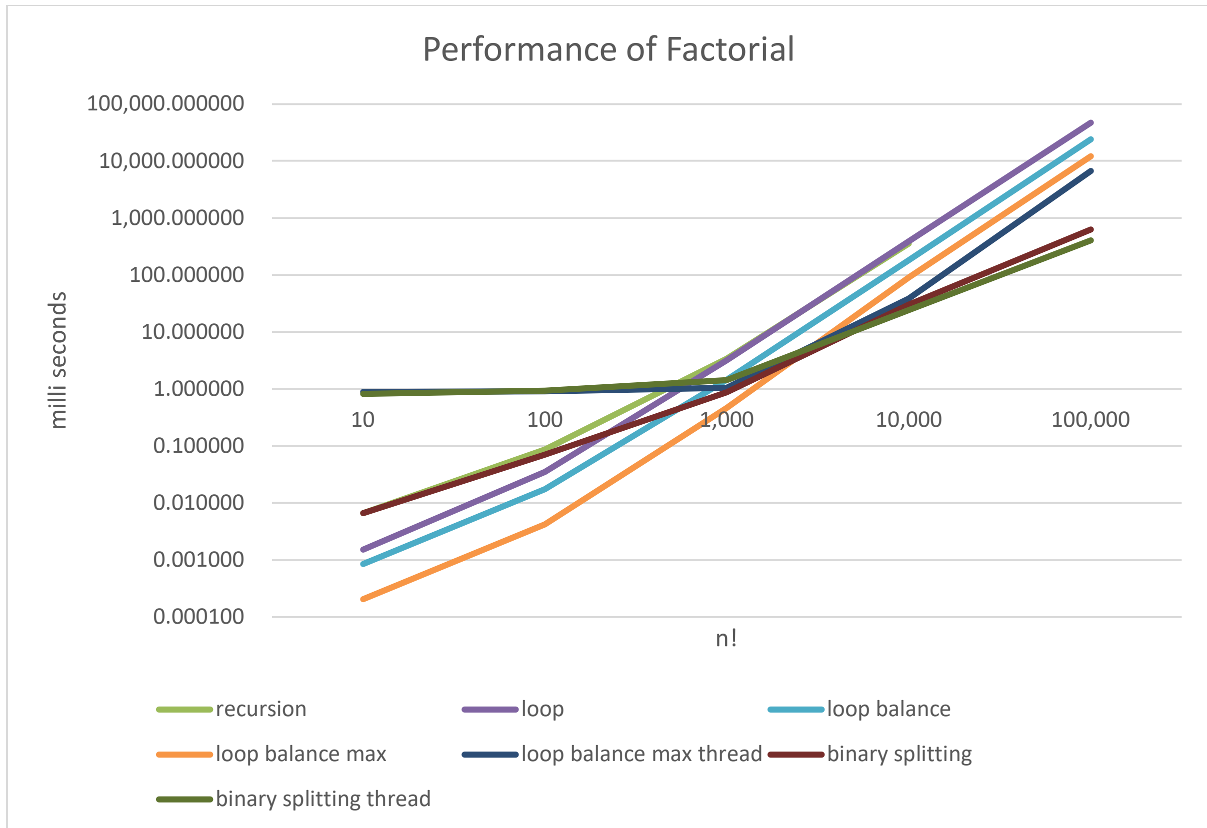


Figure 1 Performance of factorial methods.

Looking at this performance chart, we see that the factorial using standard recursion and naive looping is not performing very well. Loop balancing is better, but the loop-balancing max (optimizing the use of 64-bit arithmetic) is even better. Implementing loop balancing as a two-threaded approach can further improve performance; however, initially, the binary splitting method performed worse than the loop-based version. But around the 5000! Decimal digit mark. We see that the binary splitting method overtakes the loop-based method and begins to perform significantly better.

Recommendation for Factorials

You need a hybrid approach where below the 5000! You used the loop-based method (and maybe even the threaded version). And above 5000! Mark, you switched to the binary splitting method.

Falling Factorials

The falling factorial describes the factorial between two numbers m and n in falling (or descending notation):

$$n^{\underline{m}} = n \cdot (n-1) \cdot (n-2) \dots (n-m+1) \tag{4}$$

Falling factorials are useful when calculating the binomial coefficient:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \tag{5}$$

Since the falling factorial is:

$$\frac{n!}{(n-m)!} = n^{\underline{m}} \tag{6}$$

It can be handled with a single call to the falling factorial function rather than two calls to the factorial function.

See the next chapter for the binomial coefficients.

The falling factorial has two parameters, n and m, compared to the regular factorial, and we again optimized the computation by performing interim computations using 64-bit arithmetic.

Source falling factorial

```
int_precision fallingfactorial(const int_precision& nip, const int_precision& mip)
{
	const uintmax_t UINTMAX_T_MAX = ~((uintmax_t)0);
	uintmax_t m = (uintmax_t)mip, n = (uintmax_t)nip;
	uintmax_t kk, prod;
	int_precision res = int_precision(1);

	if (n <= 1)
		return res;
	for (kk = 1; n > m; --n, ++m)
	{
		prod = n * m;		// Never overflow
		if (kk < UINTMAX_T_MAX / prod)
			kk *= prod;
		else
		{
			res *= int_precision(kk);
			kk = prod;
		}
	}
	res *= int_precision(kk);
	if (n == m)
		res *= int_precision(n);
	return res;
}
```

Now that we have the falling factorial, we can improve the factorial by splitting a factorial call for n! into two calls. Into two.

$$\frac{n!}{(n-m)!} \; and \; (n-m)! \tag{7}$$

Where we can run each calculation in parallel, see the source below, which uses C++ threading.

Source falling factorial via threading

```
int_precision factorialthread(const int_precision& nip, const int_precision& mip)
{
	uintmax_t l = (uintmax_t)mip, k = (uintmax_t)nip;
	int_precision  res, pflow = int_precision(1),
		       fmid = kip>>int_precision(1), pfhigh=nip;

	if (k <= 1)
		return res;

	std::thread first([=, &pflow, &pfmid]()       // interval [1..k/2]
		{pflow=fallingfactorial(pflow, pfmid); });
	std::thread second([=, &pfhigh, &pfmid]()     // interval [k/2+1..k]
		{pfhigh=fallingfactorial(pfmid+1, pfhigh); });
	first.join();
	second.join();
	res = pflow * pfhigh;
	return res;
}
```

There is some overhead in setting up a thread, so the above code is worthwhile only when dealing with factorials exceeding 2,000!

Like the factorial, we can gain some speed by reusing the binary splitting method for the falling factorial, noticing that. $n^{\underline{m}}$ Is inded the two parameters to the binary splitting method *n, m*. By measuring, we see the same performance gain as for the factorial, and the crossover is around 2,000, where the binary splitting method takes over. A threaded version of the binary_ splitting method first outperformed the regular binary splitting method around 5,000. We present the source for our final hybrid version based on the above notes.

Source falling factorial hybrid implementation.

```
int_precision fallingfactorial_hybrid_thread(const int_precision& n, const
int_precision&m )
{
	const int_precision c0(0), c1(1), c2(2);

	auto fallingfactorial_loop = [&](const int_precision& n, const int_precision& m)
	{
		const uintmax_t UINTMAX_T_MAX = ~(0ull);
		uintmax_t l = static_cast<uintmax_t>(m), kk, prod, k =
static_cast<uintmax_t>(n);
		int_precision res = c1;

		if (k <= 1)
			return res;
		for (kk = 1; k > l; --k, ++l)
		{
			prod = k * l;         // Never overflow
			if (kk < UINTMAX_T_MAX / prod)
				kk *= prod;
			else
			{
				res *= int_precision(kk);
				kk = prod;
			}
		}
		res *= int_precision(kk);
		if (k == l)
```

```
                        res *= int_precision(k);
                return res;
        };

        // We need to use the std::function to be able to call the lambda function recrusively.
        std::function<int_precision(const int_precision&, const int_precision&)> fallingfactorial_binary_splitting = [&](const int_precision& a, const int_precision& b)
        {
                const int_precision diff = a - b;

                switch (static_cast<uintmax_t>(diff))
                {// Base cases
                case 0: return c1; break;
                case 1: return a; break;
                case 2:return a * (a - c1); break;
                case 3: return a * (a - c1) * (a - c2); break;
                default:// Fall through
                        break;
                }
                int_precision m = (a + b) / c2;
                return factorial_binary_splitting(a, m) * factorial_binary_splitting(m, b);
        };

        // The balance loop base is fastest below 2000!
        if (n <= int_precision(2000))
                return fallingfactorial_loop(n,m+c1);

        // Do the binary splitting method above 5000!
        // with or without threading. 2 ways Threading improve the speed with 30-60% over non threaded version
        int_precision mid = (n+m) / c2, high, low;
        if (n <= int_precision(5000))
        {
                high = fallingfactorial_binary_splitting(n, mid);
                low = fallingfactorial_binary_splitting(mid, m);
        }
        else
        {
                std::thread first([&]()
                        {
                                high = fallingfactorial_binary_splitting(n, m);
                        });             // interval [n...m]
                std::thread second([&]()
                        {
                                low = fallingfactorial_binary_splitting(m, c0);
                        });             // interval [m...0]
                first.join();
                second.join();
        }
        return high * low;
}
```

Binomial coefficients

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \ where\ 0 \le m \le n \tag{8}$$

As mentioned under partial factorials, the binomial coefficients can be implemented using the two different factorial functions.

Source for binomial

```
int_precision binomial(const int_precision& nip, const int_precision& mip)
{
	uintmax_t n(nip), m(mip);
	int_precision llip, kkip;
	if (m > n) return int_precision(0);
	if (m > n - m)  // Use the identity to reduce m
		m = n - m;
	llip = int_precision(n - m + 1);
	kkip = int_precision(m);
	return fallingfactorial(llip,nip)/factorial(kkip);
}
```

In the above function, we also use the identity that:

$$\binom{n}{m} = \binom{n}{n-m} \tag{9}$$

Sometimes it is necessary to compute a steady stream of increasing binomial coefficients. E.g., for calculating Bernoulli's number, there is a need for that. This can lead to optimization if you instead use one of the two binomial recurrences:

$$\binom{n+1}{m} = \frac{n+1}{n-m+1}\binom{n}{m} = \binom{n}{m} + \binom{n}{m-1} \tag{10}$$

Or:

$$\binom{n}{m+1} = \frac{n-m}{m+1}\binom{n}{m} \tag{11}$$

Bernoulli numbers

The Bernoulli numbers B_n are a sequence of rational numbers. E.g., one method for calculating the gamma function is to use Bernoulli numbers. In arbitrary precision for floating point, you need to specify the precision for calculating the gamma function. Instead of presenting the Bernoulli number as a floating point, we use fraction arithmetic to calculate the Bernoulli numbers. This is exact, and we can calculate them once and then convert them back to any arbitrary-precision floating-point number at any precision. The Bernoulli numbers can be generated using several different methods. In this paper, we will examine:

- Bernoulli number via recursion
- Bernoulli number via explicit formula

- Fast Bernoulli number calculation

Bernoulli using recursion

The Bernoulli numbers are defined via the coefficients of the power series of $\frac{t}{e^t-1}$. For integers, n>0 B_n is defined as:

$$\frac{t}{e^t-1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} t^n \quad (12)$$

If we multiply both sides by e^t-1 you get:

$$t = (e^t - 1) \sum_{n=0}^{\infty} \frac{B_n}{n!} t^n \quad (13)$$

Equating the coefficients of t^{n+1} you get the recursion formula below:

$$B_n = \frac{-1}{n+1} \sum_{k=0}^{n-1} B_k \binom{n+1}{k} \quad (14)$$

Where B_0=1, B_1=$\frac{-1}{2}$.

The first ten Bernoulli numbers are:

n	0	1	2	3	4	5	6	7	8	9
fraction	1	$\frac{-1}{2}$	$\frac{1}{6}$	0	$\frac{-1}{30}$	0	$\frac{1}{42}$	0	$\frac{-1}{30}$	0

The benefit of the recursion formula is simplicity, but it has the drawback that you need to generate all the preceding Bernoulli numbers.

Source Bernoulli using recursion

```
fraction_precision<int_precision> bernoulli(size_t bno)
{
	static vector<fraction_precision<int_precision> > BernoulliVector;
	size_t n, k;
	fraction_precision<int_precision> B, sum;
	int_precision bin;

	if (BernoulliVector.size() == 0)  // Uinitialized Bernoulli vector
	{	// Initialize the first 2 Bernoulli Numbers
		BernoulliVector.push_back(fraction_precision<int_precision>(1, 1));  // B0=1
		BernoulliVector.push_back(fraction_precision<int_precision>(-1, 2)); // B1=-
0.5
	}
	for (; BernoulliVector.size() < bno + 1;)
	{	// Not enough numbers in the vector then add the next number
		n = BernoulliVector.size(); // Current size
		sum = fraction_precision<int_precision>(0, 1);
		if ((n & 0x1))  // Odd?
		{// For odd index numbers, Bernoulli is zero for index > 2
			BernoulliVector.push_back(sum);
			continue;
		}
		bin = int_precision(1);
```

```
            // Bn is even. Calculate the next Bn
            for (k = 0; k < n; ++k)
            {
                B = BernoulliVector[k];
                if (k > 0)
                {       // Using binomial recursion
                    bin=int_precision(n+2-k)*bin / int_precision(k);
                    if (B.iszero())
                        continue;
                    B *= fraction_precision<int_precision>(bin);
                }
                sum += B;
            }
            sum *= fraction_precision<int_precision>(-1, n + 1);
            BernoulliVector.push_back(sum);
        }
        return BernoulliVector[bno];
}
```

Bernoulli numbers using an explicit formula

There exist many explicit formulas for generating Bernoulli numbers. One of them is the following formula:

$$B_n = \sum_{k=0}^{n} \sum_{v=0}^{k} (-1)^v \binom{k}{v} \frac{v^n}{k+1} \tag{15}$$

The benefit of using an explicit formula is that you don't need to generate a Bernoulli number in increasing order, but only the Bernoulli number you need. Unfortunately, the double looping and the repetitive recalculation of v^n make the formula slow compared to the others.
The source below includes two performance enhancements using Binomial recurrence and caching v^n calculation, which doubles the speed of the explicit formula for Bernoulli number calculation.

Source Bernoulli using explicit formula

```
fraction_precision<int_precision> bernoulliExplicit(size_t bno)
{
        static vector<fraction_precision<int_precision> > BernoulliVector;
        size_t n, k, v;
        fraction_precision<int_precision> sum;
        int_precision bin, pw;
        vector<int_precision> vp;

        if (BernoulliVector.size() == 0)  // Uinitialized Bernoulli vector
        {       // Initialize the first 2 Bernoulli Numbers
            BernoulliVector.push_back(fraction_precision<int_precision>(1, 1));  // B0=1
            BernoulliVector.push_back(fraction_precision<int_precision>(-1,2));  // B1=-
0.5

        }
        for (; BernoulliVector.size() < bno + 1;)
        {       // Not enough numbers in the vector, then add the next number
            n = BernoulliVector.size(); // Current size
            sum = fraction_precision<int_precision>(0, 1);
            if ((n & 0x1))  // Odd?
            {// For odd index numbers Bernoulli is zero for index > 2
                BernoulliVector.push_back(sum);
                continue;
            }
```

```
                // Bn is even Calculate the next Bn
                for (k = 0; k <= n; ++k)
                {
                        if (k == 0)
                                vp.resize(0);
                        bin = int_precision(1);
                        for (v = 0; v <= k; ++v)
                        {
                                if (v > 0) // binomial recurrences as a speed-up trick
                                        bin = int_precision(k + 1 - v) * bio /
int_precision(v);
                                // Speedup trick to reduce the amount you recalculate v^n
                                if (vp.size() > v)
                                        pw = vp[v];
                                else
                                {
                                        pw = ipow(int_precision(v), int_precision(n));
                                        vp.push_back(pw);
                                }

                                pw *= bin;
                                if ((v & 0x1))
                                        pw.change_sign();
                                sum+=fraction_precision<int_precision>(pw,int_precision(k+
1));
                        }
                }
                BernoulliVector.push_back(sum);
        }
        return BernoulliVector[bno];
}
```

Fast Calculation of Bernoulli Numbers

A fast method for calculating the Bernoulli numbers was given in [4]. They use the following algorithm to compute the Bernoulli number as a fraction. $B_n = \frac{a}{d}$. The Algorithm presents the six steps as follows:

Algorithm 1

1. $K = \frac{2(n!)}{(2\pi)^n}$
2. $d = \prod_{p-1|n} p$
3. $N = \left\lceil (Kd)^{\frac{1}{n-1}} \right\rceil$
4. $z = \prod_{p \leq N} \frac{1}{1-\frac{1}{p^n}}$
5. $a = (-1)^{\frac{n}{2}+1} round(dKz)$
6. $B_n = \frac{a}{d}$

Step 2 is the von Staudt-Clausen theorem, which determines the fractional part of the denominator of the Bernoulli number.

This method also has the same benefit as the explicit formula for Bernoulli numbers: it can generate them in any order. In [4], they provide an illustrative example of the calculation I present below.

Example of Bernoulli number computation

Find the Bernoulli number B_{50} with 50-digit precision.

1. $K = \frac{2(n!)}{(2\pi)^n} = \frac{2(50!)}{(2\pi)^{50}} = 75008667460769577047477 36.71552473316456403804367604$
2. The divisor of n=50 is 1,2,3,5,10,25,50 the p-1 primes are 2,3,6,11,26,51 where 6, 26, and 51 is not a prime $so\ d = 2 \cdot 3 \cdot 11 = 66$
3. Taking the ceiling function, you get N=4
4. z=1.0000000000000008881784210930815902983501390146827
5. dKz=495057205241079648212477524.99999999442615111210652
6. and therefore $B_{50} = \frac{495057205241079648212477525}{66}$

K and z from the above algorithm are arbitrary-precision floating-point numbers, and the question arises: what precision do you need in your calculation to get the correct result? In the example for B_{50}, 50-digit precision was sufficient; however, as we go higher, we need to ensure that K and the other variable have sufficient precision to obtain the correct value for the Bernoulli number.

In [3], the estimate of the magnitude of a Bernoulli number is:

$$|B_{2n}| \sim 2(\frac{n}{\pi e})^{2n} \tag{16}$$

Or

$$|B_n| \sim 2(\frac{n}{2\pi e})^{n} \tag{17}$$

You get the equivalent magnitude of decimal digits to be:

$$Magnitude\ of\ (|B_n|) \sim \frac{n(ln(n)-l\ \ (2\,\pi)-1)+ln(2)}{\ln(10)} \tag{18}$$

E.g., from the previous example with n=50, you get ~ 24, which is close enough to the exact value of 24.87. For n=100, you get 77, where the exact value is 78.45.

In step 1, we know that n! could represent some pretty huge numbers. When calculating K we could separate the power of 2:

$$K = \frac{2}{2^n}\frac{n!}{\pi^n} = 2^{1-n}\frac{n!}{\pi^n} \tag{19}$$

The 2^{1-n} can be handled by simply adjusting the exponent of the floating-point variable without any loss of precision, and we can determine the magnitudes of both the numerator and the denominator individually.

The numerator n! can be approximated using Stirling's approximation, where the magnitude is found as:

$$Magnitude\ of\ n! \sim \frac{n \cdot \ln(n) - n}{\ln(10)} \quad (20)$$

And for the denominator:

$$Magnitude\ of\ \pi^n \sim \frac{n \cdot \ln(\pi)}{\ln(10)} \quad (21)$$

In the last variable, we need to consider the variable a in. $B_n = \frac{a}{d}$ Since we have already found the magnitude of B_n, we then have the magnitude of a_n=Magnitude(B_n)+Magnitude(d_n). However, it is not easy to estimate a priori of the magnitude of d_n in step 2. However, when calculating d_n you can take log10(denominator) and add it to the magnitude of B_n, which should give the needed minimum precision for a_n.

Example:

n	Magnitude of n!	Magnitude of π^n	Magnitude of $\lvert B_n \rvert$
50	64	25	24
100	158	50	78
200	375	99	215
300	614	149	374
400	869	199	549
500	1,134	249	734
600	1,408	298	928
700	1,689	348	1,130
800	1,977	398	1,337
900	2,270	447	1,550
1,000	2,568	497	1,768

For K, π, and z, you can use the magnitude of the factorial; however, that will be a total overkill. Instead, I recommend you use a working precision equivalent to:

$$Working\ precision = Magnitude\ of\ |Bn| + Magnitude\ of |\ dn| + n/8 \quad (22)$$

This has been tested with Bernoulli numbers up to 1,000 and has delivered accurate results.

Source of Fast Bernoulli calculation

```
fraction_precision<int_precision> bernoulli3(size_t bno)
{
	static vector<fraction_precision<int_precision> > BernoulliVector;
	const size_t berpre = (size_t)ceil((bno*(log(bno)-log(2*3.14159293)-
1)+log(2.0))/log(10)); //  Bernoulli Magnitude
	const size_t precision =  std::max(bno,PRECISION);
	const size_t facpre = (size_t)ceil((bno*log(bno)-
bno+0.5*log(2*3.14159293*bno))/log(10));	//  Factorial Magnitude
	const size_t denpre = (size_t)ceil((bno*log(3.14159293)) / log(10)); //  pi
Magnitude
	size_t workprec;	// The current working precision for float_precision
```

```
	size_t i, addprecision;
	intmax_t n;
	uintmax_t N;
	vector<uintmax_t> dv;
	int_precision dip, aip(0);
	float_precision pi, K, pw, z;

	if (BernoulliVector.size() == 0)  // Uinitialized Bernoulli vector
	{	// Initialize the first 2 Bernoulli Numbers
		BernoulliVector.push_back(fraction_precision<int_precision>(1, 1));  // B0=1
		BernoulliVector.push_back(fraction_precision<int_precision>(-1, 2)); // B1=-
0.5

	}

	for (; BernoulliVector.size() < bno + 1;)
	{	// Not enough numbers in the vector, then add the next number
		n = BernoulliVector.size(); // Current size
		if ((n & 0x1))  // Odd?
		{	// For odd index numbers, Bernoulli is zero for index > 2
			BernoulliVector.push_back(fraction_precision<int_precision>(0, 1));
			continue;
		}

		// Do a new calculation in 6 steps
		// Step 1 Calculate d
		dip = int_precision(1);
		N = 1;
		addprecision = 0;
		divisible(n, dv);				// divisible set vector dv
		for (i = 0; i < dv.size(); ++i)
		{
			if (isprime(dv[i] + 1))		// To do Check for overflow in 64-bit
			{
				if ((~0ull) / (dv[i] + 1) > N)
					N *= dv[i] + 1;
				else
				{	overflow low using uintmax_t arithmetic
					addprecision += (size_t)ceil(log10(N));
					dip *= int_precision(N); // multiply dip and reset N
					N = dv[i] + 1;
				}
			}
		}
		dip *= int_precision(N);
		addprecision += (size_t)ceil(log10(N));

		// Step 2 Calculate K
		if (pi.iszero())  // Has pi been calculated?
		{
			pi.precision(facpre);		// Working precision for pi
			// overallocate pi so we only have to do it once
			pi = _float_table(_PI, std::max(facpre,(size_t)100));
		}
		// Now set adequate working precision
		workprec = berpre + addprecision + precision/8;
		K.precision(workprec);		// Set adequate working precision
		pw.precision(workprec);		// Set adequate working precision
		pw = float_precision(n, precision);
		pw = pow(pi, pw);			// pw=(PI)^n
		K=float_precision(factorial(int_precision(n)),workprec)/pw; //
K=(n!)/((PI)^n
		K.adjustExponent(1 - n);
		// K=2K/2^n=2(n!)/((2PI)^n
		K *= float_precision(dip,precision);  // K=Kd

		// Step 3 Calculate N. No need for higher precision
		N = (uintmax_t)ceil(pow(float_precision(K),
_float_precision_inverse(float_precision(n-1)))); // N=ceil((Kd)^(1/n-1))
```

```
		// Step 4 Calculate z and update K
		z.precision(workprec); // Set adequate working precision, same as K
		z = float_precision(1);
		for (i = 2; i <= N; ++i)
		{
			if (isprime(i))
			{
				z *= (float_precision(1) - pow(float_precision(i, workprec),
float_precision(-n, precision)));
			}
		}
		z = z.inverse(); // same as z=1/z;
		K *= z;

		// Step 5 Calculate a
		if ((n / 2 + 1) & 0x1)  // Odd
			K.change_sign();
		K = round(K);
		aip = (int_precision)(K);

		// Step 6  Save Bn as aip/dip
		BernoulliVector.push_back(fraction_precision<int_precision>(aip, dip));
	}
	return BernoulliVector[bno];
}
```

Bernoulli performance

The Performance chart shows the time required to generate the first 100 Bernoulli numbers, then the next 100, up to 1,000.

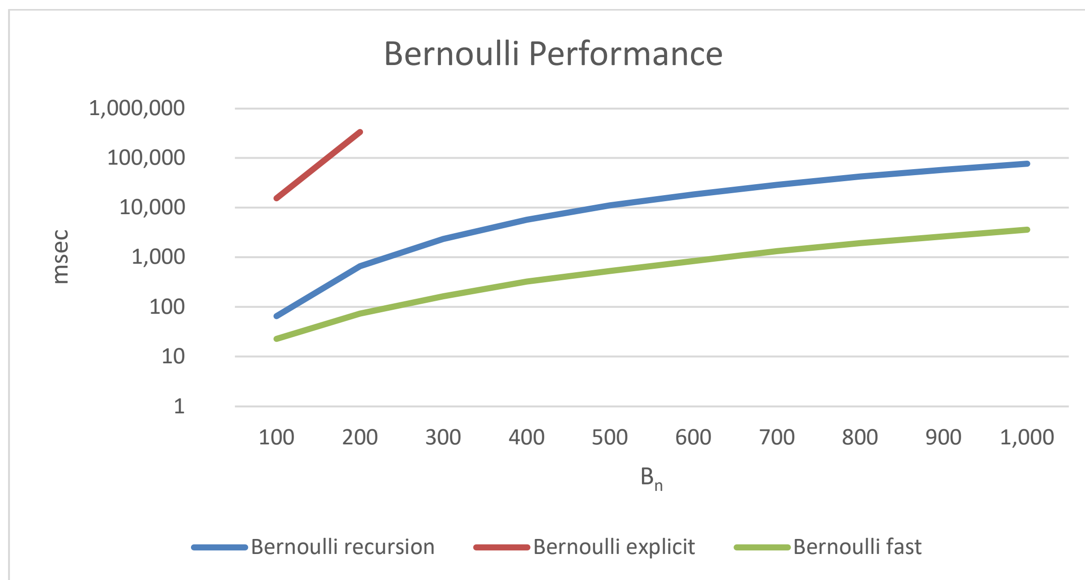


Figure 2 Bernoulli Performance

Recommendation of the Bernoulli method

There is no doubt that the fast calculation of Bernoulli numbers is several magnitudes faster than any of the other methods. I recommend you use that. If simplicity is the most important issue, then use the recursion method for Bernoulli numbers.

Bernoulli Polynomials

Bernoulli polynomials $B(x)_n$ are a natural extension of the Bernoulli numbers B_n. While the Bernoulli numbers represent constant coefficients that appear in many summation formulas, the Bernoulli polynomials form a family of functions that depend on x and are defined by a generating function:

$$\frac{te^{tx}}{e^t-1} = \sum_{n=0}^{\infty} \frac{B(x)_n}{n!} t^n \tag{23}$$

And to recall that the Bernoulli numbers that are defined as shown in the previous subchapter:

$$\frac{t}{e^t-1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} t^n \tag{24}$$

These polynomials occur frequently in numerical analysis, approximation theory, and number theory. They are essential in expressing finite sums, such as power sums of integers, in closed form, and they play a role in the Euler–Maclaurin summation formula, which connects discrete sums with continuous integrals.

Historically, Jakob Bernoulli introduced the related Bernoulli numbers in the late 17th century while studying sums of powers of integers, and the polynomial form soon followed as a natural generalization. In arbitrary-precision computation, Bernoulli polynomials are useful for constructing series expansions of functions such as the Riemann zeta and polylogarithm functions, and for improving convergence in asymptotic series involving Bernoulli terms.

The relationship between Bernoulli polynomials and Bernoulli numbers is:

$$B(x)_n = \sum_{k=0}^{\infty} \binom{n}{k} x^k B_{n-k} \tag{25}$$

Where $\binom{n}{k}$ Is the binomial coefficient. Both the Bernoulli number and Bernoulli polynomials are used in various Taylor series expansions, Riemann's Zeta functions, and many other places.

Source for Bernoulli polynomials

```
float_precision bernoulliPolynomials(const float_precision& x, const size_t n)
{
	const size_t precision = x.precision();
	uintmax_t k;
	fraction_precision<int_precision> ber;
	float_precision xp(1, precision), fp(0, precision), res(0, precision);
	int_precision bin(1);

	for (k = 0; k <= n; ++k)
	{
		if (k > 0)
		{
```

```
                xp *= x;        // update xp=x^k
                // update binomial
                bin = int_precision(n + 1 - k) * bin / int_precision(k);
            }

            ber = bernoulli(n - k);// Get Bernoulli number
            fp = xp;
            fp *= float_precision(bin, precision);
            fp *= float_precision(ber.numerator(), precision);
            fp /= float_precision(ber.denominator(), precision);
            res += fp;
        }

        res.precision(x.precision());
        return res;
}
```

Gamma function

This is one of the most important extensions of the factorial. For positive integers Γ(n) = (n−1)!
It is widely used in probability distributions (normal, gamma, beta distributions), combinatorics, integrals, and differential equations. Mathematicians have studied the Gamma function, Γ(x), since Euler in the 18th century.

The main reason for writing this document was to describe the different methods for the computation of the gamma function. However, some basic prerequisites include fast factorials, binomials, and Bernoulli numbers.

There are several ways to compute the gamma function for various inputs. The general definition for any complex number z with a real positive part is:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \tag{26}$$

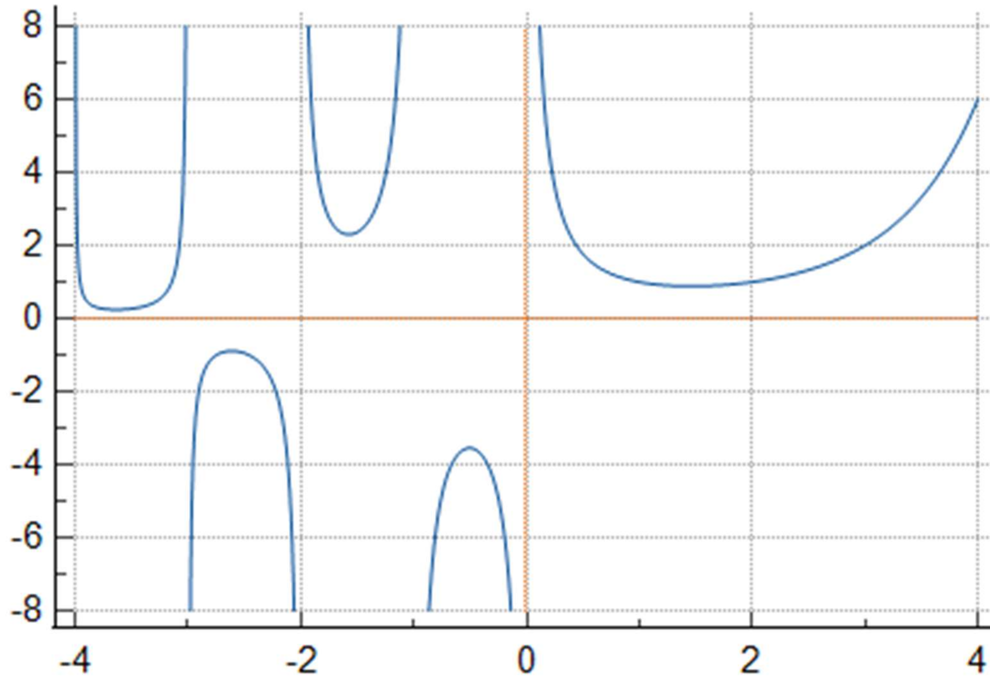


Figure 3. Gamma function in the interval [-4:+4]

The gamma function has several useful identities. E.g., the recurrence relation.

$$\Gamma(z+1) = z\Gamma(z) \tag{27}$$

Given that $\Gamma(1) = 1$ and $\Gamma(n+1) = n\Gamma(n)$ It is easy to see that the Gamma function for any positive integer n is related to the factorial as:

$$\Gamma(n) = (n-1)! \tag{28}$$

There are other useful identities, e.g., for half-integers that:

$$\Gamma\left(\frac{1}{2}+n\right) = \frac{(2n)!}{2^{2n}n!}\sqrt{\pi}\ for\ n \geq 0 \tag{29}$$

Or in the negative half of the real axis:

$$\Gamma\left(\frac{1}{2}-n\right) = (-1)^n \frac{2^{2n}n!}{(2n)!}\sqrt{\pi}\ for\ n > 0 \tag{30}$$

Another important equation is Euler's reflection formula:

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(z\pi)} => \tag{31}$$

$$\Gamma(z) = \frac{\pi}{\Gamma(1-z)\sin(z\pi)} \quad (32)$$

We notice $\Gamma(z)$ Has a pole for z=0 and all negative integer values of z.
The reflection formula can be used to compute the gamma for a complex z in the negative plane by reflecting it into the positive plane. For our computation, we will restrict to the real number x rather than the complex number z. Since we have both specific formulas for positive integer values and both positive and negative half-integer values, we will in general, use the following algorithm:

Algorithm for Gamma computation

1) If x in $\Gamma(x)$ is a positive integer, calculate it directly using factorials.
2) If x in $\Gamma(x)$ is a half-integer in the form $\Gamma\left(\frac{1}{2}+n\right)$ or $\Gamma\left(\frac{1}{2}-n\right)$ then calculate it directly using (29) and (30).
3) If x is negative, use Euler's reflection formula: $\Gamma(x) = \frac{\pi}{\Gamma(1-x)\sin(z\pi)}$ To map x into positive territory.
4) Finally, use one of the approximation methods outlined below.

Algorithm 2

Source for the Half Integer calculation

```
static float_precision gammaHalfinteger(const int_precision& ip, const intmax_t precision)
{
	// For n>=0: T(0.5+n)=(2n)!*sqrt(pi)/(n!*2^(2n))
	// For n<0 : T(0.5-n)=(-1)^n*n!*sqrt(pi)*2^(2n)/(2n)!
	float_precision fp1(0, precision), fp2(0, precision), sqpi(0,precision);
	int_precision ip1(2), ip2(ip);

	sqpi = _float_table(_PI, precision);		// pi
	sqpi = sqrt(sqpi);				// sqrt(pi)
	ip2 = abs(ip2);					// ip2=n
	ip1 *= ip2;					// ip1=2n
	fp1 = float_precision(factorial(ip1), precision);	// (2n)!
	fp2 = float_precision(factorial(ip2), precision);	// n!

	if (ip.sign() > 0)
	{						// Positive n
		fp1 *= sqpi;				// sqrt(pi)*(2n)!
		fp1 /= fp2;				// sqrt(pi)*(2n)!/n!
		fp1.adjustExponent(-(intmax_t)ip1);	// sqrt(pi)*(2n)!/(n!*2^(2n))
		return fp1;
	}
	else
	{						// Negative n
		fp2 *= sqpi;				// sqrt(pi)*n!
		fp2 /= fp1;				// sqrt(pi)*n!/(2n)!
		fp2.adjustExponent(+(intmax_t)ip1);	// sqrt(pi)*n!*2^(2n)/(2n!)
		if (((uintmax_t)ip2) & 0x1) // Odd
			fp2.sign(-1);
		return fp2;
	}
}
```

There exist several methods appropriate for arbitrary precision to compute the gamma function:

- Lanczos-Spouge method
- Stirling asymptotic series method
- Integration by parts method

In general, the Lanczos and Stirling asymptotic methods are global methods, whereas integration by parts is a local method defined on the interval [1:2]. Of course, there are techniques to expand the local method to function as a global method.

Lanczos-Spouge method

The Lanczos method from 1964 was modified by Spouge in 1994, which is a much simpler way to compute $\Gamma(x)$ It is very useful for arbitrary precision arithmetic.
It is normal given in the form:

$$\Gamma(x+1)=\frac{(x+a)^{x+\frac{1}{2}}}{e^{x+a}}\left(c_0+\sum_{k=1}^{a-1}\frac{c_k}{x+k}\right) \tag{33}$$

Where $c_0=\sqrt{2\pi}$ and $c_k=(-1)^{k-1}\frac{(a-k)^{k-\frac{1}{2}}}{(k-1)!e^{k-a}}$

Setting S(x+1)=$c_0+\sum_{k=1}^{a-1}\frac{c_k}{x+k}$ you get: $\Gamma(x+1)=\frac{(x+a)^{x+\frac{1}{2}}}{e^{x+a}}S(x+1)$

And using the recurrence relation: $\Gamma(\mathrm{x})=\frac{\Gamma(x+1)}{x}$ Yields:

$$\Gamma(x)=\frac{(x+a)^{x-\frac{1}{2}}}{e^{x+a}}S(x) \tag{34}$$

For some value of a. The variable a can be set to any arbitrary value and controls the maximum error in the calculation. In [3], they found that the lowest value to compute P correct digits in the calculation above was estimated to:

$$a=\left\lceil\left(P-\frac{\ln(P)}{\ln(10)}\right)\frac{\ln(10)}{\ln(2\pi)}-\frac{1}{2}\right\rceil \tag{35}$$

To avoid underestimating a, they use $\frac{659}{526}$ *instead of* $\frac{\ln(10)}{\ln(2\pi)}$.
Now, since c_k has an alternating sign, [3] further found that, to avoid cancellation errors when calculating c_k, the working precision of c_k needed to be 1.1515P. In my opinion, it is not enough to preserve the accuracy, so I use 1.5P instead.

In [5], instead of finding a, from the wanted precision, they state that the error is bounded by:

$$e_a(x) = \frac{1}{a^{0.5}(2\pi)^{a+0.5}} \tag{36}$$

For a given precision *P*, the variable *a* is:

Precision=	10	100	1,000	10,000	100,000	1,000,000
a=	11	122	1,249	12,523	125,278	1,252,842

Lanczos-Spouge Approximation	
Pros	**Cons**
Accuracy is easy to control and maintain	Need to compute π, e^x, and √
No need to shift/de-shift the Gamma value	
Fast Method	

Source of Lanczos-Spouge method.

```
static float_precision gammaLanczosSpouge(const float_precision& x)
{
	const intmax_t precision = x.precision()+8;
	const intmax_t workprec = 150 * precision / 100;// original 116, work for 150
	const float_precision c1(1);
	intmax_t a;
	float_precision fp(0,precision), fpip(0,precision), pi(0, precision), ekm1(0,
workprec), ck(0, workprec), sum(0, workprec);

	fp = modf(x, &fpip);
	if (fp.iszero()) // x is an integer
	{
		if(fpip<=float_precision(0))  // if integer <= 0 then throw domain error
			throw float_precision::domain_error();
		return factorial((int_precision(fpip)-int_precision(1)));
	}
	if (abs(fp) == float_precision(0.5))  // x is a half-integer
	{	// For n>=0: T(0.5+n)=(2n)!*sqrt(pi)/(n!*2^(2n))
		// For n<0 : T(0.5-n)=(-1)^n*n!*sqrt(pi)*2^(2n)/(2n)!
		// Notice modf will deliver -0.5 as fpip==-0 and fp==-0.5
		if (x.sign() < 0)
			fpip -= 1;	// Ensure if negative than the form is 0.5-n form
		return gammaHalfinteger(int_precision(fpip), precision);
	}
	// x is a regular floating-point variable
	// if T(x<0) then use Euler reflection formula T(x)=pi/(T(1-x)*sin(pi*x))
	if (x.sign() < 0)
	{
		fp = gammaLanczosSpouge(c1 - x);
		pi = _float_table(_PI, precision);	// pi
		fp *= sin(pi * x);
		fp = pi / fp;
		return fp;
	}

	// Use Lanczos-Spouge formula for x>0
	// T(x)=(x+a)^(x+0.5)/e^(x+a))*(sqrt(2PI)+sum([k=1,N]ck/(x+k)))
	// where ck=(-1)^(k-1)*(a-k)^(k-0.5)/((k-1)!*e^(k-a))
	// Step 1 determine a as (P-ln(P)/ln(10))*(ln(10)/ln(2PI)-0.5==
	// (P-ln(P)/ln(10))*(659/526)-0.5 to ensure a is not underestimated
	a = (intmax_t)ceil( 659 * (precision - log(precision) / log(10)) / 526 - 0.5 );
	fp = x;

	// Step 2 calculate the sum, doing it backward (smallest to largest value)
	sum = float_precision(0);
	size_t facprec = factorialprecision(a - 1);
```

```
        for (intmax_t k = a-1; k > 0; --k)
        {
                // Use (a-k)^(k-0.5)=sqrt(a-k)^(k-1), where ipow() is used
                // for increasing performance
                // The regular pow() for float_precision use both ln() and exp()
                // which is much slower
                ck = sqrt(float_precision(a-k, workprec));
                ck *= float_precision(ipow(int_precision(a-k),int_precision(k-1)),workprec);
                ekm1 = exp( float_precision(k-a,workprec));
                ekm1 *= float_precision(factorial(int_precision(k-1)),facprec);
                ck /= ekm1;
                if ((k - 1) & 0x1)  // Odd
                        ck.sign(-1);
                ck /= (fp + float_precision(k));
                sum += ck;
        }

        // Step 3 calculate sqrt(2PI)
        pi = _float_table(_PI, precision);    // PI
        pi.adjustExponent(+1);                // 2PI
        pi = sqrt(pi);                        // sqrt(2PI)

        // Step 4 finalize the tailing part of T(x)
        sum += pi;

        // Step 5 Finalizing the head part of T(x)
        sum *= pow(fp + float_precision(a), fp - float_precision(0.5));
        fp = exp(fp+ float_precision(a));
        fp *= sum;
        fp.precision(x.precision());
        return fp;
}
```

Stirling asymptotic series method

Stirling asymptotic series for the Gamma function is given by:

$$\ln\bigl(\Gamma(x)\bigr) \sim \left(x-\tfrac{1}{2}\right)\ln(x) - x + \tfrac{1}{2}\ln(2\pi) + \sum_{n=1}^{\infty}\frac{B_{2n}}{2n(2n-1)x^{2n-1}} \qquad (37)$$

Where B_{2n} is the Bernoulli number. In [3], they find that the optimal value for n in the summation is given by:

$$n_{optimal} \sim \pi|x|+2 \qquad (38)$$

Furthermore, they state that to reach the needed precision *P,* the following equations need to hold:

$$(2\pi-1)x + (x+1)ln(x) + 3.9 \ > \ P_{max} \cdot ln(10) => \qquad (39)$$

$$P_{max} = \left\lfloor \frac{(2\pi-1)x+(x+1)ln(x)+3.9}{\ln(10)} \right\rfloor \qquad (40)$$

For a certain magnitude of |x| we get:

|x|=	1	10	100	1,000	10,000	100,000
P_{max}=	3	35	433	5,299	62,950	729,452

This is discouraging, since for small |x| we cannot achieve reasonable precision with this method. To circumvent this deficit, we can use the recurrence. $x\Gamma(x) = \Gamma(x+1)$, M number of times to increase the magnitude. E.g., if |x| is 1 and we need the result with 35 digits, the magnitude of |x| must be greater than 10 in the table above. Instead of calculating $\Gamma(x)$, we calculate $\Gamma(x+10)$ And then divide $\Gamma(x+10)$ 10 times as outlined:

$$\Gamma(x) = \frac{\Gamma(x+10)}{x(x+1)(x+2)(x+3)(x+4)(x+5)(x+6)(x+7)(x+8)(x+9)} \quad (41)$$

In general, if you shift it a distance of M, you can write this as:

$$\Gamma(x) = \frac{\Gamma(x+M)}{\prod_{m=0}^{M-1}(x+m)} \quad (42)$$

Unfortunately, the above formula for P_{max} is not very accurate in determining the number of shifts needed and, in general, indicates insufficient shifting for the desired accuracy. Instead, we use [6], which states the number of shifts needed as follows.

$$|x + shifts| = P * \frac{\ln(10)}{\ln(2)} * 0.11038 => \quad (43)$$

$$shifts = P * \frac{\ln(10)}{\ln(2)} * 0.11038 - |x| \quad (44)$$

This formula works both ways. If |x| is small, the shift is positive. If |x| is large, the shift is negative. This is a huge benefit that we can use it both ways to reduce the argument and thereby reduce the number of terms needed for the $\sum$ (also reducing the number of Bernoulli numbers we need to compute).

Stirling Asymptotic Method	
Pros	**Cons**
Accuracy is excellent for a large magnitude of |x|	Poor Accuracy for the small magnitude of |x|
De-shifting is beneficial for large |x|	Need to compute π, e^x, ln() and √
	Need to compute Bernoulli numbers.
	Need to shift gamma value for small magnitude |x|
	Slow Computation

Source of the Stirling asymptotic method

```
static float_precision gammaStirling(const float_precision& x)
{
	const intmax_t precision = x.precision()+8;
	const intmax_t workprec =  150 * precision / 100;
	const float_precision c1(1);
	intmax_t nmax, shifts;
	float_precision fp(0,precision), fp2(0,precision), pi(0,precision), sum(0,
precision);

	fp = modf(x, &fp2);
```

```
        if (fp.iszero()) // x is an integer
        {
                if (fp2 <= float_precision(0))  // if integer <= 0 then throw domain error
                        throw float_precision::domain_error();
                return factorial((int_precision(fp2) - int_precision(1)));
        }
        if (abs(fp) == float_precision(0.5))  // x is a half-integer
        {       // For n>=0: T(0.5+n)=(2n)!*sqrt(pi)/(n!*2^(2n))
                // For n<0 : T(0.5-n)=(-1)^n*n!*sqrt(pi)*2^(2n)/(2n)!
                // Notice modf will deliver -0.5 as fpip==-0 and fp==-0.5
                if (x.sign() < 0)
                        fp2 -= 1;       // Ensure if negative than the form is 0.5-n form
                return gammaHalfinteger(int_precision(fp2), precision);
        }
        // x is a regular floating-point variable
        // if T(x<0) then use Euler reflection formula T(x)=pi/(T(1-x)*sin(pi*x))
        if (x.sign() < 0)
        {
                fp = gammaStirling(c1 - x);
                pi = _float_table(_PI, x.precision()); // pi
                fp *= sin(pi * x);
                fp = pi / fp;
                return fp;
        }

        // Use the Stirling Asymptotic method
        // Step 1
        // calculate how many shifts are needed comparing magnitude to precision precision
        // ensuring shifting = precision*ln(10/ln(2)*0.11038 - x
        // 0.110318 ~ log(2)/2pi but can be increased to between 0.17-0.24 as a
        // tuning parameter
        // This works both ways if the magnitude is already large then we do negative shifts
        // to avoid unnecessary calculations in the
        double fx = (double)x;
        shifts = (int)ceil(precision * log(10) / log(2) * 0.110318-fx);
        fp2 = x+float_precision(shifts);              // set shifted x

        // Step 2 calculate N0 and the sum
        pi = _float_table(_PI, precision);            // pi
        nmax = (intmax_t)(pi * abs(fp2));             // max number of summation
        // Do the sum from smallest to the largest number
        for (intmax_t n = nmax; n > 0; --n)
        {
                intmax_t k = 2 * n;
                fp = float_precision(k*(k - 1));      // 2k(2k-1)
                fp *= pow(fp2, float_precision(k - 1));// 2k(2k-1)x^(2k)
                fp = bernoulli(k,workprec)/fp;        // Bernoulli(2k)/2k(2k-1)x^(2k)
                sum += fp;                            // Accumulate in sum
        }

        // Step 3 calculate the final ln(T(x)) and thereafter T(x)
        pi.adjustExponent(+1);          // 2PI
        fp=log(pi);                     // log(2PI)
        fp.adjustExponent(-1);          // 0.5*log(2pi)
        fp += sum;                      // 0.5*log(2pi)+sum
        fp -= fp2;                      // -x+0.5 * log(2pi)+sum
        fp += (fp2-float_precision(0.5))*log(fp2); // ln(T(x))=(x-0.5)ln(x)-x+0.5 *
log(2pi)+sum
        fp = exp(fp);                   // T(x)

        // Step 4 readjust for shifted T(x)
        if (shifts < 0)
        {
                for (fp2 = c1; shifts < 0; ++shifts)
                        fp2 *= x + float_precision(shifts);
                fp *= fp2;
        }
        else
                if (shifts > 0)
```

```
		{
			for (fp2 = c1; shifts > 0; --shifts)
				fp2 *= x + float_precision(shifts - 1);
			fp /= fp2;
		}

	fp.precision(x.precision());
	return fp;
}
```

Optimizing the Stirling Gamma method

The above method is solid, generating the gamma value for a given input. However, it is not the best performance algorithm. Instead, we notice that in the algorithm, we can adjust both the shifting amount and the number of needed Bernoulli numbers. Increasing the shift value reduces the number of Bernoulli numbers required. And since shifting is a relatively inexpensive operation compared with the cost of computing the Bernoulli numbers, we can significantly improve the performance of the Stirling method for Gamma computation.

Instead of the equation for shifts, we can use a more aggressive shifting using the formula.

Shifts target = (P + ln(P)) · ln(10) · 1.2 (20% more aggressive)

And then compute the actual shifting by subtracting x: `shifts -= integer(x)`

For the number of Bernoulli terms, the Stirling remainder is of: $O(\frac{1}{x^{2N+1}})$ to get the error below the precision limit of: error < 10^{-P}.

We get (2N+1)·ln(x) > P·ln(10) => $N > P \cdot \frac{ln(10)}{(2 \cdot ln(x))} - 0.5$ And add an extra 50% safety margin for N.

With these two heuristic values for shifting and N Bernoulli terms, we have the following new, much higher-performing source.

Higher Performance source of the Stirling asymptotic method

```
// Calculate ln(T(x))=(x-0.5)ln(x)-x+0.5 * log(2pi)+sum
// Where sum is sum(k=1,nmax](Bernoulli(2k)/(2k*(2k-1)*x^(2k-1)))
// then finally calculate T(x)=exp(ln(T(x)))
static float_precision gammaStirling(const float_precision& x)
{
	const intmax_t precision = x.precision();
	const intmax_t base_guard = (intmax_t)ceil(log10((double)precision)) + 8;
	const intmax_t workprec =  precision + base_guard + precision / 2;
	const float_precision c1(1);
	float_precision fp(0,workprec), fp2(0,workprec), sum(0, workprec);

	fp = modf(x, &fp2);
	if (fp.iszero()) // x is an integer
	{
		if (fp2 <= float_precision(0))  // if integer <= 0 then throw domain error
			throw float_precision::domain_error();
		return float_precision(factorial((int_precision(fp2) - int_precision(1))));
	}
	if (abs(fp) == float_precision(0.5))  // x is a half integer
	{	// For n>=0: T(0.5+n)=(2n)!*sqrt(pi)/(n!*2^(2n))
		// For n<0 : T(0.5-n)=(-1)^n*n!*sqrt(pi)*2^(2n)/(2n)!
```

```
            // Notice modf will delivered -0.5 as fpip==-0 and fp==-0.5
            if (x.sign() < 0)
                    fp2 -= 1;       // Ensure if negative than the form is 0.5-n form
            return gammaHalfinteger(int_precision(fp2), precision+base_guard);
        }
        // x is a regular floating-point variable
        // if T(x<0) then use Euler reflection formula T(x)=pi/(T(1-x)*sin(pi*x))
        if (x.sign() < 0)
        {
            fp = gammaStirling(c1 - x);
            float_precision pi(_float_table(_PI, precision));     // pi
            fp *= sin(pi * x);
            fp = pi / fp;
            return fp;
        }

        // Use the Stirling Asymptotic method
        // Step 1
        // calculate how many shifts are needed comparing magnitude to precision
        // This works both ways if the magnitude is already large, then we do negative
        // shifts to avoid unnecessary calculation in the
        // Calculate shift target - use more aggressive formula for high precision
        // At high precision, we need x to be even larger relative to precision
        //double log_prec = log(double(precision));
        // Use: M = (P + ln(P)) * ln(10) * 1.2  (20% more aggressive)
        intmax_t shifts = static_cast<intmax_t>(ceil((precision + log(precision)) *
log(10.0) * 1.2));
        shifts = std::max(20ll, shifts);

        if (x < float_precision(shifts))
            shifts = shifts - static_cast<intmax_t>(x);
        fp2 = x+float_precision(shifts);                // set shifted x, fp2 is now the x for
the gamma computation
        const intmax_t working_precision = precision + (precision <= 400 ? base_guard :
precision * 3 / 10);

        // Step 2 calculate the needed number of Bernoulli's correction terms
        // The original nmax = (intmax_t)(pi * abs(fp2)); // max number of summations
        // was too conservative
        // Stirling remainder: O(1/x^(2N+1))
        // For error < 10^(-precision): (2N+1)*ln(x) > precision*ln(10)
        // So: N > precision*ln(10)/(2*ln(x)) - 0.5
        // Add 50% safety margin for high precision
        // This ensures Stirling converges properly at all precision levels
        // Use proportional working precision (50% extra)
        double fx = double(fp2); // Avoid computing log(fp2) in arbitrary precision
        double nterms_exact = (working_precision * log(10.0)) / (2.0 * log(fx)) - 0.5;
        intmax_t nterms = static_cast<intmax_t>(ceil(nterms_exact * 1.5));  // 50% safety
margin
        // Minimum terms for stability
        nterms = std::max(10ll, nterms);      // Minimu, 10 bernoulli terms

        // Step 3 Do the Bernoulli sum from smallest to largest number
        for (intmax_t n = nterms; n > 0; --n)
        {
            intmax_t k = 2 * n;
            fp = float_precision(k*(k - 1));                // 2k(2k-1)
            fp *= pow(fp2, float_precision(k - 1));         // 2k(2k-1)x^(2k)
            fp = bernoulli(k,workprec)/fp;                  // Bernoulli(2k)/2k(2k-
1)x^(2k)
            sum += fp;                                                          //
Accumulate in sum
            fx = (double)fp;                                                // DEBUG
        }

        // Step 4 calculate the final ln(T(x)) and thereafter T(x)
        //fp.precision(workprec);
        //fp2.precision(workprec);
        float_precision pi(_float_table(_PI, precision));           // pi
```

```
        pi.adjustExponent(+1);                  // 2PI
        fp=log(pi);                             // log(2PI)
        fp.adjustExponent(-1);                  // 0.5*log(2pi)
        fp += sum;                              // 0.5*log(2pi)+sum
        fp -= fp2;                              // -x+0.5 * log(2pi)+sum
        fp += (fp2-float_precision(0.5))*log(fp2); // ln(T(x))=(x-0.5)ln(x)-x+0.5 *
log(2pi)+sum
        fp = exp(fp);                           // T(x)

        // Step 5 readjust for shifted T(x)
        if (shifts < 0)
        {
                for (fp2 = c1; shifts < 0; ++shifts)
                        fp2 *= x + float_precision(shifts);
                fp *= fp2;
        }
        else
                if (shifts > 0)
                {
                        for (fp2 = c1; shifts > 0; --shifts)
                                fp2 *= x + float_precision(shifts - 1);
                        fp /= fp2;
                }

        fp.precision(x.precision());
        return fp;
}
```

Integration by parts method

A third method is the so-called Integration by parts method, which, for x in [1:2], you can apply to Euler's integral. The integral can be written as:

$$\Gamma(x) = \int_0^M t^x e^{-t}\frac{dt}{t} + \int_M^\infty t^x e^{-t}\frac{dt}{t} \tag{45}$$

The first integral is the lower incomplete gamma function, and the second integral is the upper incomplete gamma function. You can choose M so that the second integral is below the wanted precision of 10^{-P}, where P is the precision in decimal digits. The second integral can therefore be ignored. Then $\Gamma(x)$ Becomes:

$$\Gamma(x) \approx \int_0^M t^x e^{-t}\frac{dt}{t} = M^x e^{-M}\sum_{n=0}^{\infty}\frac{M^n}{\prod_{k=0}^{n}(x+k)} \tag{46}$$

Now the question is how to choose an appropriate M. In [3], they find the condition to be that:

$$M > (P + \ln(P))\ln(10) \tag{47}$$

The only thing missing now is the number of terms (N_{max}) of the series you need to calculate. Again, in [3], they find that to be:

$$N_{max} = P\frac{\ln(10)}{W(\frac{1}{e})}\text{, where W is Lambert's function } W\left(\frac{1}{e}\right) \approx 0.2785 \tag{48}$$

With that, we have the final formula:

$$\Gamma(x) \approx M^x e^{-M} \sum_{n=0}^{N_{max}} \frac{M^n}{\prod_{k=0}^{n}(x+k)} \quad (49)$$

The only issue left to fix is the condition that x should be in the interval of $1 \leq x \leq 2$

We can use the same shifting technique described earlier to map all x outside the interval into the interval [1-2], e.g., if x> 2, you use.

$$\Gamma(x) = \prod_{m=0}^{k-1}(x-m)\Gamma(x-k) \quad (50)$$

E.g., is x=5.6 then k=4. If x=0.6 then $\Gamma(0.6) = \frac{\Gamma(0.6+1)}{\prod_{m=0}^{0}(x+m)}$

If x is negative, you can use Euler's reflection formula described earlier to map x into a positive number.

Integration by parts method	
Pros	**Cons**
Fast method	Only works in the interval [1-2]
Simplicity	Use of e^x
	Need to shift/de-shift gamma value outside the interval [1-2]

Source Integration by parts

```
static float_precision gammaByParts(const float_precision& x)
{
	const intmax_t precision = x.precision() + 8;
	intmax_t powprec=precision;
	const float_precision c1(1), c2(2);
	intmax_t M, n, nmax, shifts;
	float_precision fp(0, precision), fp2(0, precision),fp3(0,precision), sum(0, 
precision);

	fp = modf(x, &fp2);
	if (fp.iszero()) // x is an integer
	{
		if (fp2 <= float_precision(0))  // if integer <= 0 then throw domain error
			throw float_precision::domain_error();
		return factorial((int_precision(fp2) - int_precision(1)));
	}
	if (abs(fp) == float_precision(0.5))  // x is a half-integer
	{	// For n>=0: T(0.5+n)=(2n)!*sqrt(pi)/(n!*2^(2n))
		// For n<0 : T(0.5-n)=(-1)^n*n!*sqrt(pi)*2^(2n)/(2n)!
		// Notice modf will be delivered -0.5 as fpip==-0 and fp==-0.5
		if (x.sign() < 0)
			fp2 -= 1;	// Ensure if negative than the form is 0.5-n form
		return gammaHalfinteger(int_precision(fp2), precision);
	}
	// x is a regular floating-point variable
	// if T(x<0) then use Euler reflection formula T(x)=pi/(T(1-x)*sin(pi*x))
	if (x.sign() < 0)
	{
		float_precision pi(0, precision),
		fp = gammaByParts(c1 - x);
		pi = _float_table(_PI, x.precision()); // pi
		fp *= sin(pi * x);
		fp = pi / fp;
		return fp;
```

```
	}

	// Use the Integration by parts method
	// First integral as the sum of the Taylor series exp(-u) near u==0
	// integral[0,M]u^(x-1)*exp(-u)du=M^x*sum([n=0,M](-1)^n*M^n/(n+x)*n1)
	// Step 1 choose M>(P+ln(P))*ln(10). Due to the alternating sign working
	// precision needs to be 2P
	M = static_cast<uintmax_t>(ceil((precision + log(precision)) * log(10)));

	// Step 2
	// calculate how many shifts is needed to bring x within [1-2]. X
	// can in the range 0<x<inf
	shifts = 0;			// Default for 1<=x<=2
	if (x < c1) shifts = +1;	// x<1 set shifts to 1;
	else
		if (x > c2)		// if x>2 then set shifts to -floor(x)+1
			shifts = -static_cast<intmax_t>(fp2) + 1;
	fp = x+float_precision(shifts);// set shifted x

	// Step 3 calculate the series sum
	nmax = static_cast<uintmax_t>(ceil(precision * log(10) / 0.2785));
	powprec = (size_t)ceil(nmax * log10(M));
	for (n = 0, fp2=c1, fp3=c1; n <= nmax; ++n, fp3*=float_precision(M))
	{
		fp2 *= (fp + float_precision(n));
		sum += fp3 / fp2;
	}

	// Step 4 finalize the gamma value
	sum *= exp(float_precision(-M,precision));
	fp = pow(float_precision(M,precision), fp)*sum;

	// Step 5 readjust for any shifted T(x)
	if (shifts < 0)
	{
		for (fp2 = c1; shifts < 0; ++shifts)
			fp2 *= x + float_precision(shifts);
		fp *= fp2;
	}
	else
		if(shifts>0)
		{	// Max 1 shifts
			fp2 = x;
			fp /= fp2;
		}

	fp.precision(x.precision());
	return fp;
}
```

Gamma Performance

The graph below shows that the best-performing method is integration by parts. It consistently performs better than Stirling and the Lanczos-Spouge method. However, it clearly shows that the method is the slowest when Bernoulli numbers are not pre-calculated. The Stirling method with cache Bernoulli numbers is 30-40% faster than without caching.

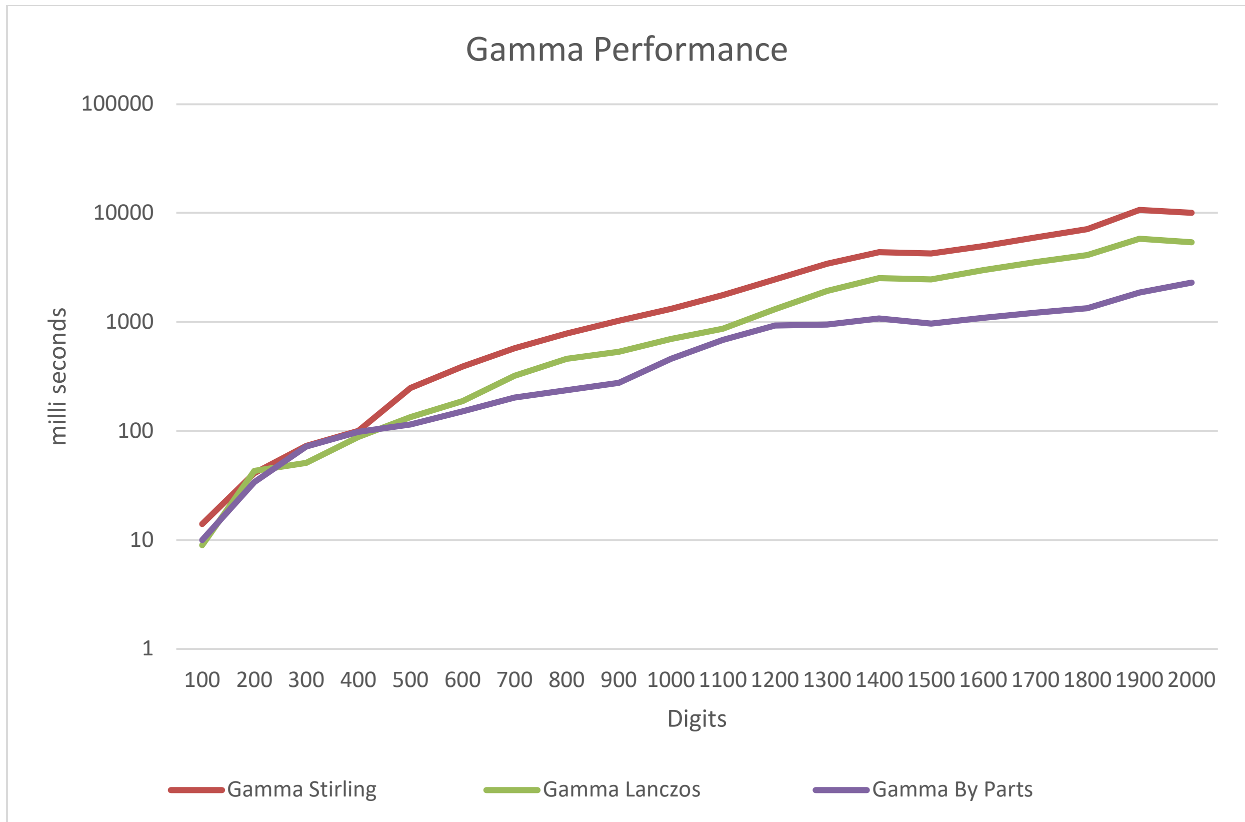


Figure 4. Gamma Performance

Recommendation for the Gamma function

The method with integration by parts is the simplest and fastest, even when it relies on the shifting technique to accommodate the x-value outside the interval [1:2]. In second place is the Lanczos-Spouge method, and third is the Stirling asymptotic method. Because the Stirling Asymptotic method relies on the Bernoulli numbers, it is not recommended unless the Bernoulli numbers are precomputed, which is impractical in arbitrary-precision arithmetic.

Log Gamma function

$\Gamma(x)$ grows very fast. Taking the logarithm avoids overflow, improves numerical stability, and simplifies expressions where gamma values appear as products or ratios. It is a standard function in statistical computing.

Why compute log gamma

In many problems, the useful quantity is the natural logarithm of the gamma function, not $\Gamma(x)$ itself: In the C++ library, the logarithm of the gamma function (tgamma()) is called lgamma().

$$\text{lgamma}(x) = \log(|\Gamma(x)|).$$

There are three main reasons to compute the logarithm of the gamma function.
First, Γ(x) grows very fast. Taking the logarithm keeps values in a safe numerical range and avoids overflow or underflow.
Second, gamma functions appear frequently in formulas, for example, in beta functions, where:

$$Beta(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} \tag{51}$$

where ln(Beta(x,y) becomes ln(Γ(x)+ln(Γ(y)-ln(Γ(x+y).
in binomial coefficients,

$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} \tag{52}$$

factorial expressions $\frac{\Gamma(n+k)}{\Gamma(n)}$ Arises in many series and recursion formulas, as well as in many statistical distributions.

Working in logarithmic form converts these products and ratios into sums and differences of ln Γ(·), which are simpler and numerically more stable.
Third, negative non-integer arguments are handled naturally through the reflection identity for the log-gamma function.

$$\Gamma(x)\cdot\Gamma(1-x) = \pi / \sin(\pi x) \;\Rightarrow\; \ln(|\Gamma(x)|) = \log(\pi) - \log(|\sin(\pi x)|) - \ln(|\Gamma(1-x)|) \tag{53}$$

Working in log space keeps this reduction stable, since we never produce large or small intermediate values.

A short note on the C and C++ interfaces

Older C libraries shipped a function named gamma, but some returned the true gamma function, and others returned its logarithm. To remove the ambiguity, the C standard introduced tgamma for the true gamma function and kept lgamma for the natural logarithm of the absolute value of Γ. C++ follows this convention in <cmath>.

- std::tgamma(x) returns Γ(x).
- std::lgamma(x) returns log(|Γ(x)|).

Three practical ways to compute ln(|Γ(x)|)

1) Take the logarithm of Γ directly

If you already have a reliable routine for Γ(x), the simplest path is to call it and take the logarithm: In C++ style, this is:

lgamma(x) = log(|tgamma(x)|).

This is easy to code, but it is only safe on a limited range. If Γ(x) overflows, the logarithm cannot recover, and if Γ(x) underflows to zero, the result becomes minus infinity. Close to poles, minor relative errors in Γ(x) are amplified by the log.

2) Stirling series for log gamma

A very effective route is to approximate log Γ(x) directly with an asymptotic series and to shift the argument into a convenient band using the recurrence log Γ(x+1) = log Γ(x) + log x. See the Stirling asymptotic series for the Gamma function. For a positive x that is not too small, we can write.

$$\ln\big(\Gamma(x)\big) \sim \left(x-\frac{1}{2}\right)\ln(x) - x + \frac{1}{2}\ln(2\pi) + \sum_{n=1}^{\infty}\frac{B_{2n}}{2n(2n-1)x^{2n-1}} \qquad (54)$$

Where B_{2n} are Bernoulli numbers, accuracy improves as x grows, and a few terms often suffice after a slight shift that maps x into the interval [1, 2). For a negative non-integer x, use the reflection identity to reduce to the positive side.

Stirling Asymptotic Method	
Pros	**Cons**
Accuracy is excellent for a large magnitude of |x|	Poor Accuracy for the small magnitude of |x|
De-shifting is beneficial for large |x|	Need to compute π, e^x, ln(), and √
	Need to compute Bernoulli numbers.
	Need to shift gamma value for small magnitude |x|
	Slow Computation

Source Stirling log gamma functions.

```
static float_precision logFactorial(const int_precision& n, const intmax_t precision)
{
	if (n <= int_precision(0))
		return float_precision(0, precision);  // ln(0!) = 0

	// Use the exact factorial function
	int_precision fact_n = factorial(n);

	// Convert to float_precision and take log with working precision
	float_precision result(0,precision);
	result = log(float_precision(fact_n, precision));
	return result;
}

/// Calculate ln(gamma()) when argument is in half integer form
///	For n>=0: ln(Γ(0.5+n)) = ln((2n)!) + 0.5*ln(π) - ln(n!) - (2n)*ln(2)
///	For n<0 : ln(Γ(0.5-n)) = ln(n!) + 0.5*ln(π) + (2n)*ln(2) - ln((2n)!)
///
static float_precision lgammaHalfinteger(const int_precision& ip, const intmax_t precision)
{
	// Rigorous working precision: guard digits = 2*log10(P) + 10
	const double log10_prec = log10((double)precision);
	const intmax_t guard_digits = static_cast<intmax_t>(ceil(2.0 * log10_prec + 10));
	const intmax_t working_precision = precision + guard_digits;

	float_precision result(0, working_precision), sqpi(0, working_precision);
	int_precision ip2(ip);
	const float_precision ln2(log(float_precision(2, working_precision)));

	sqpi = _float_table(_PI, working_precision);		// pi
	sqpi = log(sqpi) * float_precision(0.5);			// 0.5*ln(pi)
	ip2 = abs(ip2);
	// ip2=|n|
	int_precision ip1 = ip2 * int_precision(2);			// ip1=2|n|
```

```
        // Use log(n!)
        float_precision lnfact_2n(logFactorial(ip1, working_precision));  // ln((2n)!)
        float_precision lnfact_n(logFactorial(ip2, working_precision));   // ln(n!)

        if (ip.sign() > 0)
        {
                // ln(Γ(0.5+n)) = ln((2n)!) + 0.5*ln(π) - ln(n!) - (2n)*ln(2)
                result = lnfact_2n + sqpi - lnfact_n - float_precision(ip1,
working_precision) * ln2;
        }
        else
        {
                // ln(Γ(0.5-n)) = ln(n!) + 0.5*ln(π) + (2n)*ln(2) - ln((2n)!)
                result = lnfact_n + sqpi + float_precision(ip1, working_precision) * ln2 -
lnfact_2n;
        }

        result.precision(precision);
        return result;
}

/// Stirling's asymptotic expansion:
/// ln(Γ(x)) = (x-0.5)*ln(x) - x + 0.5*ln(2π) + Σ B_{2k}/(2k*(2k-1)*x^(2k-1))
/// where B_{2k} are Bernoulli numbers
///
/// Note: Requires x to be large relative to precision for good convergence
///
static float_precision stirlingLgamma(const float_precision& x, const intmax_t precision,
const intmax_t nterms )
{
        // Main Stirling formula: (x-0.5)*ln(x) - x + 0.5*ln(2π)
        const float_precision pi = _float_table(_PI, precision);
        const float_precision half(0.5);
        float_precision result = (x - half) * log(x) - x;
        result += half * log(float_precision(2) * pi);

        // Add Bernoulli correction terms
        const float_precision x2 = x * x;
        float_precision xpow(x);  // Will hold x^(2k-1)
        // Extra precision for Bernoulli numbers (proportional)
        const intmax_t bernoulli_prec = precision + intmax_t(ceil(log10(precision))) + 6;//
work_prec / 5;

        // Compute Bernoulli terms
        for (intmax_t k = 1; k <= nterms; ++k)
        {
                // Term is: B_{2k} / (2k * (2k-1) * x^(2k-1))
                intmax_t two_k = 2 * k;
                // Get Bernoulli number with extra precision
                const float_precision B_2k = bernoulli(two_k, bernoulli_prec);
                // Compute the term
                const float_precision denom = float_precision(two_k * (two_k - 1),
precision) * xpow;
                const float_precision term = B_2k / denom;
                result += term;
                // Prepare for next iteration
                xpow *= x2;
        }
        return result;
}

/// Calculate ln(Γ(x)) efficiently without computing Γ(x) itself
///
/// Method:
/// - For integers: Uses ln(n!) = ln(Γ(n+1))
/// - For half-integers: Uses analytical formula
/// - For negative x: Uses reflection formula
/// - For positive x: Uses argument reduction + Stirling's series with Bernoulli numbers
```

```
float_precision lgamma(const float_precision& x)
{
	const intmax_t precision = x.precision();
	const float_precision c1(1), c2(2);
	float_precision fp(0, precision), fp2(0, precision);

	fp = modf(x, &fp2);

	// Handle integers
	if (fp.iszero())
	{
		if (fp2 <= float_precision(0))  // if integer <= 0 then throw domain error
			throw float_precision::domain_error();
		int_precision nip = int_precision(fp2);
		nip -= int_precision(1);
		return logFactorial(nip, precision);
	}

	// Handle half-integers
	if (abs(fp) == float_precision(0.5))
	{
		if (x.sign() < 0)
			fp2 -= c1;  // Ensure negative form is 0.5-n
		int_precision nip = int_precision(fp2);
		return lgammaHalfinteger(nip, precision);
	}

	const intmax_t base_guard = (intmax_t)ceil(log10((double)precision)) + 8;

	// Handle negative x using reflection formula
	// ln(Γ(x)) = ln(π) - ln(|sin(πx)|) - ln(Γ(1-x))
	if (x.sign() < 0)
	{
		// Rigorous working precision calculation
		double log10_prec = log10((double)precision);
		intmax_t guard_digits = static_cast<intmax_t>(ceil(2.0 * log10_prec + 10));
		const intmax_t work_prec = precision + base_guard; // guard_digits;
		float_precision pi(_float_table(_PI, work_prec));
		float_precision x_work(x);
		x_work.precision(work_prec);
		float_precision sinpix(0, work_prec);
		sinpix = abs(sin(pi * x_work));
		float_precision result(0, x.precision());
		result = log(pi) - log(sinpix) - lgamma(c1 - x);
		return result;
	}

	// Handle positive x
	// For positive x: use argument reduction to shift x large enough for Stirling
	// Use similar shift formula as tgamma: M = (P + ln(P)) * ln(10)
	// This ensures Stirling converges properly at all precision levels
	// Use proportional working precision (30% extra)
	const intmax_t working_precision = precision + (precision<=400?base_guard: precision
* 3 / 10);
	// x_work is x with an increased working precision
	float_precision x_work(x);
	x_work.precision(working_precision);

	// Calculate shift target - use more aggressive formula for high precision
	// At high precision, we need x to be even larger relative to precision
	double log_prec = log(double(precision));
	// Use: M = (P + ln(P)) * ln(10) * 1.2  (20% more aggressive)
	intmax_t shifts = static_cast<intmax_t>(ceil((precision + log_prec) * log(10.0) *
1.2));// 1.0 is not OK 1.2 is
	if (shifts < 20) shifts = 20;
	if (x_work < float_precision(shifts))
		shifts = shifts - static_cast<intmax_t>(x_work);
	float_precision x_shifted(0, working_precision);
	x_shifted = x_work + float_precision(shifts, working_precision);
```

```
        // Determine number of Bernoulli correction terms needed
        // Stirling remainder: O(1/x^(2N+1))
        // For error < 10^(-precision): (2N+1)*ln(x) > precision*ln(10)
        // So: N > precision*ln(10)/(2*ln(x)) - 0.5
        // Add 50% safety margin for high precision
        double dx = double(x_shifted);
        double nterms_exact = (working_precision * log(10.0)) / (2.0 * log(dx)) - 0.5;
        intmax_t nterms = static_cast<intmax_t>(ceil(nterms_exact * 1.5));  // 50% safety
margin
        // Minimum terms for stability
        if (nterms < 10) nterms = 10;

        float_precision result(stirlingLgamma(x_shifted, working_precision, nterms));
        // Apply recurrence with extra working precision
        // ln(Γ(x)) = ln(Γ(x+shifts)) - ln(x * (x+1) * (x+2) * ... * (x+shifts-1))
        //
        // Instead of summing many log() calls (which accumulates error),
        // compute the product first, then take ONE log
        // De shifting. Compute product: x * (x+1) * (x+2) * ... * (x+shifts-1)
        if (shifts > 0)
        {
                float_precision product(x_work);
                for (intmax_t k = 1; k < shifts; ++k)
                        product *= (x_work + float_precision(k));
                // Now subtract log of the entire product (just ONE log call)
                result -= log(product);
        }

        result.precision(precision);
        return result;
}
```

3) Lanczos–Spouge for ln Γ(x) (minimal changes from Γ(x))

Basically, it is the same method as the Lanczos-Spouge.

$$\Gamma(x+1) = \frac{(x+a)^{x-\frac{1}{2}}}{e^{x+a}} S(x+1) \tag{55}$$

We need to wrap the result into the Ln of x. We keep the same choice of the Spouge parameter a, the same coefficients.

$$c_0 = \sqrt{2\pi} \text{ and } c_k = (-1)^{k-1} \frac{(a-k)^{k-\frac{1}{2}}}{(k-1)!e^{k-a}} \tag{56}$$

And the same backward accumulation of the sum:

$$S(x+1) = (c_0 + \sum_{k=1}^{a-1} \frac{c_k}{x+k}) \tag{57}$$

Taking logs gives the ln-version:

$$\ln\big(\Gamma(x+1)\big) = \left(x+\frac{1}{2}\right)\ln(x+a) - (x+a) + \ln S(x+1)\,, x > 0 \tag{58}$$

For the reflection formula in log form with negative arguments, we convert the Euler reflection to logarithms:

$$ln\Gamma(x) = \ln \pi - \ln |(\sin (\pi x)| - \ln \Gamma(x-1) \quad (59)$$

Integer and half-integer shortcuts (unchanged idea). We compute the additional ln of the result.

Source for the Lanczos-Spouge log gamma.

```
static float_precision logFactorial(const int_precision & n, const intmax_t precision)
	{
	if (n <= int_precision(0))
		return float_precision(0, precision);
	int_precision fact_n = factorial(n);
	float_precision result(0, precision);
	result = log(float_precision(fact_n, precision));
	return result;
	}

// Compute the precision required for calculating n! removing trailing zeros
static size_t factorialprecision(size_t n)
	{
	uintmax_t fivepower;
	size_t prec;
	prec = (size_t)ceil((n * log(n) - n + 0.5 * log(2 * 3.14159265 * n)) / log(10));
	// Subtract traling zeros from n!
	// Using sum([i=1,k]|n/5^i|), where k=floor(ln(n)/ln(5))
	for (fivepower = 5; fivepower < n; fivepower *= 5)
		prec += n / fivepower;
	return prec;
}

static float_precision lgammaHalfinteger(const int_precision & ip, const intmax_t precision)
	{
	const intmax_t working_precision = precision + 10;

	float_precision result(0, working_precision);
	float_precision sqpi(0, working_precision);
	int_precision ip2(ip);
	float_precision ln2(0, working_precision);
	ln2 = log(float_precision(2, working_precision));
	sqpi = _float_table(_PI, working_precision);
	sqpi = log(sqpi) * float_precision(0.5, working_precision);
	ip2 = abs(ip2);
	int_precision ip1 = ip2 * int_precision(2);

	float_precision lnfact_2n(logFactorial(ip1, working_precision));
	float_precision lnfact_n(logFactorial(ip2, working_precision));

	if (ip.sign() > 0)
		result = lnfact_2n + sqpi - lnfact_n - float_precision(ip1,
working_precision) * ln2;
	else
		result = lnfact_n + sqpi + float_precision(ip1, working_precision) * ln2 -
lnfact_2n;

	result.precision(precision);
	return result;
}
// Use Lanczos-Spouge for x > 0
// lnΓ(x) = (x-0.5)·ln(x+a) - (x+a) + ln( sqrt(2π) + Σ ck/(x+k) )
static float_precision lgammaLanczosSpouge(const float_precision& x)
	{
	const intmax_t precision = x.precision() + 8;
	const intmax_t workprec = 150 * precision / 100;
	const float_precision c1(1);
	intmax_t a;
	float_precision fp(0, precision), fpip(0, precision),
		pi(0, precision), ekm1(0, workprec),
		ck(0, workprec), sum(0, workprec);
```

```
        fp = modf(x, &fpip);

    // Integers: lnΓ(n) = ln((n-1)!)
    if (fp.iszero()) {
        if (fpip <= float_precision(0))
            throw float_precision::domain_error();
        return logFactorial(int_precision(fpip) - int_precision(1), precision);
    }

    // Half-integers: use closed form for lnΓ
    if (abs(fp) == float_precision(0.5)) {
        if (x.sign() < 0) fpip -= 1;
        return lgammaHalfinteger(int_precision(fpip), precision);
    }
    // Reflection: lnΓ(x) = lnπ − ln|sin(πx)| − lnΓ(1−x)
    if (x.sign() < 0) {
        const float_precision one(1, precision);
        const float_precision pi_loc = _float_table(_PI, precision);
        const float_precision s = abs(sin(pi_loc * x));
        if (s.iszero()) throw float_precision::domain_error();
        float_precision L = lgammaLanczosSpouge(one - x);
            return log(pi_loc) - log(s) - L;
    }

    // x > 0, Lanczos-Spouge sum stays the same
    a = (intmax_t)ceil(659 * (precision - log(precision) / log(10)) / 526 - 0.5);
    fp = x-c1;  // move x-1 outside the loop

    sum = float_precision(0);
    size_t facprec = factorialprecision(a - 1);
    for (intmax_t k = a - 1; k > 0; --k)
    {
        ck = sqrt(float_precision(a - k, workprec));
        ck *= float_precision(ipow(int_precision(a - k), int_precision(k - 1)),
workprec);
        ekm1 = exp(float_precision(k - a, workprec));
        ekm1 *= float_precision(factorial(int_precision(k - 1)), facprec);
        ck /= ekm1;
        if ((k - 1) & 0x1) ck.sign(-1);
        ck /= (fp + float_precision(k));  // ck/=(x-1+k)
        sum += ck;
    }

    // sqrt(2π) and tail combine unchanged
    pi = _float_table(_PI, precision);
    pi.adjustExponent(+1);      // 2π
    pi = sqrt(pi);              // sqrt(2π)
    sum += pi;

    // FINAL ASSEMBLY IN LOG-SPACE (replace pow/exp block)
    const float_precision half(0.5, precision);
    const float_precision t = fp + float_precision(a);
    // ln Γ(x) = (x+0.5) ln(x+a) − (x+a) + ln(sum)-ln(x)
    float_precision result = (fp + half) * log(t) - t + log(sum);
    result.precision(x.precision());
    return result;
}
```

Performance of ln(|Γ(x)|)

We compare three paths: log(|Γ(x)|), Stirling with argument shifting, and Lanczos-Spouge.

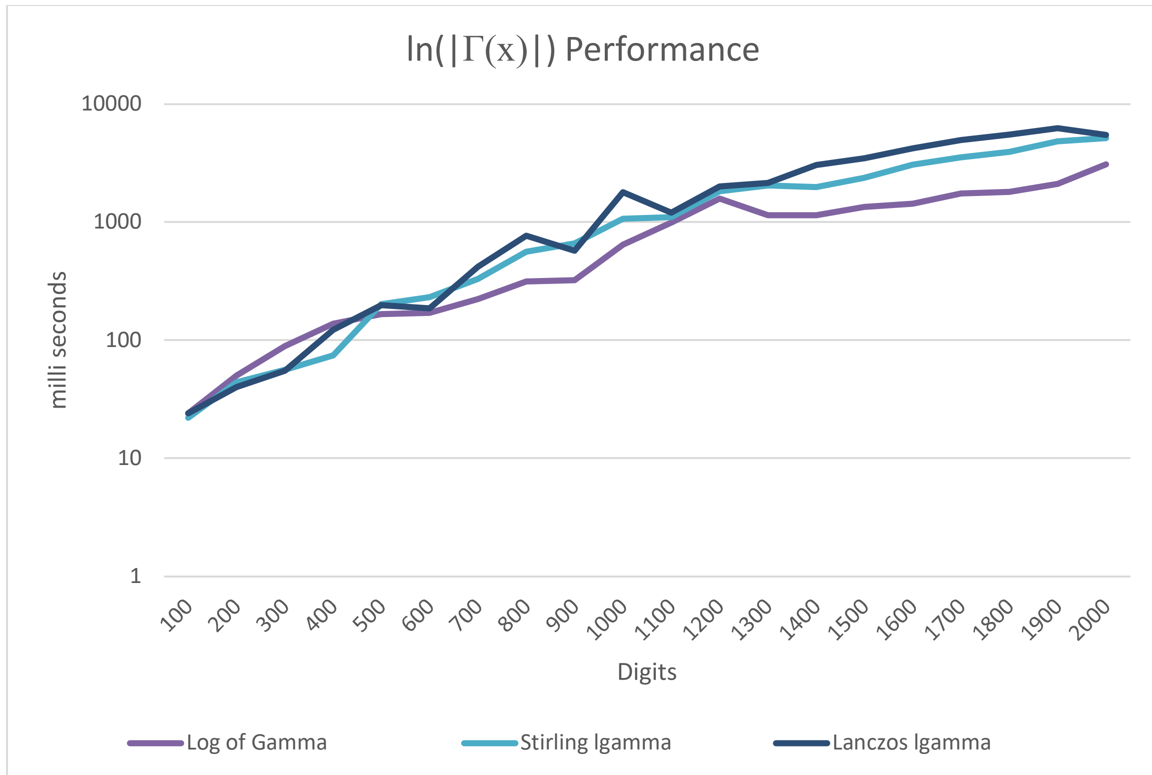


Figure 5 Log Gamma Performance

We see right away that the simple solution, just taking the ln of the gamma function, is the fastest. Striling log gamma is second, followed by the Lanczos-Spouge version.

Recommendation for the ln(|Γ(x)|) function

Since there is no speed advantage to using the specialized versions of the Stirling and Lanczos-Spouge methods, I recommend keeping it simple and taking the logarithm of the gamma function's output.

The Beta function

The Beta function appears in probability theory, binomial models, and many integrals in physics.
The integral defines the beta function:

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}dt \qquad (60)$$

And it is symmetric, meaning that B(z,w)=B(w,z).

One of the nice things about the Beta function is that it's related to the Gamma function:

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \qquad (61)$$

Since we in the previous section have described the Gamma function, we can easily implement the Beta function using the Gamma function.

Source Beta function

```
static float_precision beta(const float_precision& z, const float_precision& w)
{
	return tgamma(z) * tgamma(w) / tgamma(z + w);
}
```

The Error function

The Error function measures the probability that a normally distributed random variable falls within a given range. They are common in statistics, heat diffusion, signal processing, and cumulative Gaussian calculations. They originate from 19th-century work on the normal distribution.
The error function erf(x) is defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt \qquad (62)$$

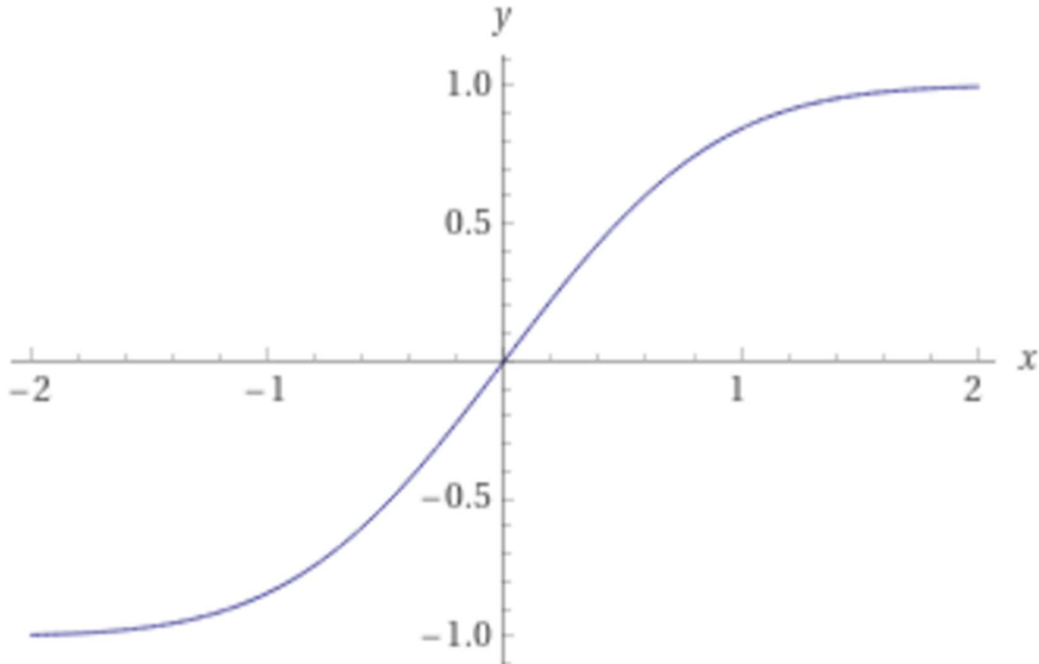


Figure 6. Error function in the interval [-2:+2]

The error function is symmetric, meaning that erf(-x)=-erf(x).

The complementary error function is defined as:

$$\mathrm{erfc(x)} = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2}\, dt = 1 - \mathrm{erf}\,(x) \qquad (63)$$

For the numerical computation of the error function with arbitrary precision arithmetic, there are three formulas suited for this job:

Formula 1: $$\mathrm{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n+1)n!} \qquad (64)$$

Formula 2: $$\mathrm{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}, !!\ is\ the\ double\ factorial \qquad (65)$$

Formula 3: $$\mathrm{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n-1)!!}{(2x^2)^n}, !!\ is\ the\ double\ factorial \qquad (66)$$

Formulas 1 and 3 have the weakness of alternating signs between the terms, while formula 2 requires the calculation of the exponential function e^x as well. Using alternating signs in a summation can lead to cancellation errors and, if not controlled, to reduced accuracy. In [9], they provide a detailed explanation of each method, along with an error bound and a practical implementation guide for each formula. For all three methods, you don't need to know how many terms you would need in the summation. You can continue until the next term is below the requested precision for x and then

terminate the summation. In [3], they recommend only using formula one for |x|≤1 and give the following number of terms needed to achieve an accuracy for P decimal digits:

$$n > 1 + e^{\left(1+W\left(\frac{P\cdot\ln(10)}{e}\right)\right)},\ W\ is\ the\ Lambert's\ W\ function \qquad (67)$$

In [9], they also provide an error bound for each formula; readers are kindly referred to [9] for details. Furthermore, [9] recommends using concurrent series summation rather than the straightforward approach. This is a bit more complicated to implement, but [9] provides a detailed explanation of how to do it properly.
Regarding formula three, which also suffers from alternating sign, I found it easier just to use the identity: erfc(x)=1-erf(x) and rely on just a single solid erf() implementation.
Another deficit of Formula 3 is that the achievable accuracy depends on the magnitude of the number. In [9], they found that the attainable accuracy for P decimal digits was:

$$P \sim e^{2\ln(x)}\frac{\ln(2)}{\ln(10)} \qquad (68)$$

And depends on the magnitude of x.
e.g.

Formula 3 erfc	**Magnitude of x**			
	1	10	100	1’000
Precision Digits	0.3	30	3,010	301,030

Since the achievable precision depends on the magnitude of x and there is nothing else you can do, formula 3 is not very useful for a general computation of the complementary error function.

Performance of the error function

The chart below shows the performance of the straightforward and concurrent-series implementations. They should only be compared pairwise. But in all cases, the method using concurrent series is many times faster than the straightforward approach.

Figure 7. Performance of the Error function

Source erf with the concurrent series method

The source is a modified version of the algorithm presented in [9]

```
static float_precision erf2Concurrent(const float_precision& x)
{
	const size_t precision = x.precision();	// Target precision
	const size_t workprec = precision + 20 + (size_t)ceil(log10(precision));	//
Working precision
	const intmax_t bitprecision = intmax_t(ceil((precision+log10(precision))* log(10) /
log(2)));	// Target bit precision
	intmax_t yexpo, lmax, nmax, i, k, kmin, eps;
	float_precision res(0, workprec);
	float_precision xsq(0, workprec), y(2, workprec), ypl(0,workprec), fpacc(0,
workprec);
	double dsq, a, e1, e1xsq;
	const int extra = 9;

	xsq = x; xsq = xsq.square();	// x^2
	y *= xsq;			// 2x^2
	yexpo = y.exponent();		// y bit exponent
	eps = bitprecision + extra;

	// Compute N using double arithmetic.
	// N/(ex^2)*log2(N/(ex^2)) >= a, where
	a=(targetprecision+3+max(0,x.exponent())-x^2*log2(e))/)ex^2)
	dsq = x; dsq *= dsq; e1 = exp(1); e1xsq = e1 * dsq;
	a = (eps + std::max(intmax_t(0), x.exponent()) - dsq * log2(e1))/e1xsq;
	if (a >= 2.0)
		nmax = intmax_t(ceil(e1xsq * 2 * a / log2(a)));
	else if (a >= 0)  // [0..2]
		nmax = intmax_t(ceil(e1xsq*pow(2.0,0.25)*pow(2.0,a/2)));
	else
	{	// a<0
		nmax = intmax_t(ceil(e1xsq * pow(2.0, a)));
		if (nmax < 2 * dsq)
			nmax = intmax_t(ceil(2 * dsq));
```

```
        }

        eps = -eps;  eps += std::min(intmax_t(0), x.exponent() - 1);
        // Compute L. Optimal with L~sqrt(N)
        lmax = intmax_t(sqrt(nmax));

        ypl = pow(y, float_precision(lmax));  // y^lmax
        vector<float_precision> S(lmax);      // Create a vector of float_precision [lmax]
        for (i = 0; i < lmax; ++i)
                S[i].precision(workprec);     // Set working precision for the Vector
elements

        // Calculate the inital fpacc
        fpacc = _float_table(_PI, workprec);  // pi
        fpacc = sqrt(fpacc);                  // sqrt(pi)
        fpacc = exp(-xsq) / fpacc;            // exp(-x^2)/sqrt(pi)
        fpacc *= abs(x);                      // |x|*exp(-x^2)/sqrt(pi)
        fpacc.adjustExponent(+1);             // 2|x|exp(-x^2)/sqrt(pi)
        if (fpacc.iszero())
                return res = float_precision(x.sign());// Return either +1 or -1

        // Need to do at least kmin loops before we can consider exiting the loop
        kmin = int(std::max(double(y), nmax * 0.9));
        for(k=1, i=0;k<=nmax;++k)     // Loop 1:N
        {
                S[i++] += fpacc;
                if (i == lmax)
                {
                        i = 0;
                        fpacc *= ypl;
                }
                fpacc /= float_precision(2 * k + 1);
                if (k>= kmin && i==0 && (fpacc.exponent() < eps - yexpo * i))
                        break;
        }

        // Res evaluated via Hormer's schema
        res = S[lmax - 1];
        for (i = lmax - 2; i >= 0; --i)
                res = S[i] + y * res;

        res.precision(precision);
        if (x.sign() < 0)             // is negative?
                res.sign(-1);         // erf(-x)=-erf(x)
        return res;
}
```

Source erfc

```
static float_precision erfc(const float_precision& x)
{
        return float_precision(1) - erf(x);
}
```

Recommendation for the Error function

I recommend using formula two implemented with the concurrent series method. The benefit is that it is stable because it avoids alternating signs between the terms, and even though it requires computing the exponential function, it is still significantly faster than any of the other methods and works well for both large and small magnitudes of x.

Lambert W function

The Lambert W function solves many problems in which the unknown appears in both the base and the exponent, such as in combinatorics, delay differential equations, and certain optimization formulas. It entered widespread use in the late 20th century due to practical applications in computing and physics.

The Lambert W function is the solution to $we^w=z$ where z is any complex number. Since we are only dealing with the real value x, we can solve $we^w=x$ for any $x \geq -\frac{1}{e}$ and we get $w=W_0(x)$ if $x\geq 0$ and two values $w=W_0(x)$ and $W_{-1}(x)$ if $-\frac{1}{e} \leq x < 0$.
W_0 is called the primary branch, and that is the one we want to compute.

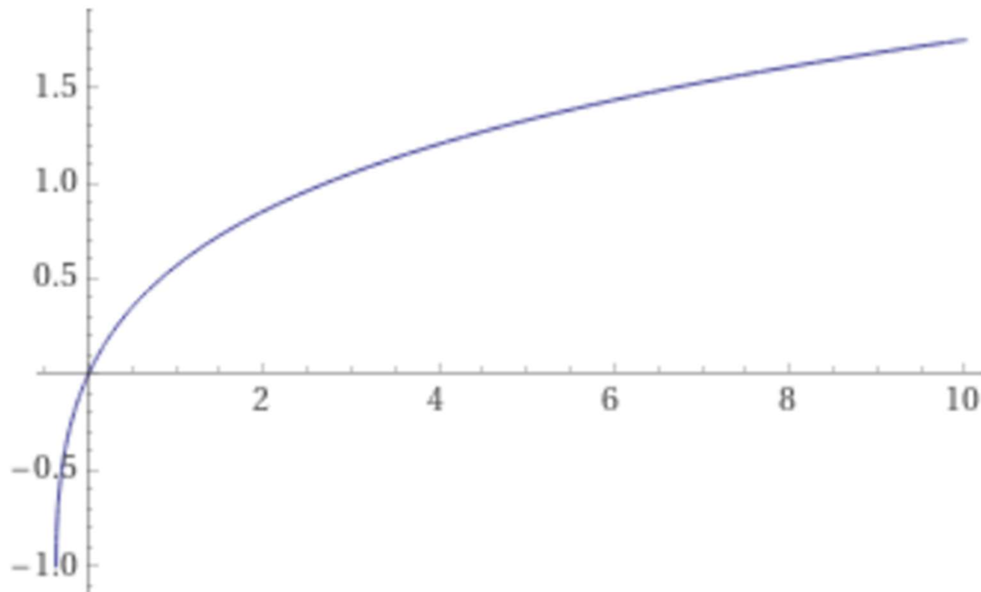


Figure 8 Lambert W function w0(x) in the interval]-1/e:10]

There are three methods suitable for arbitrary-precision arithmetic. These are:

- Newton's iterative method (quadratic convergence).
- Halley's iterative method (cubic convergence).
- Boyd-Iacono iterative method (quadratic convergence).

As always for iterative methods, we need to find a suitable starting point for our iterations. Since we use the same starting point for all three iterative methods, we will describe it first, then address each method in turn.

A Suitable starting point for Lambert W Iteration.

We do not usually have a Lambert W function available, so we cannot easily achieve 15-16 digit accuracy using the built-in double type in C or C++. In the literature, they usually suggest a starting point for the Lambert W function as follows.

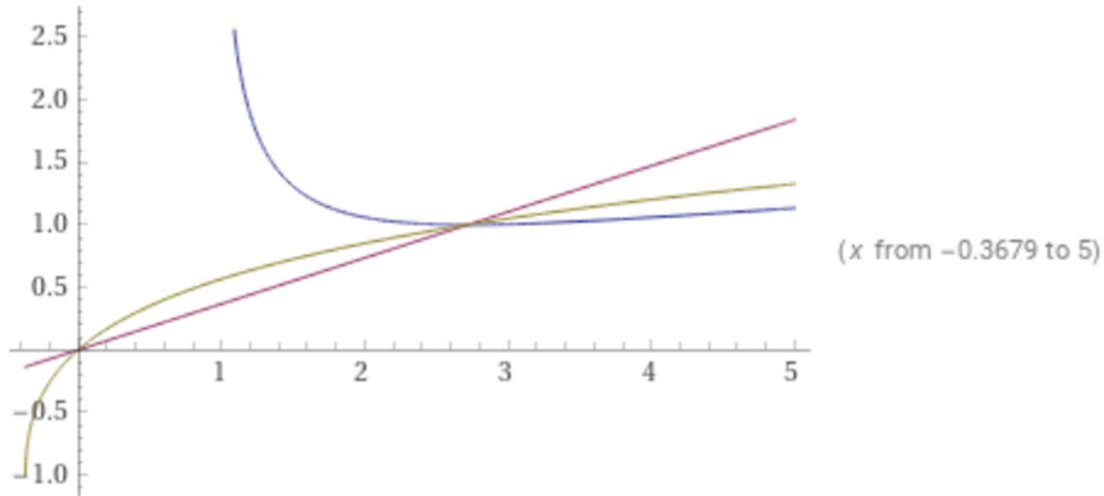


Figure 9: Amber color is $W_0(x)$

If x in [e,∞]:	$w_0(x) = \ln(x) - \ln(\ln(x))$	"blue line in the above figure
If x in [0,e[:	$w_0(x) = \frac{x}{e}$	"magenta line in the above figure"
If x in [-1/e,0[:	$w_0(x) = \frac{e\cdot x}{1+e\cdot x+\sqrt{1+e\cdot x}}\ln(1+\sqrt{1+e\cdot x}$	

Not an impressive, precise start point; however, it usually gives a relative error of less than 10^{-1} as the start point.

Newton's quadratic method

A classic Newton method can be used, and you will iterate through the following iterations:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n} - x}{e^{w_n} + w_n e^{w_n}} \tag{69}$$

Newton's method has a quadratic convergence rate, meaning that the number of correct digits doubles with every iteration. Unfortunately, it is required to evaluate e^{w_n} For every iteration. This is certainly not ideal since that will be the dominant time-consuming part of our computation.

Halley's cubic method

Alternatively, a cubic convergence rate is Halley's iteration:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n} - x}{e^{w_n(w_n+1) + \frac{(w_n+2)(w_n \cdot e^{w_n} - x)}{2(w_n+2)}}} \tag{70}$$

Again, we see that we need to calculate e^{w_n} And two divisions for every iteration.

Boyd's quadratic method

Boyd & Iacono iteration has quadratic convergence, which is the same as the Newton method.

$$w_{n+1} = \frac{w_n}{1+w_n}(1 + \ln(\frac{x}{w_n})) \tag{71}$$

It looks simpler than the Newton method; however, you also need to compute the time-consuming function ln(x) for every iteration and perform two divisions.

Initial performance of the Lambert W function

Performance-wise, Halley is faster up to around 6,000 digits of precision, but the Boyd method takes over thereafter, even though Halley uses fewer iterations. The reason is that our implementation of ln(x) in arbitrary precision is faster than the exp(x) function, so Boyd iteration will win despite having only a quadratic convergence rate.

However, there is a technique for speeding up this classic iterative method, described in [10], that avoids iterating with full precision for each iterative variable. We dynamically adjust the precision as we iterate to accommodate the target precision at each step. E.g., if we need 1,000-digit precision for Lambert W(x), we do not need more than 20 digits for the first 5 iterations (since our initial guess was around 10^{-1}). Then we can gradually increase them to our target precision of 1,000 digits over the subsequent six iterations. It would not be very wise to carry these first five and subsequent iterations with a precision of 1,000 digits. Needless to say, this dramatically speeds up the computation.

Source Newton method with dynamic precision

```
static float_precision LambertNewtonDyn(const float_precision& x)
{
	const size_t precision = x.precision();
	const intmax_t exponent = x.exponent();
	const size_t workprec = precision + 2 + (size_t)ceil(log10(precision));
	const double e1 = exp(1);
	const intmax_t limit = -intmax_t(ceil(precision * log(10) / log(2)));
	intmax_t eps, np;
	float_precision wn, wtmp, f, f1, dx; // w default precision (20digits)
	int icnt;
	double fd = x;

	// Only works for x>= -1/e
	// find a suitable start point
	wn = x;
	if (x >= float_precision(e1))  // xin [e,infinity[
	{	// start point = log(wn)-log(log(wn))
		// remove exponent to allow the start point to be calculated using double
		wn.exponent(0);
		fd = double(wn);		// No overflow since wn is now in range [1..2[
		fd = log(fd) + exponent * log(2);	// log(wn)
		fd -= log(fd);			// log(wn)-log(log(wn))
		wn = float_precision(fd);
	}
	else
		if (x >= float_precision(0))  // x in [0,e[
			wn /= float_precision(e1);
		else
		{	// x in ]-1/e,0[	ex*ln(1+sqrt(1+ex))/(1+ex+sqrt(1+ex))
```

```
				double ex = wn;
				ex *= e1;
				double exp1sq = sqrt(1 + ex);
				ex *= log(1 + exp1sq) / (1 + ex + exp1sq);
				wn = float_precision(ex);
			}

	// Newton quadratic iteration
	for (icnt = 0;; ++icnt)
	{
		wtmp = exp(wn);		// only evaulate exp(wn) once per iteration
		f = wn * wtmp;		// wn*wexp
		f1 = wtmp + f;		// f1(x)=wexp+wn*wexp=(1+wexp)wn
		f -= x;			// f(x)=wn*wexp-x
		dx = f / f1;
		eps = abs(dx).exponent();
		if (dx.iszero() || eps <= limit)
			break;		// No more improvements.
		// Adjust precision so it fits into the next iteration target precision
		// Overallocate 1.5 to make Newton more efficient
		np = -intmax_t(ceil( 1.5 * eps ));
		np = std::max(np, intmax_t(PRECISION));
		np = std::min(np, intmax_t(workprec));
		if (wn.precision() < size_t(np))
		{
			wtmp.precision(np); wn.precision(np);
			f.precision(np); f1.precision(np); dx.precision(np);
		}
		wn -= dx;
	}

	wn.precision(precision);
	return wn;
}
```

Source Halley methods with dynamic precision

```
static float_precision LambertHalleyDyn(const float_precision& x)
{
	const size_t precision = x.precision();
	const intmax_t exponent = x.exponent();
	const size_t workprec = precision + 2 + (size_t)ceil(log10(precision));
	const double e1 = exp(1);
	const float_precision c1(1), c2(2);
	const intmax_t limit = -intmax_t(ceil(precision * log(10) / log(2)));
	intmax_t eps, np;
	float_precision wn, wtmp, f, f1, dx;
	int icnt;
	double fd;

	// only works for x>= -1/e
	// find a suitable start point
	wn = x;
	if (x >= float_precision(e1))  // xin [e,infinity[
	{	// start point = log(wn)-log(log(wn))
		// remove exponent to allow the start point to be calculated using double
		wn.exponent(0);
		fd = double(wn);	// No overflow since wn is now in range [1..2[
		fd = log(fd) + exponent * log(2);	// log(wn)
		fd -= log(fd);				// log(wn)-log(log(wn))
		wn = float_precision(fd);
	}
	else
		if (x >= float_precision(0))  // x in [0,e[
			wn /= float_precision(e1);
		else
		{	// x in ]-1/e,0[	ex*ln(1+sqrt(1+ex))/(1+ex+sqrt(1+ex))
			double ex = wn;
```

```
                        ex *= e1;
                        double exp1sq = sqrt(1 + ex);
                        ex *= log(1 + exp1sq) / (1 + ex + exp1sq);
                        wn = float_precision(ex);
                }

        // Halley cubic iteration
        for (icnt = 0;; ++icnt)
        {
                wtmp = exp(wn);                 // only evaulate exp(wn) once per iteration
                f = wn * wtmp;                  // f(x)=wn*e^wn
                f -= x;                         // f(x) = wn * e ^ wn - x
                f1 = wn + c2;                   // wn+2
                f1 *= f;                        // (wn+2)(wn*e^(wn)-x)
                dx = wn + c1;                   // wn+1
                f1 /= dx;                       // ((wn+2)(wn*e^(wn)-x)/(wn+1)
                f1.adjustExponent(-1);          // ((wn+2)(wn*e^(wn)-x)/(2(wn+1))
                wtmp *= dx;                     // e^wm(wn+1)
                f1 = wtmp - f1;                 // e^wm(wm+1)-((wn+2)(wn*e^(wn)-x)/(2(wn+1))
                dx = f / f1;                    // dx=f(x)/f'(x)
                eps = abs(dx).exponent();
                if (dx.iszero() || eps <= limit)
                        break;          // No more improvements.
                //wn -= dx;
                // Adjust precision so it fits into the next iteration target precision
                np = -intmax_t(ceil(2.75*eps ));
                np = std::max(np, intmax_t(PRECISION));
                np = std::min(np, intmax_t(workprec));
                if (wn.precision() < size_t(np))
                {
                        wtmp.precision(np); wn.precision(np);
                        f.precision(np); f1.precision(np); dx.precision(np);
                }
                wn -= dx;
        }

        wn.precision(precision);
        return wn;
}
```

Source Boyd method with dynamic precision

```
static float_precision LambertBoydDyn(const float_precision& x)
{
        const size_t precision = x.precision();
        const intmax_t exponent = x.exponent();
        const size_t workprec = precision + 2 + (size_t)ceil(log10(precision));
        const double e1 = exp(1);
        const float_precision c1(1);
        const intmax_t limit = -intmax_t(ceil(precision * log(10) / log(2)));
        intmax_t eps, np;
        float_precision wn, wtmp, wpre;
        int icnt;
        double fd;

        // Only works for x>= -1/e
        // find a suitable start point
        wn = x;
        if (x >= float_precision(e1))  // xin [e,infinity[
        {       // start point = log(wn)-log(log(wn))
                // remove exponent to allow the start point to be calculated using double
                wn.exponent(0);
                fd = double(wn);        // No overflow since wn is now in range [1..2[
                fd = log(fd) + exponent * log(2);      // log(wn)
                fd -= log(fd);                         // log(wn)-log(log(wn))
                wn = float_precision(fd);
        }
        else
                if (x >= float_precision(0))  // x in [0,e[
```

```
                    wn /= float_precision(e1);
            else
            {       // x in ]-1/e,0[  ex*ln(1+sqrt(1+ex))/(1+ex+sqrt(1+ex))
                    double ex = wn;
                    ex *= e1;
                    double exp1sq = sqrt(1 + ex);
                    ex *= log(1 + exp1sq) / (1 + ex + exp1sq);
                    wn = float_precision(ex);
            }

    // Boyd quadratic iteration
    for (icnt = 0;; ++icnt)
    {
            wpre = wn;
            wtmp = x / wn;          // x/wn
            wtmp = log(wtmp);       // only evaluate log() once per iteration
            wtmp += c1;             // 1+log(x/wn)
            wtmp *= wn;             // wn*(1+log(x/wm))
            wn += c1;               // wn=wn+1
            wn = wtmp / wn;         // wn*(1+log(x/wm))/(wn+1)
            eps = abs(wn - wpre).exponent();
            if (wn == wpre || eps <= limit)
                    break;          // No more improvements.
            // Adjust precision so it fits into the next iteration target precision
            np = -intmax_t(ceil(1.5*eps));
            np = std::max(np, intmax_t(PRECISION));
            np = std::min(np, intmax_t(workprec));
            if (wn.precision() < size_t(np))
            {       // adjust precisions
                    wtmp.precision(np); wn.precision(np); wpre.precision(np);
            }
    }

    wn.precision(precision);
    return wn;
}
```

Performance of Lambert W function

All "dynamic" methods are somewhat similar, but the Boyd methods begin to take off after approximately. 6,000 digits and is thereafter the fastest method. See the performance chart below.

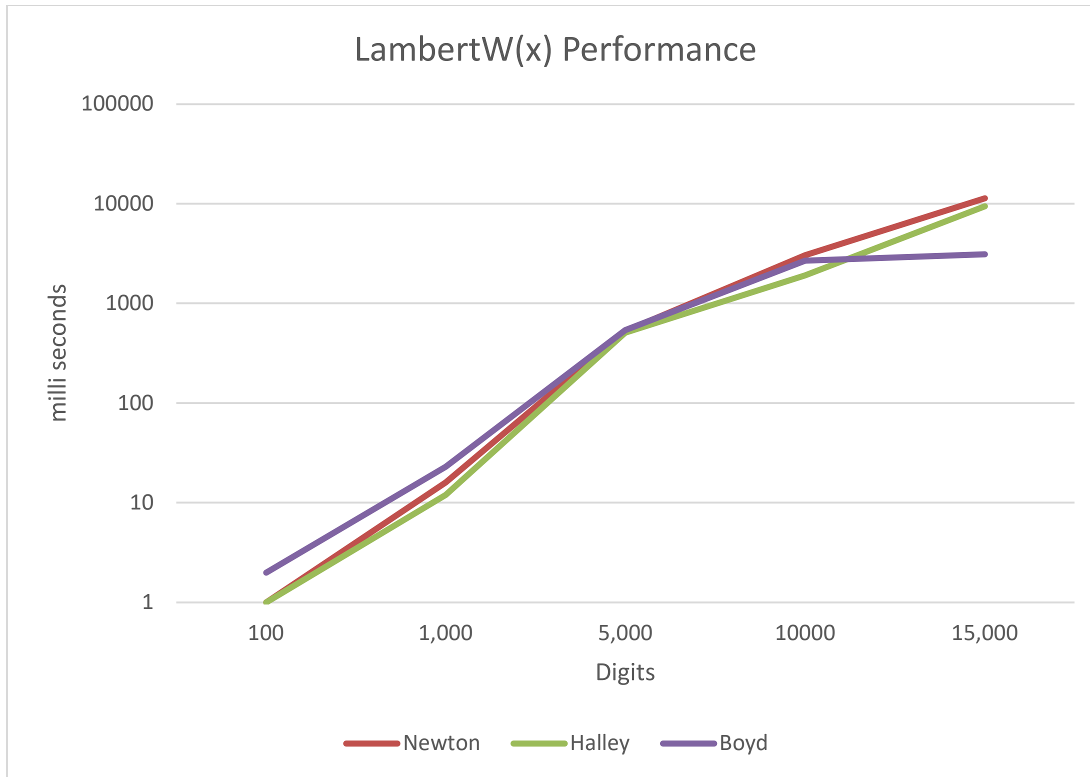


Figure 10. Performance of Lambert W(x) function

Recommendation for Lambert W function

- The preferred method is Boyd's method.
- Although Halley is close, if your arbitrary-precision library has a faster exp(x) function than log(x), then the Halley method is preferred.

Riemann Zeta function

The Riemann Zeta function ζ(s) is a central object in number theory. It is defined as an infinite sum for Re(s) > 1 and is extended to other values through analytic continuation. It appears in series acceleration, physics (quantum and statistical mechanics), and special constant computations. The study of its zeros is at the heart of the famous Riemann Hypothesis.

The Riemann zeta function is defined for any complex value z as:

$$\zeta(z) = \sum_{n=0}^{\infty} \frac{1}{n^z} \tag{72}$$

The graph for any real value $\zeta(x)$ is below with a pole for s=1

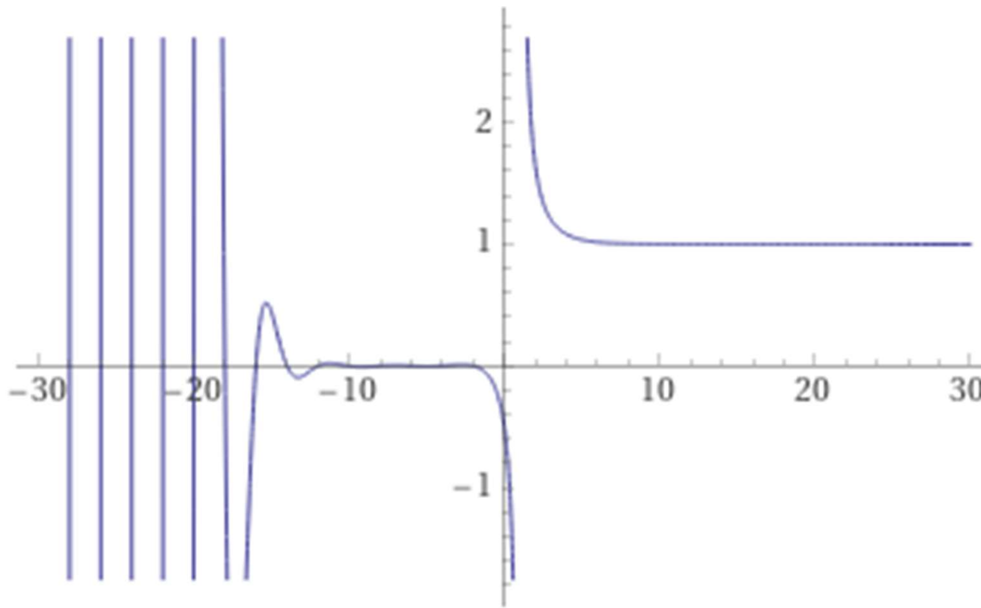


Figure 11 Zeta(x) in the interval [-30:+30]

There are several interesting identities. One of them is this formula, which is helpful for mapping s that are negative over to the positive real axis.

$$\zeta(1-s)=\frac{2\Gamma(s)}{(2\pi)^s}\cos\left(\frac{\pi\cdot s}{2}\right)\zeta(s) \tag{73}$$

There are many others and quite a few for special values of s when s is an even or odd integer. We are looking into a more general method to calculate $\zeta(s)$ For any real values. Peter Borwein published several methods in 1995 [11], and in particular, his algorithm 3 is of interest due to its simplicity. The formula ([11]) below is valid for any real s>-(n-1).

$$\zeta(s)=\frac{-1}{2^n(1-2^{1-s})}\sum_{j=0}^{2n-1}\frac{e_j}{(j+1)^s} \tag{74}$$

Where e_j is defined as:

$$e_j=(-1)^j\left(\left(\sum_{k=0}^{j-n}\frac{n!}{k!(n-k)!}\right)-2^n\right) \tag{75}$$

The $\sum$ is zero for j<n. The parameter *n* must be chosen to ensure the desired precision is achieved. The formula above has an error estimation $O(8^{-n})$. To achieve *P* decimal digits precision, you need:

$$n=\left\lceil\frac{\ln(10)}{\ln(8)}P\right\rceil \tag{76}$$

Unfortunately, if we look at the formula for $\zeta(s)$ We notice that we have two power function calls, $(j+1)^s$ and 2^{1-s}. The latter must be repeated 2n times. The power function x^y

requires a call to both log() and the exp() function, if s is not an integer, and is therefore a costly function to call, so we can't expect too high performance when computing the zeta function.
However, we get a little bit of a break when s is large, where the use of the actual definition for the zeta function performs faster than the Borwein formula. If the condition

$$s > 1 + P\frac{\ln(10)}{\ln(P)} \tag{77}$$

Where *P* is the target precision (in decimal digits) to meet, we can resort to the following series for faster computation.

$$\zeta(s) \approx \sum_{k=0}^{N} \frac{1}{k^s} \tag{78}$$

And a suitable value for N is:

$$N = \left\lceil 10^{\frac{P}{s-1}} \right\rceil \tag{79}$$

Now, there are some handy shortcuts we can make that are easy to compute. These are if *s* is equal to zero, is a negative integer, or is a positive even integer.

Short-cut identities for ζ. B_n is n'th Bernoulli number and n is an integer:

$$\zeta(0) = -\frac{1}{2} \tag{80}$$

$$\zeta(-n) = (-1)^n \frac{B_{n+1}}{n+1} \tag{81}$$

$$\zeta(2n) = (-1)^n \frac{B_{2n}(2\pi)^{2n}}{2(2n)!} \tag{82}$$

For odd positive integers, there is unfortunately no easy formula; however, there is a myriad of series you can find in various published papers that promise to be faster than the above general formula for zeta.

Optimization of the ζ(s) series. If we look at the computation for e_j in equation (64).

$$e_j = (-1)^j \left(\left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!}\right) - 2^n\right) \tag{83}$$

We notice that $\frac{n!}{k!(n-k)!}$ Are the binomial coefficients or $\binom{n}{k}$ and it is usually faster to call our optimized binomial (n,k) function than just calculating the three factorials. e_j becomes:

$$e_j = (-1)^j \left(\left(\sum_{k=0}^{j-n} \binom{n}{k}\right) - 2^n\right) \tag{84}$$

However, the real optimization trick is when we realized that we can replace the $\sum$ with the following recurrence:

$$if\ j < n\!: sum_j = 0$$
$$if\ j \geq n\!: sum_j = sum_{j-1} + \binom{n}{j-n}$$

And e_j now becomes:

$$e_j = (-1)^j(sum_j - 2^n) \tag{85}$$

This trick speeds up the computation by more than 1,600-2,000 times when calculating ζ(3) with 500-digit precision.

We can now present our final algorithm for the ζ(s) function.

Algorithm 3 for ζ(s) where s is any real value not equal to 1

```
if s=0 return -0.5
if s an integer?
    if s negative?
            if s even =>  return 0
            return compute (81)
    if s even?
            return compute (82)
// s is real or s is an odd integer goes here
if s < 0.5?
    return compute (73)
if s > 1+P·log(10)/log(P)   // if s large?, P is the decimal precision required
    return compute (78)
// s is > 0.5 but not large
    return compute (74)
```

Source for zeta function computation.

The code below is a hybrid implementation that uses all the above formulas optimally.

```
float_precision zeta(const float_precision& x)
{
	// Handle NaN
	if (isnan(x))
		return x;

	const size_t precision = x.precision();       // Target precision
	const intmax_t n = intmax_t(ceil(precision * log(10) / log(8)));
	const size_t workprec = precision + 10 + (size_t)ceil(log10(precision));     //
Working precision
	const float_precision c0(0), c1(1, workprec), c2(2, workprec);
	intmax_t j;
	int_precision ip;
	float_precision s(x), res(0, workprec), jsum(0, workprec), tmp(0, workprec);
```

```
	if (x == c1)
		throw float_precision::domain_error();

	// Handle simple stuff. e.g. zero, negative integers, and positive even integers
	if (x.iszero())
		return float_precision(-0.5, precision);  // zeta(0)=-0.5

	tmp = modf(x, &res);  // s is the fraction part, res the integer part or 0
	if (tmp.iszero())  // Fraction part is zero, then we are dealing with true integers
	{	// Fraction part is zero
		fraction_precision<int_precision> B;
		ip = int_precision(res);  // Get the integer as an int_precision integer

		if (ip.sign() < 0)
		{
			if (ip.even())
				return float_precision(0, precision);  // zeta(-2k)=0
			int_precision n = -ip;			// n = -x  (positive odd here)
			B = bernoulli((size_t)(n + 1));		// B_{n+1}
			// B_{n+1}/(n+1)
			B *= fraction_precision<int_precision>(int_precision(1), 
int_precision(1)+n);
			res = float_precision(B.numerator(), workprec);
			res /= float_precision(B.denominator(), workprec);
			if (n.odd())				// (-1)^n
				res.change_sign();
			res.precision(precision);
			return res;
		}

		// Positive n.  Now check for even integers otherwise, we have to do it the 
regular way
		if (ip.even())
		{	// Even integers goes here
			B = bernoulli((size_t)ip);  // Get Bernoulli(n)
			B *= fraction_precision<int_precision>(int_precision(1), 
int_precision(2) * factorial(ip));// Bernoulli(n)/(2*n!)
			// Convert fraction_precision to float_precision
			res = float_precision(B.numerator(), workprec);
			res /= float_precision(B.denominator(), workprec);
			// Multiply with (2pi)^n
			tmp = _float_table(_PI, workprec);	// pi
			tmp.adjustExponent(+1);			// 2pi
			tmp = pow(tmp, float_precision(ip, workprec));
			res *= tmp;
			int_precision n = ip >> 1;		// since ip is even and positive 
here
			if (n.even())  // (-1)^(n+1)
				res.change_sign();
			res.precision(precision);
			return res;
		}
		// Positive odd number falls through and goes to normal processing
	}

	// Continue with regular zeta algorithm
	s.precision(workprec);
	if (s < float_precision(0.5))
	{ // use the identity of zeta(1-s)=zeta(s)*....
		res = zeta(c1 - x);
		tmp = _float_table(_PI, workprec);	// pi
		tmp.adjustExponent(+1);			// 2pi
		tmp = pow(tmp, s);			// (2PI)^s
		res *= tmp;				// zeta(1-s)*(2pi)^s
		tmp = tgamma(x);			// T(x)
		tmp.adjustExponent(+1);			// 2T(x)
		tmp = tmp.inverse();			// 1/(2T(x))
		res *= tmp;				// zeta(1 - s)*(2pi)^s/(2T(x))
		tmp = _float_table(_PI, workprec);	// tmp=pi from cached value
```

```
                tmp.adjustExponent(-1);                 // pi/2
                tmp *= s;                               // s*pi/2
                tmp = cos(tmp);                         // cos(s*pi/2)
                tmp = tmp.inverse();                    // 1/cos(s*pi/2)
                res *= tmp;                             // zeta(1 -
s)*(2pi)^s/((2T(x))*cos(s*pi/2))
                res.precision(precision);
                return res;
        }

        // Using zeta definition instead if certain conditions is fulfilled
        double sd = double(x);
        if (sd > 1 + log(10) / log(precision) * precision)
                return _zetadef(x);  // This is faster for larger s

        int_precision itwopn = ipow(int_precision(2), int_precision(n));  // 2^n
        int_precision ipsum(0);
        res = c0;  // ensure res=0
        for (j = 0; j <= 2 * n - 1; ++j)
        {
                // Calculate ej, unsing int_precision to avoid any loss of the binomial sum
                ip = int_precision(0);
                if (j >= n)
                {
                        ipsum += binomial(int_precision(n), int_precision(j - n));
                        ip = ipsum;
                }
                ip -= itwopn;                           // sum(binomial(n,k)-2^n
                tmp = float_precision(ip, workprec);   // sum(binomial(n,k)-2^n
                if (j & 0x1)  // Odd
                        tmp.change_sign();              // ej=(-1)^j*sum(binomial(n,k)-2^n

                tmp /= pow(float_precision(j + 1, workprec), s); // ej/((j+1)^s)
                if (tmp.sign() < 0)
                        res += tmp;     // Sum up the negative ej in res
                else
                        jsum += tmp;// Sum up the positive ej in jmsum
        }
        // Finalise the zeta(s) value
        tmp = c1 - pow(c2, float_precision(c1 - s));  // 1-2^(1-s)
        tmp *= float_precision(itwopn, workprec);     // 2^n*(1-2^(1-s))
        res += jsum;            // sum([j=0,2n-1]ej/(j+1)^s
        res.change_sign();      // -1*sum([j=0,2n-1]ej/(j+1)^s
        res /= tmp;             // (-1*sum([j=0,2n-1]ej/(j+1)^s)/(2^n*(1-2^(1-s)))
        res.precision(precision);
        return res;
}
```

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2) Numerical Recipes in C++, 3rd edition, Cambridge University Press, New York, NY 2007
3) The Yacas book of algorithms, Version 1.3.3, April 1, 2013, by the Yacas team
4) K.J. McGowan Computing Bernoulli Number quickly, December 8, 2005
5) J.L. Spouge, Computation of the Gamma, Diagamma, and Trigamma functions. SIAM Journal on Numerical Analysis. 31(3): 931-000
6) Frederik Johansson, Arbitrary-precision computation of the gamma function. 2021. Hal-03346642. HAL open science.
7) Alexander Yee, Binary Splitting Recursion Library
8) G.Free, Computation of Catalan's Constant using Ramanujan's formula. 1990 ACM
9) S. Chevillard, The functions erf, and erfc computed with arbitrary precision and explicit error bounds. Information and Computation, volume 216, July 2012, pages 72-95
10) Henrik Vestermark, "HVE The math behind arbitrary". HVE The Math behind arbitrary precision.docx (hvks.com)
11) P. Borwein, "An efficient Algorithm for the Riemann Zeta function", Canadian Mathematical Society, Conference paper.

Appendix

Cross-reference clarity

Function	Recommended Method	Precision Range	Performance	Comments
Factorial	Loop-based for small values and Binary Splitting for larger values	All	Fast	
Bernoulli	The Fast computation	All	Moderate	
$\Gamma(x)$	Integration by Parts	All	Fastest	Needs shifting
$\ln \Gamma(x)$	Ln $\Gamma(x)$	All	Stable	
$B(x,y)$	$\Gamma(x)\Gamma(y)/\Gamma(x+y)$	All	Depends on Γ	
erf(x)	Series/Continued fraction	Moderate	Very good	Works well near 0–3
$\zeta(s)$	Euler–Maclaurin	Moderate	Slow for large s	—
$W_0(x)$	Boyd iteration	All	Excellent	Needs initial guess