

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

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

By Henrik Vestermark (hve@hvks.com)

Abstract:

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

Introduction:

We start up with simple functions like the factorial, falling, and rising factorials, and binomial coefficients continuing with Bernoulli numbers and various optimization technic to improve performance.

We then turn our attention to the Gamma, Beta, Error, Zeta, and Lambert W_0 functions.

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

This paper is part of a series of arbitrary precision papers describing 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 & 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](#)

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

10. Fast conversion from 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](#)

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

Contents

Abstract:.....	1
Introduction:.....	1
The Arbitrary precision library.....	5
Internal format for float_precision variables.....	6
Normalized numbers.....	6
Factorials.....	7
Source factorial via recursion.....	7
Source factorial looping.....	7
Source factorial.....	8
Falling Factorials.....	9
Source Partial factorial.....	9
Source factorial via threading.....	10
Binomial coefficients.....	10
Source for binomial.....	10
Bernoulli numbers.....	11
Bernoulli using recursion.....	11
Source Bernoulli using recursion.....	12
Bernoulli numbers using explicit formula.....	13
Source Bernoulli using explicit formula.....	13
Fast Calculation of Bernoulli numbers.....	14
Example of Bernoulli number computation.....	14
Source of Fast Bernoulli calculation.....	16
Bernoulli performance.....	18
Recommendation of Bernoulli method.....	18
Bernoulli Polynomials.....	18
Source for Bernoulli polynomials.....	18
Gamma function.....	19
Algorithm for Gamma computation.....	20
Source for the Half Integer calculation.....	20
Lanczos-Spouge method.....	21
Source of Lanczos-Spouge method.....	22
Stirling asymptotic series method.....	23
Source of Stirling asymptotic method.....	25
Integration by parts method.....	26
Source Integration by parts.....	27
Gamma Performance.....	29
Recommendation for the Gamma function.....	29
The Beta function.....	30
Source Beta function.....	30
The Error function.....	30
Performance of the error function.....	32
Source erf with the concurrent series method.....	33
Source erfc.....	34
Recommendation for the Error function.....	34
Lambert W function.....	35

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

A Suitable starting point for Lambert W Iteration.....	35
Newton's quadratic method	36
Halley's cubic method	36
Boyd quadratic method.....	37
Initial performance of Lambert W function.....	37
Source Newton method with dynamic precision	37
Source Halley methods with dynamic precision.....	38
Source Boyd method with dynamic precision	39
Performance of Lambert W function	40
Recommendation for Lambert W function.....	41
Riemann Zeta function.....	41
Source for zeta function computation.....	44
Reference	47

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

The Arbitrary precision library

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

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

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

You can add a few parameters to the declaration. The first is the optional initial value and the second optional parameter is the 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 to manipulate 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 to manipulate the exponent of the variables. The method is called `.exponent()` and returns or sets the exponent as a power of two exponent (same as for our regular build-in types *float* and *double*) E.g.

```
f.exponent(); // Return the exponent as 2e  
f.exponent(0) // Remove the exponent  
f.exponent(16) // Set the exponent to 216
```

There is a second way to manipulate the exponent and that is the class method `.adjustExponent()`. This method just 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 with 2.  
f.adjustExponent(-1); // Subtract 1 from the exponent, the same as dividing the number with 2.
```

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

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

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

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

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

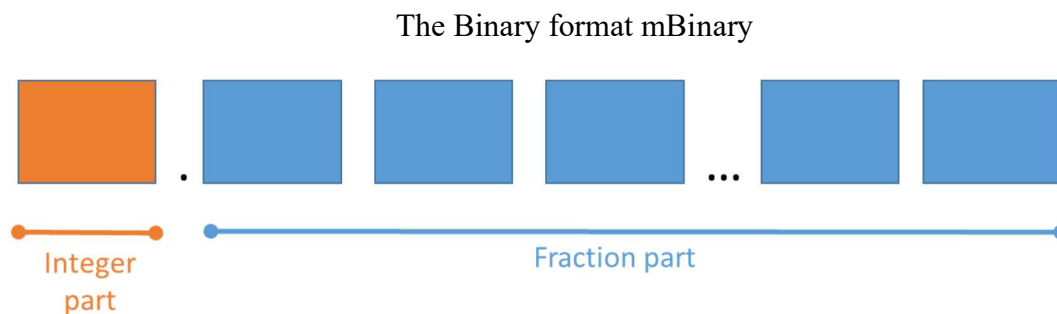
Internal format for float_precision variables

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 like the sign, exponent, precision, and rounding mode but these are not important to understand the code segments.

Normalized numbers

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

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 one of the simple formulas to implement. Its definition is:

$$n! = n \cdot (n - 1) \cdot (n - 2) \dots 2 \cdot 1 \quad (1)$$

Or using a recursive definition:

$$n! = n \cdot (n - 1)! \quad (2)$$

Where $0! = 1$

Most often you see it implemented as below straight forward code using the recursion.

Source factorial via recursion

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

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

As we were told back in the computer science class recursion is good but unrolling it to a loop-based algorithm is better. Above code rewritten into 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*.

Fast Gamma, Beta, Error, Zeta function & Bernoulli numbers for Arbitrary 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 go to “unlimited” factorials, however, the above algorithm is not particularly fast since arbitrary precision slows down the computation. One improvement could be to do some of the intermediate computations using 64-bit arithmetic. E.g. we could group two consecutive numbers together and then add this number to an arbitrary precision result and to limit any overflow we could rearrange the factorial and start by multiplying the top n with 1 thereafter (n-1)2 etc.

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

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

```
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 gives a nice speedup since the pairwise multiplication of k and l is done using 64-bit arithmetic. Continue along with that idea you can generalize it and take the pairwise multiplication many times depending on the range of numbers you are multiplying.

Source factorial

```
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);
}
```


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

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

The performance from the simple loop-based to the more advanced version above represents a speedup of a factor of 6-7 over the straightforward loop-based version. With arbitrary precision arithmetic, you always want to take speedup over simplicity.

Falling Factorials

Falling factorial describe factorial between two number m and n in falling (or descending notation):

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

Falling factorials are useful when calculating the binomial coefficient:

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

Since the falling factorial is:

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

And can be handled by one call to the falling factorial function instead of two calls to the factorial function.

See the next chapter for the binomial coefficients.

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

Source Partial factorial

```
int_precision fallingfactorial(const int_precision& mip, const int_precision& nip)
{
    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;
        }
    }
}
```

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

```
    }
  }
  res *= int_precision(kk);
  if (n == m)
    res *= int_precision(n);
  return res;
}
```

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

$$\frac{n!}{(n-m)!} \text{ and } (n-m)! \quad (7)$$

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

Source factorial via threading

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

    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 first worthwhile when dealing with factorial exceeding 2000!

Binomial coefficients

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \text{ where } 0 \leq m \leq n \quad (8)$$

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;
}
```

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

```
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:

$$\binom{n}{m} = \binom{n}{n-m} \quad (9)$$

Sometimes there is a need for the calculation of a steady stream of an increasing number of 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:

$$\binom{n+1}{m} = \frac{n+1}{n-m+1} \binom{n}{m} = \binom{n}{m} + \binom{n}{m-1} \quad (10)$$

Or:

$$\binom{n}{m+1} = \frac{n-m}{m+1} \binom{n}{m} \quad (11)$$

Bernoulli numbers

The Bernoulli numbers B_n are a sequence of rational numbers. E.g. one of the methods 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 instead use fraction arithmetic to calculate the Bernoulli numbers. This is exact and we can calculate them once and then convert them as needed back to any arbitrary precision floating point number in any precisions. 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 number 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 with e^t-1 you get:

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

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

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 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) // Uninitialized 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];
}
```

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

Bernoulli numbers using explicit formula

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

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

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 below source 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) // Uninitialized 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) * bin /
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));
                }
            }
        }
    }
}
```

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

```

        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, calculating 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. $a = (-1)^{\frac{n}{2}+1} \text{round}(dKz)$
6. $B_n = \frac{a}{d}$

Step 2 is the von Staudt-Clausen theorem to determine the fractional denominator part of the Bernoulli number.

This method also has the same benefit as the explicit formula for Bernoulli numbers that can be generated in any order. In [4] they provide an illustrative example of a calculation that I have brought up below.

Example of Bernoulli number computation

Find the Bernoulli number B_{50} using 50 digits precision you get.

1. $K = \frac{2(n!)}{(2\pi)^n} = \frac{2(50!)}{(2\pi)^{50}} = 7500866746076957704747736.71552473316456403804367604$
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. $z=1.00000000000000008881784210930815902983501390146827$
5. $dKz=495057205241079648212477524.99999999442615111210652$
6. and therefore $B_{50} = \frac{495057205241079648212477525}{66}$

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

K and z from the above algorithm are arbitrary precision floating point numbers and the question arises of what kind of precision you would need to use in your calculation to get the correct result. In the example for B_{50} 50 digits 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:

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

Or

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

You get the equivalent magnitude of decimal digits to be:

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

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:

$$K = \frac{2}{2^n} \frac{n!}{\pi^n} = 2^{1-n} \frac{n!}{\pi^n} \quad (19)$$

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

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

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

And for the denominator:

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

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 $a_n = \text{Magnitude}(B_n) + \text{Magnitude}(d_n)$. However, it is not easy to determine an estimation priori of the magnitude of d_n in step 2. However, when calculating d_n you can take

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

$\log_{10}(\text{denominator})$ 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 $ B_n $
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 with:

$$\text{Working precision} = \text{Magnitude of } |B_n| + \text{Magnitude of } |dn| + n/8 \quad (22)$$

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) // Uninitialized 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
```


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

```

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

```

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

```
        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

Recommendation of 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

Similar to the Bernoulli numbers that are defined as:

$$\frac{t}{e^t-1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} t^n \quad (23)$$

You have the Bernoulli $B(x)_n$ polynomials defined as:

$$\frac{te^{tx}}{e^t-1} = \sum_{n=0}^{\infty} \frac{B(x)_n}{n!} t^n \quad (24)$$

The relationship between Bernoulli polynomials and Bernoulli numbers are:

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

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)
```

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

```
{
    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

The main reason for writing this document was to describe the different methods for the computation of the gamma function. However, some of the basic prerequisites are fast factorial, 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:

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

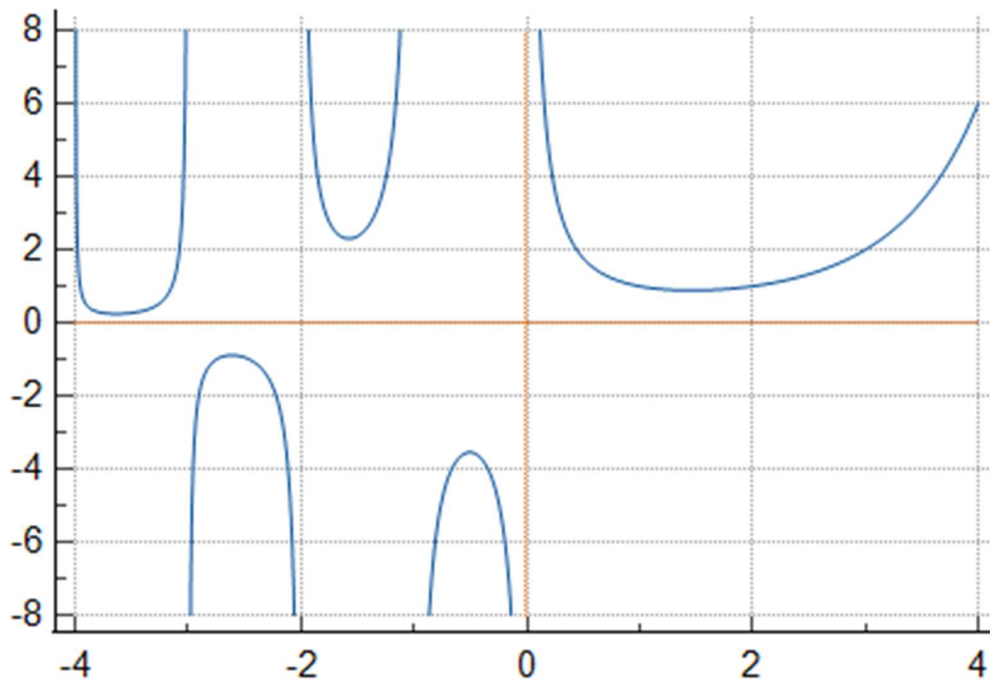


Figure 2. Gamma function in the interval [-4:+4]

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

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

$$\Gamma(z + 1) = z\Gamma(z) \quad (27)$$

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

$$\Gamma(n) = (n - 1)! \quad (28)$$

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

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

Or in the negative half of the real axis:

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

Another important equation is Euler's reflection formula:

$$\Gamma(z)\Gamma(1 - z) = \frac{\pi}{\sin(z\pi)} \Rightarrow \quad (31)$$

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

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 it to the real number x instead of 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)
```

Fast Gamma, Beta, Error, Zeta function & Bernoulli numbers for Arbitrary 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 Lanczos and the Stirling asymptotic method are global methods, whereas the integration by parts is a local method defined in the interval [1:2]. Of course, there are techniques to expand the local method to function as a global method.

Lanczos-Spouge method

Is Lanczos method from 1964 was modified by Spouge in 1994 which is a much simpler way to compute $\Gamma(x)$ and is very useful for arbitrary precisions arithmetic.

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

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

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

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

$$a = \left\lceil \left(P - \frac{\ln(P)}{\ln(10)} \right) \frac{\ln(10)}{\ln(2\pi)} - \frac{1}{2} \right\rceil \quad (34)$$

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 they further found out in [3] 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 give instead that the error is bounded by:

$$e_a(x) = \frac{1}{a^{0.5}(2\pi)^{a+0.5}} \quad (35)$$

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 $\sqrt{\quad}$
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)

```

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

```
        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);
        ekml = exp( float_precision(k-a, workprec));
        ekml *= float_precision(factorial(int_precision(k-1)), facprec);
        ck /= ekml;
        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;
    fp.precision(x.precision());
    return fp;
}
```

Stirling asymptotic series method

Stirling asymptotic series for Gamma function is given by:

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

$$\ln(\Gamma(x)) \sim \left(x - \frac{1}{2}\right) \ln(x) - x + \frac{1}{2} \ln(2\pi) + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n(2n-1)x^{2n-1}} \quad (36)$$

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

$$n_{\text{optimal}} \sim \pi|x| + 2 \quad (37)$$

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

$$(2\pi - 1)x + (x + 1)\ln(x) + 3.9 > P_{\text{max}} \cdot \ln(10) \Rightarrow \quad (38)$$

$$P_{\text{max}} = \left\lfloor \frac{(2\pi-1)x + (x+1)\ln(x) + 3.9}{\ln(10)} \right\rfloor \quad (39)$$

For a certain magnitude of $|x|$ we get:

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

This is discouraging since for small $|x|$ we cannot get a reasonable precision out of 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 $|x|$ needs to be more than 10 from the table above. Instead of calculating $\Gamma(x)$, we calculate $\Gamma(x + 10)$ and then divide $\Gamma(x + 10)$ 10 times as outlined:

$$\Gamma(x) = \frac{\Gamma(x+10)}{x(x+1)(x+2)(x+3)(x+4)(x+5)(x+6)(x+7)(x+8)(x+9)} \quad (40)$$

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

$$\Gamma(x) = \frac{\Gamma(x+M)}{\prod_{m=0}^{M-1} (x+m)} \quad (41)$$

Unfortunately, the above formula for P_{max} is not very accurate to generate the number of shifts needed and in general indicates a shifting that is not sufficient for the desired accuracy. Instead, we use from [6] that states the number of shifts needed as follows.

$$|x + \text{shifts}| = P * \frac{\ln(10)}{\ln(2)} * 0.11038 \Rightarrow \quad (42)$$

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

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).

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

Stirling Asymptotic method	
Pros	Cons
Accuracy is great for 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 $\sqrt{\quad}$
	Need to compute Bernoulli numbers
	Need to shift gamma value for small magnitude $ x $
	Slow Computation

Source of 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

```

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

```

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;
}

```

Integration by parts method

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

$$\Gamma(x) = \int_0^M t^x e^{-t} \frac{dt}{t} + \int_M^\infty t^x e^{-t} \frac{dt}{t} \quad (44)$$

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:

$$\Gamma(x) \approx \int_0^M t^x e^{-t} \frac{dt}{t} = M^x e^{-M} \sum_{n=0}^{\infty} \frac{M^n}{\prod_{k=0}^n (x+k)} \quad (45)$$

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

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

$$M > (P + \ln(P))\ln(10) \quad (46)$$

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

$$N_{\max} = P \frac{\ln(10)}{W\left(\frac{1}{e}\right)}, \text{ where } W \text{ is Lambert's function } W\left(\frac{1}{e}\right) \approx 0.2785 \quad (47)$$

With that we have the final formula:

$$\Gamma(x) \approx M^x e^{-M} \sum_{n=0}^{N_{\max}} \frac{M^n}{\prod_{k=0}^n (x+k)} \quad (48)$$

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

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

$$\Gamma(x) = \prod_{m=0}^{k-1} (x-m) \Gamma(x-k) \quad (49)$$

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)!
```

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

```
// 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)*n!)
// 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;
}
```

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

Gamma Performance

The performance in the graph below indicates the best-performing method is the method with integration by parts. It consistently performs better than Stirling and the Lanczos-Spouge method. Notice that Stirling is measured with a small, medium, and large argument. The small argument is approx. 20 times slower than for the medium and large arguments. This is deceptive and the reason is that Stirling-small was first computed and it builds up the Bernoulli numbers that are cached so when Stirling-Medium and Stirling large are called then the Bernoulli number is already in the cache. However, it clearly shows that the method is very slow when Bernoulli numbers are not pre-calculated but even when cached the Stirling method was significantly slower than the two other methods.

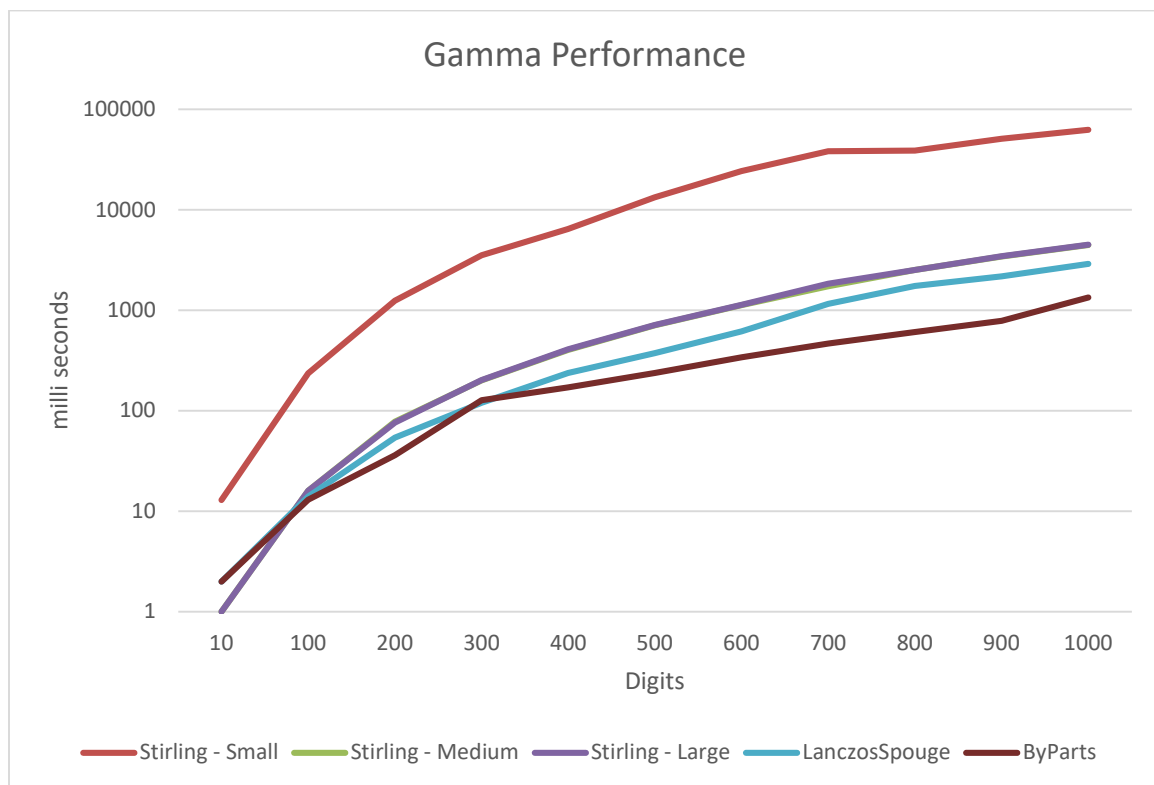


Figure 3. Gamma Performance

Recommendation for the Gamma function

The method with integration by parts is the simplest and fastest method even when this method 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, the method is not recommended even if the Bernoulli number is pre-calculated it is still significantly slower than the two other methods. The Stirling-small argument does

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

compute the Bernoulli number while Stirling-Medium and Stirling-Large use the cached Bernoulli number.

The Beta function

The beta function is defined by the integral:

$$B(z, w) = \int_0^1 t^{z-1} (1-t)^{w-1} dt \quad (50)$$

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

One of the nice things about the Beta function is that its related to the Gamma function:

$$B(z, w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \quad (51)$$

Since we in the previous section have described the Gamma function we can easily implement the Beta function using the Gamma function (tgamma in the C library)

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 is used within statistical computations and many other fields. The error function $\text{erf}(x)$ is defined as:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (52)$$

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

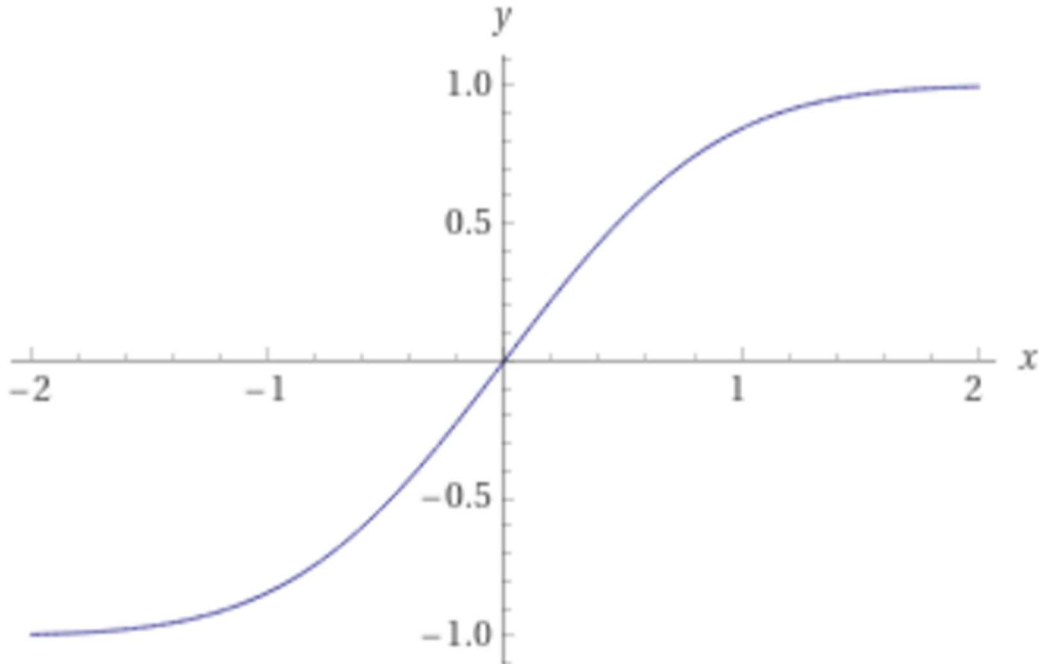


Figure 4. Error function in the interval [-2:+2]

The error function is symmetric meaning that $\text{erf}(-x) = -\text{erf}(x)$.

The complementary error function is defined as:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 1 - \text{erf}(x) \quad (53)$$

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

$$\text{Formula 1: } \text{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n+1)n!} \quad (54)$$

$$\text{Formula 2: } \text{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}, \text{ !! is the double factorial} \quad (55)$$

$$\text{Formula 3: } \text{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n-1)!!}{(2x^2)^n}, \text{ !! is the double factorial} \quad (56)$$

Formula 1 and 3 have the weakness of using alternating signs between the terms, while formula 2 needs a calculation of the exponential function e^x as well. Using alternating sign in a summation always give rise to cancellation errors and thereby lack of accuracy if not controlled. In [9] they give a details explanation of each method together with an error bound for each formula and a practical implantation guide for the formula. For all three methods, you don't need to know how many terms you would need in the summation. You can just continue until the next term is below the requested precision for

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

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

$$n > 1 + e^{\left(1 + W\left(\frac{P \cdot \ln(10)}{e}\right)\right)}, \text{ } W \text{ is the Lambert's } W \text{ function} \quad (57)$$

In [9] they also give an error bound for each formula and the readers kindly refer to [9] for the details. Furthermore, in [9] they also recommend using concurrent series summation instead of the straight forward way. This is a little bit more complicated to implement but [9] gives a detailed explanation of how to do it properly.

In regards to formula 3 which also suffer from alternating sign, I found it easier to just use the identity: $\text{erfc}(x) = 1 - \text{erf}(x)$ and rely on just a single solid $\text{erf}()$ implementation. Another deficit of formula 3 is that the achievable accuracy depends on the magnitude of the number. In [9] they found that the achievable accuracy for P decimal digits was:

$$P \sim e^{2 \ln(x) \frac{\ln(2)}{\ln(10)}} \quad (58)$$

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 implementation and the implementation using concurrent series. They should only be compared pairwise. But in all cases, the method using concurrent series is many times faster than the straightforward approach.

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

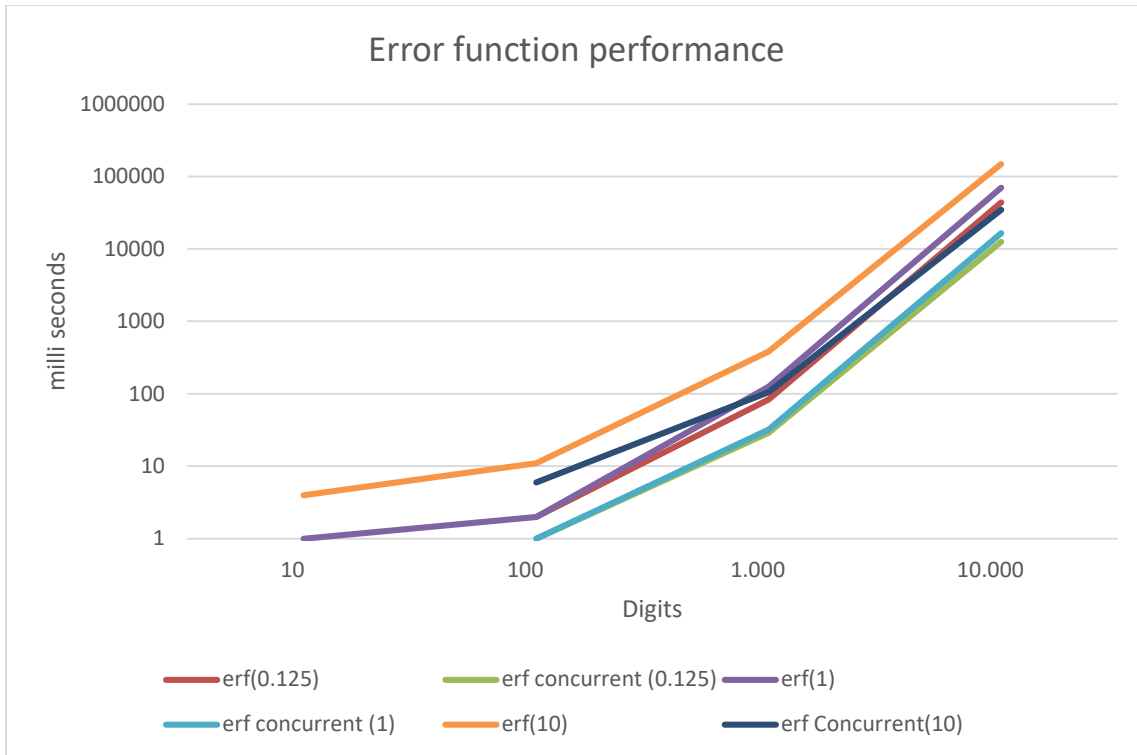


Figure 5. Performance of the Error function

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)))/x^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));
    }
}
```

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

```
    }

    eps = -eps; eps += std::min(intmax_t(0), x.exponent() - 1);
    // Compute L. Optimal with  $L \sim \sqrt{N}$ 
    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); //  $|x| \cdot \exp(-x^2)/\sqrt{\pi}$ 
    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 a 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 Horner'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 due to not using alternating signs between the terms and even though it requires a computation of the exponential function it is still significantly faster than any of the other methods and works well for both large magnitudes of x and smaller magnitudes of x .

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

Lambert W function

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.

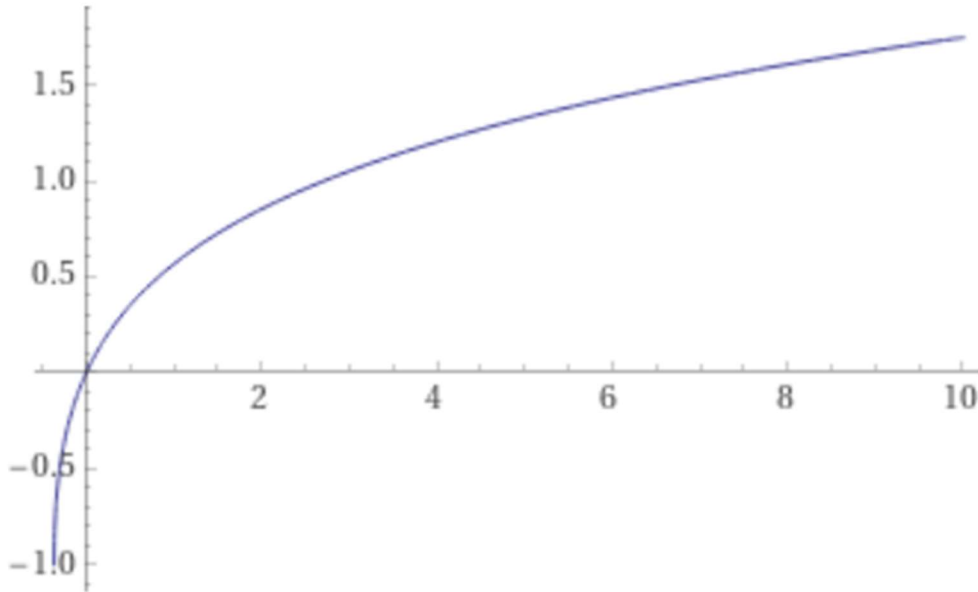


Figure 6 Lambert W function $w_0(x)$ in the interval $]-1/e;10]$

There are 3 different methods suitable for arbitrary precision. 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 start point for all three iterative methods we will describe it first and then address each of the above methods.

A Suitable starting point for Lambert W Iteration.

We do not usually have a Lambert W function available where we easily can get the first 15-16 digits accuracy using the build-in double type in C or C++. If you look in the literature they usually suggest a starting point for Lambert W function as follows.

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

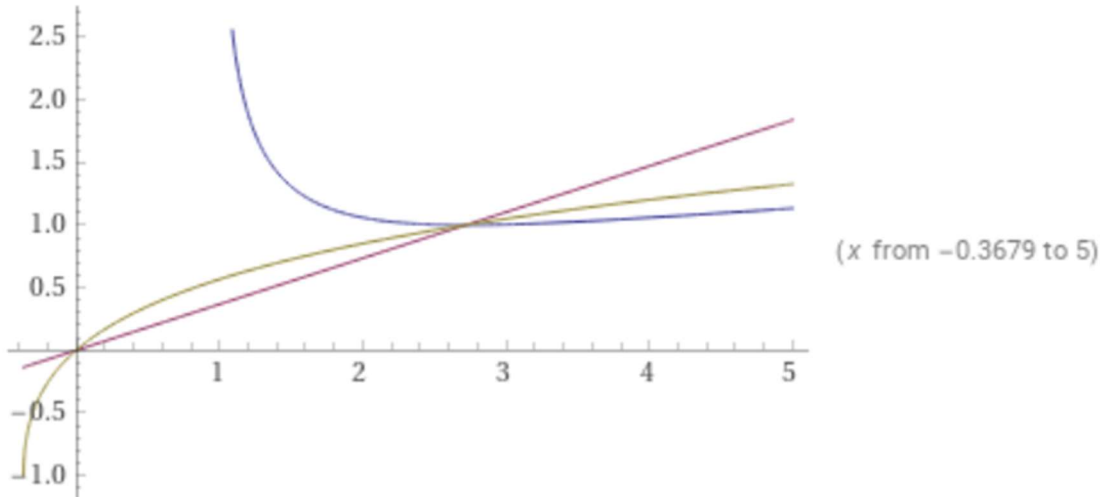


Figure 7 amber color is $W_0(x)$

If x in $[e, \infty[$:	$w_0(x) = \ln(x) - \ln(\ln(x))$	“blue line in the above figure
If x in $[0, e[$:	$w_0(x) = \frac{x}{e}$	“magenta line in 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 iteration:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n - x}}{e^{w_n} + w_n e^{w_n}} \quad (59)$$

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:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n - x}}{e^{w_n(w_n+1) + \frac{(w_n+2)(w_n \cdot e^{w_n - x})}{2(w_n+2)}}} \quad (60)$$

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

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

Boyd quadratic method

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

$$w_{n+1} = \frac{w_n}{1+w_n} \left(1 + \ln\left(\frac{x}{w_n}\right)\right) \quad (61)$$

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

Initial performance of Lambert W function

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

However, there is a technique for speeding up this classic iterative method which is described in [10]. Instead of iterating with the use of full precision for each iterative variable. We instead dynamically change them as we iterate to be able to accommodate the target precision for each iteration step. E.g. if we need 1,000 digits precision of Lambert $W(x)$ we do not need more than 20 digits for the first 5 iterations (remember that our initial guess was around an accuracy of 10^{-1} and then we can gradually increase them to our target precision of 1,000 digits over the next 6 iterations. It would not be very wise to carry these first 5 and subsequent iterations with a precision of 1,000 digits. Needless to say that 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 startpoint point
    wn = x;
    if (x >= float_precision(e1)) // xin [e, infinity[
    {
        // start point = log(wn)-log(log(wn))
        // remove exponent to allow the startpoint 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);
    }
}
```

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

```

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 explsq = sqrt(1 + ex);
            ex *= log(1 + explsq) / (1 + ex + explsq);
            wn = float_precision(ex);
        }

    // Newton quadratic iteration
    for (icnt = 0;; ++icnt)
    {
        wtmp = exp(wn); // only evaluate 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 fit into 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 startpoint
    wn = x;
    if (x >= float_precision(e1)) // xin [e,infinity[
    {
        // start point = log(wn)-log(log(wn))
        // remove exponent to allow the startpoint being 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

```

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

```

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 evaluate 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 fit into 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 startpoint point
    wn = x;
    if (x >= float_precision(e1)) // xin [e,infinity[
    {
        // start point = log(wn)-log(log(wn))
        // remove exponent to allow the startpoint being 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)
    }
}

```

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

```
    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/wn))
    wn += c1; // wn=wn+1
    wn = wtmp / wn; // wn*(1+log(x/wn))/(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 somehow similar but the Boyd methods begin to takeoff after approx. 6,000 digits and is thereafter the fastest method. See the performance chart below.

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



Figure 8. Performance of Lambert W(x) function

Recommendation for Lambert W function

- The preferred method is Boyd’s method.
- Although Halley is close after. However, if your arbitrary precision has a faster $\exp(x)$ function than $\log(x)$ then the Halley method is the preferred one.

Riemann Zeta function

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

$$\zeta(z) = \sum_{n=0}^{\infty} \frac{1}{n^z} \tag{62}$$

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

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

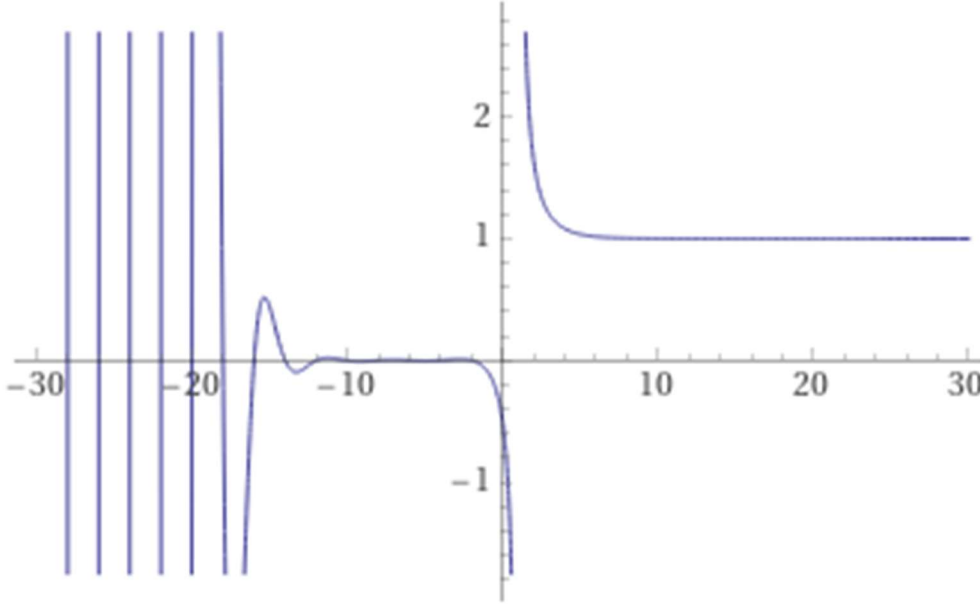


Figure 9 Zeta(x) in the interval [-30:+30]

There are several interesting identities. One of them is this formula that is useful for negative, s that map s , into the positive real axis.

$$\zeta(1 - s) = \frac{2\Gamma(s)}{(2\pi)^s} \cos\left(\frac{\pi s}{2}\right) \zeta(s) \quad (63)$$

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 $s > -(n-1)$.

$$\zeta(s) = \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s} \quad (64)$$

Where e_j is defined as:

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} \right) - 2^n \quad (65)$$

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

$$n = \left\lceil \frac{\ln(10)}{\ln(8)} P \right\rceil \quad (66)$$

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 has to be repeated $2n$ times. The power function

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

x^y requires a call to both $\log()$ and the $\exp()$ function, if s , is not an integer and is therefore a very expensive 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

$$s > 1 + P \frac{\ln(10)}{\ln(P)} \quad (67)$$

Where P , is the target precision (in decimal digits) is met. We can resort to the below series for faster computation.

$$\zeta(s) \approx \sum_{k=0}^N \frac{1}{k^s} \quad (68)$$

And a suitable value for N is:

$$N = \left\lceil 10^{\frac{P}{s-1}} \right\rceil \quad (69)$$

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:

$$\zeta(0) = -\frac{1}{2} \quad (70)$$

$$\zeta(-n) = (-1)^n \frac{B_{n+1}}{n+1} \quad (71)$$

$$\zeta(2n) = (-1)^n \frac{B_{2n}(2\pi)^{2n}}{2(2n)!} \quad (72)$$

For odd positive integers, there is unfortunately not an easy formula, however, there is a myriad of series you can find in various published papers that promise faster than the above general formula for zeta.

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

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} \right) - 2^n \quad (73)$$

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:

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \binom{n}{k} \right) - 2^n \quad (74)$$

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

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

$$\begin{aligned} \text{if } j < n: \text{sum}_j &= 0 \\ \text{if } j \geq n: \text{sum}_j &= \text{sum}_{j-1} + \binom{n}{j-n} \end{aligned}$$

And e_j now becomes:

$$e_j = (-1)^j (\text{sum}_j - 2^n) \quad (75)$$

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

We can now present our final algorithm for the $\zeta(s)$ function

Algorithm 3 for $\zeta(s)$ 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 (70)
  if s even?
    return compute (71)
// s is real or s is odd integer goes here
if s < 0.5?
  return compute (62)
if s > 1+P·log(10)/log(P) // if s large?, P is the decimal precision required
  return compute (67)
// s is > 0.5 but not large
  return compute (63)
```

Source for zeta function computation.

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

```
static float_precision zeta(const float_precision& x)
{
    const size_t precision = x.precision(); // Target precision
    const intmax_t n = intmax_t(ceil(precision * log(10) / log(8)));
    // Working precision
    const size_t workprec= precision + 10 + (size_t)ceil(log10(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();
}
```

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

```

// 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
    // Get the integer as an int_precision integer
    int_precision ip = int_precision(res);
    fraction_precision<int_precision> B;

    if (ip.sign() < 0) // Negative?
    {
        if (ip.even()) // is even?
            // Return zero for all negative even integers
            return float_precision(0, precision);

        // Odd negative ends here
        B=bernoulli((size_t)1+(size_t)ip); // Get Bernoulli(n+1)
        // Bernoulli(n+1)/(n+1)
        B *= fraction_precision<int_precision>(1,int_precision(1) - ip );
        // Convert fraction_precision to float_precision
        res = float_precision(B.numerator(), workprec);
        res /= float_precision(B.denominator(), workprec);
        if (ip.odd()) // n is odd?
            res.change_sign(); // 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)
        // Bernoulli(n)/(2*n!)
        B *= fraction_precision<int_precision>(1,2*factorial(ip));
        // 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;
        if (ip.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)
}

```

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

```
    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, unusing 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 jsum
}
// 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;
}
```

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

Reference

- 1) Arbitrary precision library package. [Arbitrary Precision C++ Packages \(hvks.com\)](http://hvks.com)
- 2) Numerical recipes in C++, 3rd edition, Cambridge University Press, New York, NY 2007
- 3) The Yacas book of algorithm, 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 \(hvks.com\)](http://hvks.com)
- 11) P. Borwein, "An efficient Algorithm for the Riemann Zeta function", Canadian Mathematical Society, Conference paper.