The Math behind arbitrary precision for integer and floating point arithmetic.

By Henrik Vestermark (hve@hvks.com)

Abstract:

We are all used to the fast microprocessors available nowadays and the computational speed of basic arithmetic, trigonometric or logarithmic functions is done at a lightning-fast speed. However, when building arbitrary integers and floating point packages, that can handle decimals in the range from a few to several million digits it is all back to the basic of math to build an arbitrary precision package with reasonable speed.
This paper describes the underlying math behind this package and is a completely updated and expanded version of the original paper from 2013.

Introduction:

Building an arbitrary software package that can handle all arithmetic for integers and floating points for any precision, is down to the basics of simple math. This paper describes what formula and math that lies behind the arbitrary precision packages starting with arbitrary integer precision followed by the floating-point math. For the floating-point math when a floating-point number is broken down to its base component of <integer>, <fraction>, and <exponent> it too uses the integer precision math to do the calculation for the floating point. After the basic floating-point operators like addition, subtraction, multiplication, and division, we build upon these functions to implement √, Logarithm and exponential functions continuing with Trigonometric and Hyperbolic functions, and finalize the paper with the more exotic functions, like Gamma, Beta, Error, Zeta and Lambert function. For each of the functions, there is a description of various optimization technics to improve performance particularly when needed precision exceeds 100 digits and goes into the million digits precision and higher.

Change log

23-February 2023. Correcting Grammar and minor corrections plus added a new section about the Gamma, Beta, Error, Lambert, and Zeta functions and the following special constants, Euler-Mascheroni, Catalan, and Apery Zeta(3).
15-January 2023. Updated some inconsistency in the "Cos(x) using sin(x)" section and corrected the recommendation in the same section.
24-October 2022. I have updated the entire document with new and updated content. It grows from 40 pages to more than 100 pages. The previous 2013 version has become outdated and I have written several papers after 2013 discussing various methods to use for elementary functions like exp(x), log(x), trigonometric functions, hyperbolic functions, and various constants like e, ln(2), ln(10) and π. This has now been consolidated into this version of the Math behind arbitrary precision arithmetic. The more details papers that all contain a reference source code in C++ can all 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
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

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handle arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision i;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.
One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for int_precision variables

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

```
vector<uintmax_t> mNumber;
```

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

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

The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

Float_precision class

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 the float or double you just replace the type name with *float_precision*. E.g.

```
float_precision f;  // Declare an arbitrary precision float with default precision
```

You can add a few parameters to the declaration. The first is the optional initial value, and the second parameter is a 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(f.precision()-10);  // Lower the precision with 10 digits
f.precision(f.precision()+20); // Increase precision with 20 digits
```

There is another method to manipulate the exponent of the variable. 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 2^exponent
f.exponent(0)          // Remove the exponent
f.exponen(16)          // Set the exponent to 2^16
```

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. There are additional methods, 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 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

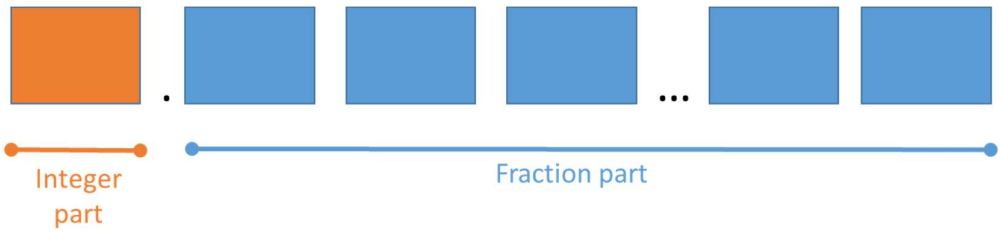
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
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 mBinary



- 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.

The normalized number for float_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. See [1].

Integer Arithmetic

In integer arithmetic, we use the notation i_n for an *n*-digit integer number *i* where *n* is greater or equal to zero. In our description, we assume the integer is in base 10 to simplify the description of the math behind the scenes. However, in our actual implementation of the arbitrary precision packages, we use binary digits for better storage utilization (base=2^{64} in a 64-bit environment and base=2^{32} in a 32-bit environment).
Also, we denote *i[n]* as the most significant digit of *i* and *i[0]* as the least significant digit of *i*. The integer i_n can also be described for any given base as:

$$i_n=i[n]\beta^n+ i[n-1]\beta^{n-1}+\ldots+ i[2]\beta^2+ i[1]\beta^1+i[0]\beta^0$$

For Base β=10 you get:

$$i_n=i[n]10^n+ i[n-1]10^{n-1}+\ldots+ i[2]10^2+ i[1]10+i[0]$$

For Base β=2^{64} you get:

$$i_n=i[n](2^{64})^n+ i[n-1](2^{64})^{n-1}+\ldots+ i[2](2^{64})^2+ i[1](2^{64})+i[0]$$

For the number i_n the notation i[p] for p > n always return 0.

Addition:

To implement addition we use the simple schoolbook method by adding each digit starting from the least significant digit of the number to the highest.

Consider two positive integers a_n and b_m, the result c_k of adding a_n and b_m together are:

Algorithm: addition.

```
BASE=2^64
carry=0
for(i=0..max(n,m))
        c[i]=(a[i]+b[i]+carry)%BASE
        carry=(a[i]+b[i]+carry)/BASE
if(carry !=0)
        c[max(n,m)+1]=carry
```

If either a_n or b_m is negative we resolve the sign using the below table

+	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n+b_m$	$C=a_n-\|b_m\|$
$a_n < 0$	$C=b_m-\|a_n\|$	$C=-(\|a_n\|+\|b_m\|)$

Subtraction:

To implement subtraction we again use the simple school book method by subtracting each digit starting from the least significant digit of the number to the highest.

Consider two positive integers a_n and b_m the result c_k of subtracting a_n and b_m are:

Algorithm: Subtraction

```
BASE=2^64
carry=BASE
for(i=0..max(n,m))
        carry=(BASE-1+a[i]-b[i]+carry/BASE)
        c[i]=carry%BASE
if(carry<BASE)
        // c is negative
else
        // c is positive
```

If either a_n or b_m is negative we resolve the sign using the below table

-	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n-b_m$	$C=a_n+\|b_m\|$
$a_n < 0$	$C=-(\|a_n\|+b_m)$	$C=-(\|a_n\|+\|b_m\|)$

Multiplication:

Multiplication is also trivial

$$a_n * b_m = c_{n+m}$$

And as for the multiplication, we divide the case into two scenarios: one where m=1 and one where m>1. For m=1 we use the iteration:

Algorithm: Multiplication with a single digit

```
BASE=2^64
carry=0
for(i=0..n)
        c[i]=(a[i]*b[0]+carry)%BASE
        carry=(a[i]*b[0]+carry)/BASE
if(carry!=0)
        c[n+1]=carry
```

For m>1 we repeatedly use the above mention formula for multiplying a single digit and the addition of the intermediate results.

Algorithm: Multiplication

```
BASE=2^64
```

```
ck=an*b[0]              // Single digit multiplication
for(i=1..m)
        tmp=an*b[i]     // Single digit multiplication
        ck=ck+tmp*BASE^i
```

Notice that multiplying the intermediate result with $BASE^i$ is easy since you just post-fix the temp result with *i* number of zeros. The above algorithm is of complexity $O(n^2)$ and is typically referred to as school book multiplication.

If either a_n or b_m is negative we resolve the sign using the below table

*	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n * b_m$	$C=-(a_n * \|b_m\|)$
$a_n < 0$	$C=-(\|a_n\| * b_m)$	$C=\|a_n\| * \|b_m\|$

Schoolbook multiplication is not the fastest way to do multiplication and is easily beaten by the performance of other multiplication methods. A few others are relevant to consider for multiplication:

- Karatsuba
- Toom-Cook 3
- Linear convolution
- Fast Fourier Transformation (FFT)
- Schönhage-Strassen
- Fürer's method

Karatsuba

Invented in 1960 by A. Karatsuba. Before that, it was believed that it could not be faster than the schoolbook multiplication. Karatsuba showed that you can reduce the multiplication of two n-digit numbers to three multiplication and two add/subtraction instead of the usual four multiplication.

Algorithm: Karatsuba multiplication

```
function karatsuba(a,b)
        if(a<10|| b< 10)
                Return a*b      // Do multiplication of small numbers directly
        m=Numberofdigit(a)  // NumberofDigit() return the number of digits in base 10
        if(m>NumberofDigit(b))
                m=NumberofDigit(b)
        m=integer(m/2)

        // Splitting
        [ahigh,alow]=split(a,m)         // Split a into two half ahigh and alow
        [bhigh,blow]=split(b,m)         // Split b into two half bhigh and blow

        // Evaluation
        z0=karatsuba(alow,blow)
        z1=karatsuba(alow+ahigh,blow+bhigh)
        z2=karatsuba(ahigh,bhigh)
```

```
        // Recomposition
        return (z2*10^(2*m))+((z1-z2-z0)*10^m)+z0
```

Karatsuba algorithm reduces the complexity to $O(n^{1.58})$

Toom-Cook

The toom-Cook algorithm was invented in 1963 by A. Toom and S. Cook. Instead of splitting the number into two halves as used by Karatsuba, they could split it into any number k, however with increasing complexity. Karatsuba algorithm is equivalent to k=2 and named Toom-Cook-2. The most common variation is splitting the number into 3 parts (Toom-Cook-3), however, GNU arbitrary precision also offers four parts splitting (Toom-Cook-4).

The complexity of Toom-Cook-3 is $O(n^{1.46})$ and Toom-Cook-4 is $O(n^{1.40})$

Algorithm: Toom-Cook-3

```
function toomcook3(a,b)
        if(a<10|| b< 10)
                return a*b       // Do multiplication of small numbers directly
        m=Numberofdigit(a)  // NumberofDigit() return the number of digits in base 10
        if(m>NumberofDigit(b))
                m=NumberofDigit(b)
        m=integer(m/3)

        // Splitting
        // Split a into three half ahigh, amid and alow
        [ahigh,amid,alow]=split3(a,m)
         // Split b into three half bhigh, bmid, and blow
        [bhigh,bmid,blow]=split3(b,m)

        // Evaluation
        p1=alow+ahigh+amid
        p2=alow+ahigh-amid
        p3=2(p2+ahigh)-alow
        q1=blow+bhigh+bmid
        q2=blow+bhigh-bmid
        q3=2(q2+bhigh)-blow

        // Pointwise multiplication
        i0=toomcook3(alow,blow)
        i1=toomcook3(p1,q1)
        i2=toomcook3(p2,q2)
        i3=toomcook3(p3,q3)
        i4=toomcook3(amid,bmid)

        // Interpolation
        i3=(i3-i1)/3
        i1=(i1-i2)/2
        i2=-i0
        i3=(i2-i3)/2+2*i4
```

```
        i2=i2+i1-i4
        i1=i1-i3

        // Recomposition
        I1=i1*10^m
        I2=i2*10^2m
        I3=i3*10^3m
        I4=i4*10^4m
        result=i0+i1+i2+i3+i4
        if(sign(a)*sign(b)<0)
                result=-result
        return result
```

Linear Convolution

Here we do a linear convolution of the two numbers by first splitting the binary number up into a vector of 8-byte numbers. Then do a pointwise multiplication where we perform the carry operations of the linear convolution and then the last step is to combine the 8-byte number back into a binary number.

Algorithm: Linear convolution

```
function linear_convolution(a,b)
        // Split the binary number a into an 8-byte vector of binary numbers
        vec_a=split(a)
        // Split the binary number b into an 8-byte vector of binary numbers
        vec_b=split(b)
        size_a=size(vec_a)
        size_b=size(vec_b)
        for(i=0..size_a)
                for(j=0..size_b)
                        vec_linear[i+j]+=vec_a[size_a-1-j]*vec_b[size_b-1-i]
        nextCarry=0
        for(i=0..size_a+size_b-1)
                vec_linear[i]+= nextCarry
                nextCarry=vec_linear[i] / 256
                vec_linear[i] = vec_linear[i] % 256
        if(nextCarry>0)
                vec_linear[i+1]=nextCarry % 256
        // Combine the 8-byte binary number to a binary number
        return combine(vec_linear)
```

In the algorithm above we use a sample size of a single byte. It is not optimal to split floating-point variables into chunks of bytes. You can improve the performance by sampling at 16-bit or 32-bit at a time. The benefit is that the double for loop needs to pass through fewer entries. However, you risk an overflow even when using 64-bit internal multiplication. If you use 16-bit or 32-bit sampling size you would need to be able to handle overflow which needs to be propagated in the vec_linear vector. Our implementation in our arbitrary precision library uses a sample size of 32-bit.

Fast Fourier Transformation (FFT)

It is beyond the scope of this paper to explain the theory behind FFT multiplication, but readers can reference [2] for more information. However, you convert the two numbers a_n and b_m via a series of Fourier transformations then *multiply* them together and do the inverse Fourier transformation of the result back to the digital domain.
FFT complexity is O(n*log(n)) and thereby preferable over the other multiplication method.

Algorithm: FFT

```
function FFT(a,b)
        length=max(size(a),size(b)
        for (n = 1; n < length; n <<= 1) ;
        n <<= 1;                    // Ensure n is a true power of 2 and larger than length
        vec_a=split(a)              // Split an into bytes and return it as a vector of doubles
        vec_b=split(b)              // Split bm into bytes and return it as a vector of doubles
        DFT(vec_a,n,1)              // 1 indicate forward transformation of a vector
        DFT(vec_b,n,1)              // 1 indicate forward transformation of b vector
        // Do multiplication and stored the result in vec_b
        vec_b[0] *= vec_a[0]
        vec_b[1] *= vec_a[1]
        for (i = 2; i < n; i += 2)
                t = vec_b[i]
                vec_b[i] = vec_b[i]*vec_a[i]–vec_b[i+1]*vec_a[i+1]
                vec_b[i+1] = t*vec_a[i+1]+vec_b[i+1]*vec_a[i]
        DFT(vec_b,n,-1)             // -1 indicate Reverse transformation
        // propagate carry
        for (cy = 0, i = 0; j <= n - 1; ++i)
                tmp = vec_b[n-1-i]/(n>>1)+cy+0.5
                cy = (unsigned long)(tmp/256)  // Byte Radix 2^8=256
                vec_b[n-1-i] = tmp-cy*256;
        result=compose(b)    // Compose it back to the binary number format
        return result
```

Note: the DFT function is the discrete Fourier transformation, see [2] for the actual C source code.

Limitation of FFT

FFT uses floating-point arithmetic using the double type in C++. In the initial step, you would need to map the vector of binary 64-bit digits into a double. You do that by splitting each 64-bit binary digit into eight bytes and then converting each byte into a double. Because we use floating-point arithmetic in FFT we need to be careful with how large the two numbers we multiply can be. In [2] they give a formula for the number of decimal digits *n*, the FFT can handle without inaccuracy in the result as a function of the initial splitting into bytes and how large our double mantissa is in bits (53-bit in a C++ double)

$$\log_2((\text{Sample size})^2) + \log_2(n) + k \cdot \log_2(\log_2(n)) < 53 \qquad (1)$$

E.g., a byte (8-bit) has a sample size of $2^8=256$. Where k is "a few". Let's choose k=2 we get for n=100,000,000 (100 million) that 52<53 which is true.

If we solve the above equation for *n*, we get an *n* of approx. 175M digits. Unfortunately, k as two is not enough since we begin to get random errors in the multiplications result around

multiplication with 125M digits. Instead, I recommend you use k as a factor of 2.15 and you get a maximum limit of multiplication size of around 116M digits.
That leads to the question of what to do if you need a higher number of digits in multiplications. An easy fix is to lower the sample size to 4-bit instead of 8-bit. Using the above formula again, you get a maximum of 17.8 billion digits you can multiply. If that is still not enough you can do a 2-bit sample size and get a limitation of 229 billion digits. Every time you lower the sample size by half it will increase the memory requirements by four reducing the overall performance of FFT multiplication. If you go the other way and use a sample size of 16-bit, you get a limitation of approx. 8,396 digits, which is excessively small to be useful in practice.

The limitations can vary depending on which system you are running and if they support floating-point arithmetic above 64-bit. IEEE-754 does specify an extended 80-bit version sometimes going under the name of long double. (Not all compiler supports it, so the author's arbitrary precision packages only support the standard 64-bit double)

Limitations in FFT

C++ type	Double	Long double*
Bits in Mantissa	53-bit	64-bit
Sample size		
16-bit	8,396	5.3M
8-bit	116M	120B
4-bit	17.8B	20T
2-bit	229B	280T

*) Not supported on all compilers and systems

Schönhage-Strassen

This algorithm was invented in 1971 by A. Schönhage and V. Strassen. The Complexity is O(n*log(n)*log(log(n))). Again it is beyond the scope of this paper to give a detailed introduction to this method.

Fürer's Method

Has an even lower complexity than Schönhage-Strassen. However, is usually not seen in practice since it is only faster when dealing with extremely high precision.

Division & Remainder:

There are several approaches you can take to calculate the division.
In its simple form solving:

$$\frac{a_n}{b_m}$$

You can repeatedly subtract b_m from a_n until the condition $a_n<b_m$ is met and then the number of times you could subtract b_m is the integer result of this division. a_n is called the *dividend (or numerator)* and b_m is called the *divisor* or *denominator*. The result of the division let's call it c_k is the *quotient* of the division. If the continuing subtraction of b_m into a_n does not

result in a_n=0 then a_n contains the remaining portion of the division and lets called is d_j. For shorthand, this is sometimes written as:

$$\frac{a_n}{b_m} = c_k \; REM \; d_j$$

If b_m is a single digit we do it by school book manner by dividing the single digit b_0 into a_n starting at the most significant digit of a[n]. The result is the most significant digit of the quotient *c[k]*. Then add the remaining of that division into a's second most digit and repeat the process until all a_n digit has been divided. Now c_k is the quotient of the division and the last remaining digit is then d_j.

Algorithm. Division with a single digit

```
BASE=2^64
rem=0
for(i=0..n)
        c[i]=(BASE*rem+a[i])/b[0]
        rem=(BASE*rem+a[i])%b[0]
```

Now if b_m is more than a single digit *(m>1)* we could resort to the process of subtracting b_m from a_n.

Algorithm. Division & Remainder

```
ck=0
while(an>bm)
        an=an-bm
        ck=ck+1
dj=an           // ck is the result of the division and dj is the remainder
```

However, we quickly find out that we will run into a problem when dividing a large number a_n that is several magnitudes higher than b_m. E.g. let's assume that a_n is a number with 8 digits or a_8 magnitude is in the range of 10^8 and b_m is a two digits number of magnitude 10^2 then you will have to loop through the subtraction approx. 10^{8-2} or 10^6 times which is doable but time-consuming. If instead, we are dealing with a number a_n that is a 100-digit number then the looping will be in the order of 10^{98} Subtractions, even if we can do a subtraction in 10^{-6} seconds then it will still take us 10^{+92} seconds or approx. 10^{83} years, which clearly will get us nowhere.

Instead, we use the fact that multiplication is much faster than division. Let's say that a_n is an *n*-digit number and b_m is an *m*-digit number and of course n>m then instead of subtracting b_m we try to subtract b_m*$BASE^{n-m}$. If b_m*$BASE^{n-m}$ is less than a_n then we have replaced $BASE^{n-m}$ subtractions with one subtraction and one multiplication. This subtraction effectively ensures that the number a_n now is one digit less than *n* and the next subtraction can then be with BASE*$BASE^{n-1-m}$ Repeating this process you get an approx. the number of loops and operation of the multiplication and subtraction as ~ 2(n-m). We cannot always assume that b_m*$BASE^{n-m}$ is less than a_n in which case we subtract b_m*$BASE^{n-m-1}$ instead. This lead to a worst-case scenario that is 10 times higher than the approximation we found before or ~ 20(n-m). Therefore, instead of 10^{92} seconds, we have reduced the workload to something around ~ 0.002 seconds. If a_n is a number with 1 million digits the time will be ~ 20 seconds or 1 billion digits it will be 20,000 seconds, fast but not fast enough. Therefore, we use the last trick we have for division and that is to use the iterative method for division used with

floating point division. (See floating point division). We simply convert our integer numbers a_n and b_m to floating point numbers with *n* decimals, do the division using the iterative division method (describe later) and then convert the division result back to an integer. Instead of linear scaling of operations with the number of digits, we get a logarithm scaling of the number of operations, which is of course much faster.

If either a_n or b_m is negative we resolve the sign using the below table

/	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n/b_m$	$C=-(a_n/\|b_m\|)$
$a_n < 0$	$C=-(\|a_n\|/b_m)$	$C=\|a_n\|/\|b_m\|$

Proper detection of a carry

We have not addressed the issue of proper detection of a carry. The above algorithm for addition, subtraction, and division assumed that we could detect overflow by just dividing the operations performed with the BASE, which in our case is 2^{64}. In a 64-bit environment, the largest unsigned number is 2^{64}-1, which means we technically cannot perform the algorithms stated above. For the algorithm to work the BASE, need to be less than 2^{64}. Since you can define the arbitrary precision arithmetic with any BASE you can just choose a "nice" number less than 64.e.g. BASE=2^{60}. That way you have allocated the four top bits of a 64-bit integer to be used for carry detection. Of course, this is less storage efficient. Consider a 1,000 decimal number in a 64-bit environment. If BASE=2^{60} it will require a vector of 56 64-bit integers to store the number. However, if BASE=2^{64} it will require only 52 64-bit integers to store the number. This means that a BASE of 2^{60} will require ~7.7% more storage to hold it over a BASE of 2^{64}. Since we prefer to make the best utilization of available storage, we would prefer a BASE of 2^{64}. Now how do we detect overflow in such an environment? We can use a trick here that if you add two numbers e.g. c=a+b. If the addition overflow it will always result in c less than either a or b and we can use that fact to test for overflow.

Algorithm: Carry detection in addition.

```
c=a+b
if(c<a) // It could also be: if(c<b)
        carry=1
else
        carry=0
```

Example: Assuming a BASE=10 (single-digit system)

```
a=6, b=4                // Will result in overflow
c=a+b=6+4=~~1~~0        // The 1 digit is discarded and stroked out
if(c<a)                 // 0<6
        carry=1         // Yes carry was detected
else
        carry=0         // No vary was not detected.
```

if a=5 then c=9 and 9<6 is false and therefore no carry is detected.

However, in our algorithm, we did have two addition: a+b+carry (the previous carry propagated forward). Technically we can get an overflow from either tmp=a+b or c=tmp+carry but not both. That is easy to convince yourself of. Assuming BASE=10 as

before and a=b=9 (maximum single digit) the first addition tmp=a+b=9+9=~~1~~8 with a carry detected and the result is 8. Now carry can have either a zero or one value and even with carry = 1 the next addition tmp=tmp+carry can at maximum yield 8+1=9 with no carry from that operation.
Assuming that a+b does not generate a carry but gets a maximum value of 9. If the carry from the previous operations was set then you will have tmp=9+carry=9+1=~~1~~0 since the result 0 is less than either 9 or 1 then there is a carry detected that can be propagated forward.

Algorithm for carry detected in addition

```
BASE=2^64
carry=0
for(i=0..max(n,m))
        c[i]=a[i]+carry
        if(c[i]<a[i]) carry=1 else carry=0      // Set or reset carry
        c[i]=c[i]+b[i]
        if(c[i]<b[i]) carry=1                   // Set carry or keep carry
if(carry !=0)
        c[max(n,m)+1]=carry
```

Algorithm for carry detection in subtraction

```
borrow=0
for(i=0..max(n,m))
        c[i]=a[i]-(b[i]+borrow)
        if(a[i]<b[i]+borrow) borrow=1       // Set borrow
        else if([i]!=0) borrow=0            // Reset borrow or keep borrow
        if(borrow!=0)
        //result underflow
else
        // Result OK
```

With this form of carry detection, we can now utilize the full amount of memory regardless of any bit-size environment (32-bit or 64-bit).

Useful functions for integer arithmetic

Several useful functions are typically presented in arbitrary precision packages. These are:

- Integer power x^y
- Integer power x^y % z
- gcd(a,b) // Greatest Common Divisor
- lcm(a,b) // Least Common Multiple

Integer power x^y:

To calculate x^y where both x and y are integers we of course do not multiply x with x, y times. Instead, we use the entity when y is an even number:

$$x^y = x^{\frac{y}{2}+\frac{y}{2}} = x^{\frac{y}{2}}x^{\frac{y}{2}} = (x \cdot x)^{\frac{y}{2}} \quad (2)$$

Algorithm for ipow(x,y)

```
ipow(x,y)
        r=1
        while(y>0)
                if(y is odd)
                        r=r*x
                x=x*x
                y=y/2
        return r
```

Integer power x^y modulus z

Where x, y, and z are all integers. Instead of first calculating x^y and then taking the modulus z, which can lead to a very high number of digits for the interim result, and then carrying out the modulus z to get the answer. E.g. $2^{1000000}$ is around a number with over 300,000 digits and then taking the modulus of z e.g. 77 can be very time-consuming since we first have to build a digit with over 300,000 digits and then apply the modulus operator that is a very costly operation (see discussion under division and remaining).
To avoid large numbers we can incorporate the modulus operator into our calculation of x^y to avoid dealing with a high number of digits in the interim result and we get the following algorithm:

Algorithm for ipow_mod(x,y,z)

```
ipow_mod(x,y,z)
        r=1
        x=x%z
        while(y>0)
                if(y is odd)
                        r=r*x
                        r=r%z
                x=x*x
                x=x%z
                y=y/2
        return r
```

Greatest Common Divisor (GCD)

The Greatest Common Divisor (GCD) is the largest positive integer that divides both a and b. Very commonly used and is part of any arbitrary precision packages.

Algorithm: gcd(a,b)

```
gcd(a,b)
        while(b>0)
                tmp=b
                b=a%b
                a=tmp
        return a
```

The above version has the deficit that uses the % operator which is notorious time-consuming in arbitrary precision. Instead, it is better to use the "binary" version that only required subtraction and shifting which are considered fast operations in arbitrary precision.

Algorithm binary gcd_binary(a,b)

```
gcd_binary(a,b)
        tmp = a | b
        shift = ctz(tmp)        // ctz returns the number of least significant zero bits
        a >>=ctz(a)             // ctz returns the number of least significant zero bits
        do
                b >>= b.ctz();
                if (a > b)
                        tmp = b
                        b = a
                        a = tmp
                b -= a;
        while (b != 0);
        return a << shift;
```

Least Common Multiple (LCM)

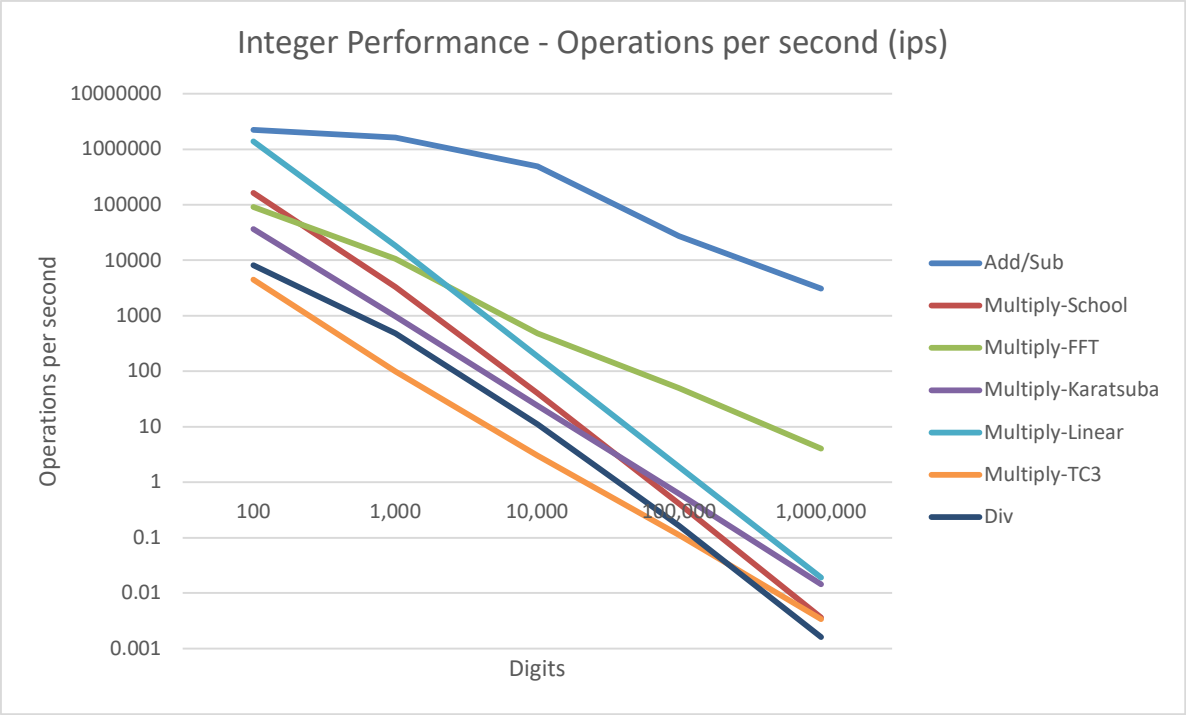
The Least Common Multiple is the smallest positive integer that is divisible by both arguments and it will internally also use the gcd() algorithm.

Algorithm lcm(a,b)

```
lcm(a,b)
        gcd_ab = gcd(a, b);
        a /= gcd_ab;
        a*= b;
        return a;
```

Performance of Arbitrary Integer precision:

Below show the performance of the integer precision. Y-axis is a logarithm scale of the number of operations per second (Ops). X-axis is the number of digits. For add/subtract/multiplication both operands is of the same number of digits. For the division, the denominator has half as many digits as the dividend.



Integer Operation per second vs. number of digits

It is not a surprise that addition and subtraction are the faster operations.
For multiplication, the use of multiplication via linear convolution is the fastest multiplication for smaller numbers up to approx. 5,000-6,000 digits where after multiplication using FFT takes over. This is expected. It takes some initial code to set up an FFT multiplication and that is why it first takes over around the 5,000-6,000 digits mark. Division and remainder as expected is the most time-consuming task and are many times slower than multiplication. The lesson learned is that you should try to avoid division as much as possible. This knowledge comes into play when using the Taylor series for exponential, logarithm, Trigonometric, and Hyperbolic functions where we will use a technic called coefficient scaling to lower the number of divisions in the Taylor series.

Below is a table where we set the addition/subtraction to 1 and the others are scaled after that.

Performance ratio	Digits				
	100	**1,000**	**10,000**	**100,000**	**1,000,000**
Addition/Subtraction	1	1	1	1	1
Multiplication-School	14	500	12,198	67,167	851,389
Multiplication-FFT	24	153	1,021	560	766
Multiplication-Karatsuba	61	1,670	20,330	44,727	211,379
Multiplication-Linear	2	89	2,623	14,452	161,316
Multiplication-TC3	500	16,445	162,641	246,703	901,471
Division	274	3,392	44,357	168,454	1,915,625

We notice that schoolbook multiplication is of no use in arbitrary precision arithmetic. FFT is the way to go above 5,000-6,000 digits. Below that multiplication, using linear convolution is the fastest. An implementation should automatically determine which multiplication method to use based on the size of the integer in digits. Both the Karatsuba and Toom-cook3 performed slower than the FFT algorithm.

The surprise is division. We expected it to be slower but it is many times slower than any of the other operations. Particularly when you scale the number of digits. I recommend that for integer division (unless it is simple) you consider switching to a floating-point division since it scales many magnitudes better than the integer division for arbitrary precision (see later).

Floating-point arithmetic:

In arbitrary precision, a floating-point number can be described by the following components:

- The sign
- The integer part of the number
- The fraction part of the number
- The exponent of the number
- The precision (since that is dynamic in arbitrary precision)
- The rounding mode (optional if there is a need for various rounding modes)

You can use a decimal representation where the integer and fraction part is stored as decimal strings (usually in base 10) or as binary numbers (base 2 as the native CPU supports). Furthermore, we store the *float_precision* number as normalized binary numbers in the format:

$$Floating\ point\ number\colon\ sign\ \cdot i_1 \cdot f_n \cdot \beta^{e_p}$$

Where the sign is either + 1: -1. For the special case where the floating-point number is zero, we force the sign to be +1 meaning that -0 is not a valid number.
Where i_1 indicates that there is only one digit as the integer part.
Where f_n indicates that there are n digits in the fraction part.
Where e_p indicated that the exponent contains p digits.
In addition, β is the base typically base2 for a binary implementation.

You could have other arrangements for your internal floating-point presentation, however, this is the one chosen for the author's arbitrary precision packages.

Furthermore, for efficiency, you do not store the integer part and the fraction part in separate internal variables (although you could). Since a normalized number always has one digit (in base 2 or the number 1-9 in base 10) as the integer part, we can store the entire number in one variable.
In our actual implementation of the arbitrary precision packages, we use a vector of binary digits for better storage utilization where the base=2^{64} in a 64-bit environment and base=2^{32} in a 32-bit environment.
Lastly, the exponent is stored as a single 64-bit signed number, which should be more than adequate to hold any exponent for an arbitrary precision number.

For base 10 you get:

$$Floating\ point\ number\colon\ sign\ \cdot i_1 \cdot f_n \cdot 10^{e_p}$$

Example: 1.234E2
I_1=1
F_3=234
E_p=2

The normalized number for float_precision

A *float_precision* variable is always stored as a normalized number with a one in the integer portion of the number (for binary implementation and 1-9 in decimal implementation). The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros in the fraction part. [1].

If i_1 is outside this range the number is un-normalized. With any of the arithmetic operations, the intermediate result can be an un-normalized number. However, our arbitrary precision packages always guaranteed that the result of any arithmetic operations would always be returned as a normalized number.

Mixed Precision for float_precision

Since our floating-point number can be of different precision and we of course allow mixed precision for our arithmetic operators, it is important to understand how the precision works. In any assignment statement (C or C++ language) =, +=, -=, *=, /=,%= the resulting precision is always the precision of the variable on the left-hand side of the operator and if necessary the expression on the right-hand side is rounded accordingly. For binary operator like +,-,*,/,%. The mixed precision is handled by always aligning the argument on both sides of the operator to the argument that has the highest precision. E.g. in an expression of $a+b$ where a is a 3-digit precision number and b is a 5-digit precision number, the operations $a+b$ are carried out using 5-digit precision.

Rounding control for float_precision

Rounding control: The default is, of course, rounding to the nearest but the arbitrary precision packages also allow you to control the rounding process by rounding towards zero, rounding up, and rounding down in the same way as implemented in a microprocessor. Controlling the rounding makes it very easy to implement interval arithmetic (which is also part of this arbitrary precision package).

Addition:

When adding two floating-point numbers a_n and b_m. The fraction part can have different precision and the exponent part can be different as well. To do addition we first align the two numbers exponent to the same exponent. This is done by aligning the number with the lowest exponent to the highest exponent by adding leading zeros to the number with the lowest exponent.
E.g. a_n=1.2345E5 and b_m=6.78E1
We align b_m to the same exponent of a_n by adding leading zeros to the number:
a_n=1.2345E5 b_m=0.000678E5
The next issue is that the two numbers can be of different precision, this is no different than in the standard C programming language you can have a floating number *float* type which is 32-bit precision, and a *double* type which is 64-bit precision. The rule for mixed floating point arithmetic dictates that when two numbers are of different precision the number with the lowest precision is first converted to the same precision as the number with the highest precision and then the operation is performed. E.g. using our two numbers where a_n is a 5-digit precision and b_m is now a 7-digit precision number we then align it to the 7-digit precision. a_n=1.234500E5 and b_m=0.000678E5

Now we can add the two numbers together:

$$\begin{aligned} & 1.234500E5 \\ + & \ 0.000678E5 \\ = & \ 1.235178E5 \end{aligned}$$

The addition is performed in the same way as for integer arithmetic. After the addition, we can then round the result back to the precision of *a* (5 digit) and we get 1.2352E5.
Now sometimes the intermediate result can be an un-normalized number e.g. adding two 2-digit precision numbers 9.5E0+2.4E0=11.9E0 but we then normalized and rounded the number to 2-digit precision: 1.2E1.

If either a_n or b_m is negative we resolve the sign using the below table as we did for integer arithmetic.

-	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n-b_m$	$C=a_{n+}\|b_m\|$
$a_n < 0$	$C=-(\|a_n\|+b_m)$	$C=-(\|a_n\|+\|b_m\|)$

Here is an example of how the process is working. The number a_n is a four-digit precision number (1.235E+3) and b_m is a 2 digits precision number (-2.4E0) in the operation a+=b.

Step 1: Extract the sign, number, and exponent from the numbers.
Step 2: Align mantissa to max exponents.
Step 3: Align *a;* to the current temporary precision, which is five due to the alignment of the number of *b.*
Step 4: Perform the addition of *a* and *b*.
Step 5: Round the result to four digits precision (a's precision).
Step 6: Reapply the exponent from a and the result is 1.233E+3.

	A	+	B	=	C
Number	1.235E+03		-2.4E+00		
Precision	4		2		
Sign	+		-		
Mantissa	1.235		2.4		
Exponent	3		0		
Align to max exponents	1.235		0.0024		
Align precision	1.2350		0.0024		
Add the two numbers					1.2326
Round to precision					1.233
Reapply exponent					1.233E+3

*) Addition is done using the same code as for integer arithmetic.

Subtraction:

Is done using the addition function since *a-b* is the same as *a+(-b).* We simply just change the sign on *b* and then call the addition function.
Using the same example as under the addition section we get:

Subtraction	A	-	B	=	C
Number	1234.56		-2.4		
Precision	4		2		
Sign	1		-1		
Mantissa	1.235		2.4		
Exponent	3		0		
Align to max exponents	1.235		0.0024		
Align precision	1.2350		0.0024		
Subtract the two numbers					1.2374
Round to precision					1.237
Reapply exponent					1.237E+3

*) the subtraction of the two numbers is done using the same code as for integer subtraction

Multiplication:

As for multiplication, we have a choice of different multiplication methods. (Same choice as mentioned under integer arithmetic.) We have previously found that the linear convolution is the fastest for a smaller number of precisions (up to 5,000-6,000 digits) where the FFT multiplication takes over. See the description of integer multiplications for details.
Before we call the FTT function, we strip off the sign and exponent and then use the resulting sign as follows

.

*	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C=a_n * b_m$	$C=-(a_n * \lvert b_m \rvert)$
$a_n < 0$	$C=-(\lvert a_n \rvert * b_m)$	$C=\lvert a_n \rvert * \lvert b_m \rvert$

Moreover, for the exponent we simply add them together since:

$$a10^{e1} \cdot b10^{e2} = (ab)10^{e1+e2}$$

Now since we compute the FFT using IEE754 arithmetic and we know that for a 64-bit floating point, we have 53 bits in the floating point mantissa we can then derive a bound for how large a number we can multiply using only 64-bit FFT transformation. This is the same bound as the outline under integer multiplication.

Division:

To handle floating-point division we rewrite the equation a/b to a(1/b). Multiplication is a much faster operation than division so it makes sense to do it this way. Now we only need to figure out how to quickly calculate the inverse of b=(1/b). This same issue faces many microprocessors or early RISC (Reduced Instruction Set CPU) that did not have hardware support for the division operator. Instead, they use a Newton iteration using the following algorithm for calculating 1/b: However, there exist other higher-order methods that we will examine in this chapter and are detailed in [8].

Newton's method for inverse

We can use a classic Newton iteration using the following algorithm for calculating 1/b:

$$x_{n+1} = x_n(2 - x_n y) \quad (3)$$

$$Where\ y = b\ and\ x_0 \approx \frac{1}{b}\ (initial\ guess)$$

$$and\ x_n\ converged\ towards\ \frac{1}{b}$$

Algorithm 1

Traditional this method has been used due to its simplicity

This can also be found the following way by restating the problem of finding $\frac{1}{x} = y$.

Applying it to the Newton method, you get:

Where $f(x) = y - \frac{1}{x},\ \ f'(x) = \frac{1}{x^2}$

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n}}{\frac{1}{x_n^2}} =>$$

$$x_{n+1} = x_n - x_n^2\left(y - \frac{1}{x_n}\right) =>$$

$$x_{n+1} = x_n - x_n(x_n y - 1) =>$$

$$x_{n+1} = x_n(2 - x_n y) \quad (4)$$

Notice the algorithm only requires us to do one subtraction and two multiplications per iteration.

Example of Newton for inverse

To see how this algorithm works let us find the inverse of 1.6 using an initial start guess of 0.1.

y=	1.6
x_0=	0.1

n	x	Error
1	0.184	4.4E-01
2	0.31383	3.1E-01
3	0.470078	1.5E-01
4	0.586598	3.8E-02
5	0.622641	2.4E-03
6	0.624991	8.9E-06
7	0.625	1.3E-10
8	0.625	0.0E+00

After eight iterations, the difference between the iteration and the build-in division operator is zero and the result of 1/1.6 is 0.625.

Now the only question that remains is how to find a suitable starting point for the iteration since we cannot perform an initial division as the guess of 1/b. Instead, we look at how our arbitrary precision number is built up. i_1 is the one-digit integer and f_n is the n fraction parts digits, e_p is the exponent power in base 2.

$$\frac{1}{b} = \frac{1}{i_1 \cdot f_n 2^{e_p}} = \frac{1}{i_1 \cdot f_n} 2^{-e_p} \quad (5)$$

We can extract the exponent portion and find the inverse $\frac{1}{i_1 . f_n}$ and then multiply the result with 2^{-e_p} to find our inverse of 1/b. Extracting the exponent will leave us with a number [1..2[. Since we do have the support of hardware division using the IEE754 standard (a 64-bit floating-point number) we can get our initial start guess with approximately 15-16 digits accuracy and then begin to iterate towards a higher number of accuracy. In case you do not have access to IEEE754, you can do a lookup table to find a suitable starting point.

The Newton method for division is very fast and has quadratic convergence meaning that for each iteration we will double the number of correct digits. To set this into perspective, assume we have a number with 128 digits (2^7) and we start with approx. 2^4 correct digits then we should expect only three iterations to get our result. For 1,000 digits it will require approx. six iterations and for 1,000,000 digit precision approx. sixteen iterations.

Cubic convergence method for inverse

A higher-order Newton-like method exists with a cubic convergence rate. We can iterate toward the inverse by using the following:

$$x_{n+1} = x_n + x_n(1 - yx_n) + x_n(1 - yx_n)^2 \quad (6)$$

We notice that compared to the Newton method we have an extra addition of $x_n(1 - yx_n)^2$ which adds one extra addition and one extra multiplication.

Alternatively, the iteration can be written as:

$$z_n = 1 - yx_n$$
$$x_{n+1} = x_n + x_n(z_n) + x_n(z_n)^2 \quad (7)$$

Algorithm 2

It can be found using Householders 2nd order method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f(x_n)^2 f''(x_n)}{2f'(x_n)^3} \quad (8)$$

Where $f(x) = y - \frac{1}{x}$, $f'(x) = \frac{1}{x^2}$, $f''(x) = -\frac{2}{x^3}$

This yield:

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n}}{\frac{1}{x_n^2}} - \frac{\left(y - \frac{1}{x_n}\right)^2 (-\frac{2}{x_n^3})}{2\left(\frac{1}{x_n^2}\right)^3} =>$$

$$x_{n+1} = x_n + x_n^2\left(\frac{1}{x_n} - y\right) + x_n^3(\frac{1}{x_n} - y)^2 =>$$

$$x_{n+1} = x_n + x_n(1 - yx_n) + x_n(1 - yx_n)^2 \quad (9)$$

This method will require one subtraction, two addition, and four multiplication.

We could be tempted to factor out the x_{n-1} as outlined below:

$$z_n = 1 - yx_n$$
$$x_{n+1} = x_n(1 + z_n + (z_n)^2) \quad (10)$$

This will require three addition/subtraction and three multiplication. However, all the multiplication needs to carry out using full precision.

The cubic convergence rate means that for each iteration you triple the number of correct digits requiring fewer iterations than the Newton method.

Example of Cubic method for inverse

To see how this algorithm works let us find the inverse of 1.6 using an initial start guess of 0.1.

y=	1.6	
x_0=	0.1	
n	**x**	**Error**
1	0.25456	3.7E-01
2	0.494865	1.3E-01
3	0.619358	5.6E-03
4	0.625	4.6E-07
5	0.625	0.0E+00

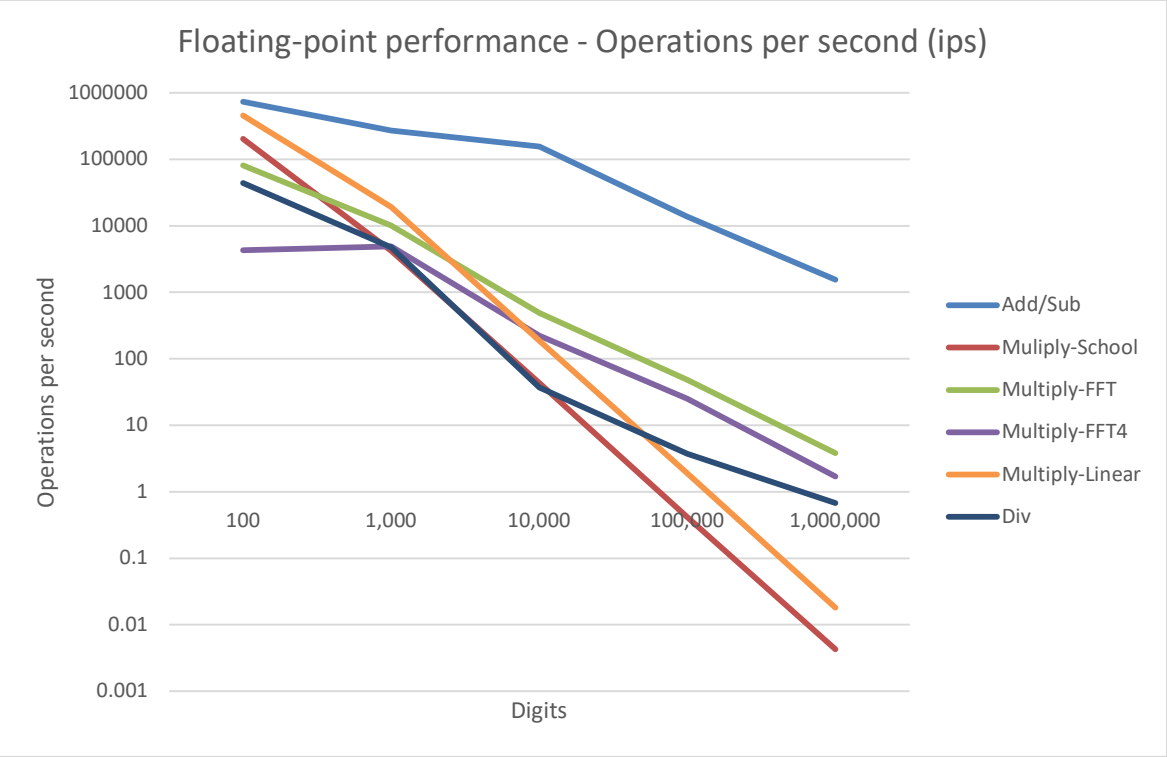
Which method for the inverse?

Both Newton (second order) and the cubic (third order method) have advantages and disadvantages. If you begin to measure the performance, you will notice that sometimes the Newton method is faster, and sometimes the cubic method is faster. It all boils down to how many iterations you need. Now a third-order method requires 1.58 iterations less than a second-order method. However, you cannot do a fraction of iterations since it has to be an integral number. To get the best of both worlds we have chosen a hybrid implementation where we pre-calculate the number of expected iterations for each method and then round it up to the nearest higher integer number. Divide the number of Newton iterations by the

number of cubic iterations. If the division is > 1.58 then we choose Cubic iterations. If less or equal (<=1.58) we choose the Newton method.

Performance of Arbitrary Floating point precision:

Below show the performance of the floating-point precision. Y-axis is a logarithm scale of the number of operations per second (Ops). X-axis is the number of digits. For add/subtract/multiplication both operands is of the same number of digits. For the division, the denominator has half as many digits as the dividend.



Floating point Operations per second versus No of Digits
Not surprisingly the performance ratio between addition/multiplication and division increases with a higher number of digits in arithmetic operations.

Performance ratio	Digits				
	100	**1,000**	**10,000**	**100,000**	**1,000,000**
Addition/Subtraction	1	1	1	1	1
Multiplication-School	4	65	3,653	33,345	359,302
Multiplication-FFT	9	27	324	289	404
Multiplication-FFT4	170	55	704	555	914
Multiplication-Linear	2	14	845	7,375	85,833
Division	17	58	4,246	3,727	2,272

We notice that old fashion school book multiplication doesn't scale well with a higher number of digits and it is not useful for arbitrary precision arithmetic. Multiplication using FFT is the way to go beyond 5,000-6,000 digits. Below 5,000 digits multiplication using

linear convolution is the fastest choice. The FFT-4 is performance measured with only a 4-bit sample size instead of the usual 8-bit sample size for FFT multiplication. 4-bit is needed beyond 116M digits and due to the lower sample size, your performance drops by a little more than a half.
The big surprise is division. It is still much slower than multiplication but significantly faster than the division algorithm for integer arithmetic. Therefore we recommend when needing to perform integer division that the integer is converted to a *float_precision* variable, then divide using the *float_precision* division, and then convert back to an integer precision variable. (conversion back and forth between an *int_precision* variable and a *float_precision* variable is very fast). In general, the performance graph shows that it is wise to avoid the division at nearly all costs.

Needed extra functionality

In the previous section, we establish the four basic arithmetic operations: Addition, Subtraction, Multiplication, and Division. These are the basic blocks for all arbitrary precision arithmetic. The basic block is used to implement other mandatory functions for arbitrary precision math packages. These functions are:

- Square roots
- Elementary functions:
  - Exponential function.
  - Logarithms functions
  - Power functions x^y
  - Universal constants
    - e, π, ln2, ln10
- Trigonometric functions
  - Sine, Cosine, Tangent, Arcsine, Arccosine, Arctangent
- Hyperbolic functions
  - Sinh, Cosinh, Tanh, ArcSinh, ArcCosh, ArcTanh
- Special functions
  - Gamma, Beta, Zeta, Error, Lambert W functions, and others
- Special constants
  - Euler-Mascheroni, Catalan, and Apéry Zeta(3)

Square root:

There exist several methods to compute the square root. Among them are:

1) Newton's Method.
2) Halley's Method.

The most common one for arbitrary precision libraries is the Newton method. A more detailed description can be found in [8] together with source code, performance charts, etc. Also [4] is a good reference for square roots calculation.

Newton's Method for square root

For the function sqrt(y) we can use a Newton iteration algorithm to get our result. The Newton iteration is defined by:

$$x_{n+1} = \mathrm{x}_n - \frac{f(x_n)}{f'(x_n)} \tag{11}$$

This method can be found the following way by restating the problem of finding Sqrt(y), we instead try to find the reciprocal square root of y which is: $\frac{1}{\sqrt{y}}$. Once it has been found we can find $\sqrt{y} = y\frac{1}{\sqrt{y}}$. By just multiplying the result with y.

Now to find the$\frac{1}{\sqrt{y}}$. We use the equation $\frac{1}{x^2} = y \Rightarrow \frac{1}{x^2} - y = 0$.

Using Newton's formula, we get using f(x)= $\frac{1}{x^2} - y$, f'(x)=$\frac{-2}{x^3}$

$$x_{n+1} = \mathrm{x}_n - \frac{\frac{1}{x_n^2} - y}{\frac{-2}{x_n^3}} \Rightarrow$$

$$x_{n+1} = \mathrm{x}_n + \frac{1}{2}x_n^3\left(\frac{1}{x_n^2} - y\right) \Rightarrow$$

$$x_{n+1} = \tfrac{1}{2}\mathrm{x}_n(3 - yx_n^2) \tag{12}$$

We now have our algorithm for finding the square root without any division.

$$x_{n+1} = \tfrac{1}{2}\mathrm{x}_n(3 - yx_n^2) \tag{13}$$

$$Where\ \ x_0 \approx \frac{1}{\sqrt{y}}\ (initial\ guess)$$

$$and\ x_n\ converged\ towards\ \frac{1}{\sqrt{y}}$$

Algorithm 3

For the initial guess x_0 we simply use the c library sqrt(b) function for the double variable. Now for this to work for arbitrary precision we need to use a little trick to ensure that we can

call the c library sqrt function with a double argument that fits the range of the IEEE754 double standard. See the initial guess section below.

Notice the algorithm only requires us to do one subtraction and four multiplications per iteration. Well, multiplication by 0.5 can be done by just adjusting the exponent and therefore should not count as a 'real' multiplication. We end up with one subtraction and three multiplication per iteration and then a final multiplication for the calculation of the square root.
Also as for the Newton method, we will have quadratic convergence meaning that for each iteration we will double the number of correct digits in our result.

The Initial guess

As for the initial guess, we can extract the exponent 2^{e_p} out of the equation, then multiply the result with $2^{\frac{e_p}{2}}$ after the iteration (assuming e_p is an even integer) and remember our exponent is an integer in base two. I_1 is the one-digit integer and f_n is the n fraction parts digits.

$$\frac{1}{Sqrt(y)} = \frac{1}{Sqrt(i_1.f_n 2^{e_p})} = \frac{1}{Sqrt(i_1.f_n)} 2^{-\frac{e_p}{2}} \quad (14)$$

If ep is odd, we have to use (since the exponent needs to be an integer):

$$\frac{1}{Sqrt(y)} = \frac{1}{Sqrt(i_1.f_n 2^{e_p})} = \frac{1}{Sqrt(i_1.f_n * 2)} 2^{-\frac{e_p - 1}{2}} \quad (15)$$

This simplifies the initial guess since we know that factoring out the exponent will leave us with an arbitrary precision number between [1..2[(for even exponent) and [1..4[for odd exponent. With the number well within the range of IEEE754, we can find a good initial guess of $\frac{1}{Sqrt(y)}$ using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

Example of Newton's method for square root

To see how this algorithm works let us find the Sqrt of 1.6 using an initial start guess of 1/1.6=0.625.

Newton 1/sqrt(y)			
Sqrt(y)	1.6		
y=	1.6		
x_0=	0.625		
n	**x**	**Sqrt(y)**	**Error**
1	0.7421875	1.1875	7.74E-02
2	0.786218643	1.257949829	6.96E-03
3	0.790533565	1.264853704	5.74E-05
4	0.790569413	1.26491106	3.90E-09
5	0.790569415	1.264911064	0.00E+00

After 5 iterations the difference between the iteration and the build in Sqrt() operator is 0 and the result of Sqrt(1.6) is 1.264911064

Brent's improvement

Brent [7] point out that you can improve the Newton algorithm by iterating using:

$$x_{n+1} = x_n + x_n(1 - yx_n^2) \quad (16)$$

Algorithm 4

Which is identical from a mathematical point of view but different from a computational point of view. Brent points out that you can perform the multiplication between x_{n-1} and $(1 - yx_n^2)$ in $x_n(1 - yx_n^2)$ using only half the precision in the multiplication. You gain one addition but do not need the multiplication with full precision. From a computational point of view, you do save some time or gain some performance using this formula for the iteration, particularly for a higher number of digits.

Halley's method for square root

Halley's method has a cubic convergence rate compared to Newton's quadratic order. Cubic convergence rate means that for every iteration you get 3 times as many correct digits compare to the Newton method which only gives you 2 times as many correct digits. Higher order convergence results in fewer iterations step at the expense of a more complex calculation per iteration. Normally it tends to even out that the time you save in fewer iterations steps is lost by a more complex iteration.

Halley square root method is using the following iterations step for finding $\frac{1}{\sqrt{y}}$:

$$z_n = yx_n^2$$
$$x_{n+1} = x_n \frac{1}{8}(15 - z_n(10 - 3z_n)) \quad (17)$$

Algorithm 5

And then we get the final result:

$$\sqrt{y} = yx_{n+1} \quad (18)$$

It can be found using Householders 2nd order method aka. Halley's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f(x_n)^2 f''(x_n)}{2f'(x_n)^3} \quad (19)$$

Where $f(x) = y - \frac{1}{x^2}$, $f'(x) = \frac{2}{x^3}$, $f''(x) = -\frac{6}{x^4}$

This yield:

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n^2}}{\frac{2}{x_n^3}} - \frac{\left(y - \frac{1}{x_n^2}\right)^2 \left(-\frac{6}{x_n^4}\right)}{2\left(\frac{2}{x_n^3}\right)^3} =>$$

$$x_{n+1} = x_n - x_n^3 \frac{1}{2}\left(y - \frac{1}{x_n^2}\right) + x_n^5 \frac{3}{8}(y - \frac{1}{x_n^2})^2 =>$$

$$x_{n+1} = x_n - x_n\frac{1}{2}(yx_n^2 - 1) + x_n\frac{3}{8}(yx_n^2 - 1)^2 =>$$

$$Substitute\ z_n = yx_n^2\ you\ get\ \ x_{n+1} = x_n - x_n\frac{1}{2}(z_n - 1) + x_n\frac{3}{8}(z_n - 1)^2 =>$$

$$x_{n+1} = x_n\frac{1}{8}(8 - 4(z_n - 1) + 3(z_n - 1)^2) =>$$

$$x_{n+1} = x_n\frac{1}{8}(15 - 4z_n + 3(z_n^2 + 1 - 2z_n)) =>$$

$$x_{n+1} = x_n\frac{1}{8}(15 - 10z_n + 3z_n^2) =>$$

$$x_{n+1} = x_n\tfrac{1}{8}(15 - z_n(10 - 3z_z)) \qquad (20)$$

Per iteration, we have five multiplication and two subtraction. Compare to Newton we have additional subtraction and two extra multiplication so each iteration will take a little bit longer however you will have fewer iterations to perform. (Approx. 2/3).

Example of Halley's method for square root

Halley 1/Sqrt(y)			
Sqrt(y)	1.6		
y=	1.6		
x_0=	0.625		
n	**x**	**Sqrt(y)**	**Error**
1	0.775146	1.240234375	2.47E-02
2	0.790555	1.264887927	2.31E-05
3	0.790569	1.264911064	1.93E-14
4	0.790569	1.264911064	0.00E+00

As expected, we get a faster iteration and reach the result after only four iterations.

Which method for the square root?

Both Newton (second order) and the Halley (third order method) have advantages and disadvantages. If you begin to measure the performance, you will notice that sometimes the Newton method is faster, and sometimes the Halley method is faster. It all boils down to how many iterations you need. Now the third-order Halley method requires 1.58 iterations less than a second-order Newton method. However, you cannot do a fraction of iterations since it has to be an integral number. We have chosen a hybrid implementation where we pre-calculate the number of iterations for each method and then round it up to the nearest higher integer number. Divide the number of Newton iterations by the number of Halley iterations. If the division is > 1.58 then we choose Halley iterations. If less or equal (<=1.58) we choose the Newton method.

N'th root

Now that we have found a better way of doing the square root we also need to consider if we can use a similar technic when dealing with the $\sqrt[n]{x}$. By default, we resort to the power function which evaluates to:

$$\sqrt[n]{x} = x^{\frac{1}{n}} = e^{\frac{1}{n} * log_e(x)} \quad (21)$$

Which use two very expensive and time-consuming functions exp(x) and log(x). Instead, we can create a faster way to calculate $\sqrt[n]{x}$. Using the same principle as the sqrt(). The result is a huge speed-up improvement.

As can be seen below the speed of the nroot() is more or less constant regardless of the n[th] root and it is several magnitudes better than the traditional calculation via the pow() function.

Let us end the discussion of the sqrt() and nroot() by devising the Newton formula for the nroot. It is quite similar to the way we got the algorithm for the sqrt() function. We are trying to find a function to the solution $x = \sqrt[n]{S} => \ x^n = S => \ \frac{1}{x^n} = \frac{1}{S}$

Letting $y = \frac{1}{S}$ you get: $f(x) = \frac{1}{x^n} - y = 0 \ and \ f'(x) = -nx^{-n-1}$

Using the Newton method, you get:

$$x_{i+1} = x_i - \frac{x_i^{-n} - y}{-nx_i^{-n-1}} =>$$

$$x_{i+1} = x_i + \frac{1}{n}(x_i - x_i^{n+1}y) =>$$

$$x_{i+1} = x_i + \frac{1}{n}x_i(1 - x_i^n y) =>$$

$$x_{i+1} = x_i \frac{1}{n}(n + 1 - x_i^n y) \quad (22)$$

And now:

$$\sqrt[n]{S} = \frac{1}{x_{i+1}} \quad (23)$$

We still have a division $\frac{1}{n}$ but it is with the constant *n* so we can calculate it once before the start of the iteration avoiding any division while iterating.

We could have done a more direct approach as we saw for the square root:

$$x = \sqrt[n]{S} => \ x^n = S => x^n - S = 0 \quad (24)$$

You get $f(x) = x^n - S = 0 \ and \ f'(x) = nx^{n-1}$

Using the Newton method, you get:

$$x_{i+1} = x_i - \frac{x_i^n - S}{nx_i^{n-1}} =>$$

$$x_{i+1} = x_i - \frac{1}{n}(x_i - \frac{S}{x_i^{n-1}}) =>$$

$$x_{i+1} = \frac{1}{n}((n-1)x_i + \frac{S}{x_i^{n-1}}) \quad (25)$$

We end up with an extra division that we need to calculate per iteration and therefore it will be slower than the first version as we saw when calculating the square root.

There exist other higher-order methods like the Halley but that will be slower than the Newton version. In the Booth Arbitrary precision library, they did some testing and the Newton method came out ahead of all other methods. See [20]

As for the nrooth algorithm, it can also benefit from using dynamic precision as outlined in [8] for both the inverse and sqrt root functions

Elementary functions:

For all elementary functions regardless of whether it is logarithmic, exponential, trigonometric, or hyperbolic functions, we use either a Taylor series or some iteration method like the Newton's to find our results.

For all these methods, we do the same. Before we apply the Taylor series or Newton iterations, we first reduced the argument to improve the efficiency of our methods or reduce the problem to a domain where it is faster to evaluate the function. We reduced the argument x to x_1 and then evaluate the function using x_1 at $f(x_1)$ and then finally restore the original f(x) using the result of $f(x_1)$.

We also rely on another trick to increase performance by using coefficient scaling of a group of Taylor terms to reduce the need to perform division for every Taylor term.

For argument reduction, we either use Additive/Subtractive argument reduction where $x_1=x-k$ *for some value of k.* This is particularly useful for function with a periodic nature like Sine and Cosine functions that has a period of 2π.
Another way is to use a Multiplicative argument reduction where $x_1=x/k$. e.g. the Exponential double formulae: $\exp(2x) = \exp(x)^2$ where $k=2$. Here the original argument is reduced by a factor of two but then we have to square the result after our calculation to restore the original calculation. Needless to say that you can repeatedly apply the reduction formulae to your original problem

Exponential functions

There are a couple of ways you can calculate exp(x) in arbitrary precision. Traditional a Taylor series expansion has been used but some have suggested the use of the sinh() function to calculate the exp(). This chapter will examine:

1) Exp(x) using Taylor series.
2) Exp(x) using Sine Hyperbolic function.
3) Exp(x) using the Binary Splitting method

The most common one for arbitrary precision libraries is the standard Taylor series expansion method. For other methods and more details, see [9].

e^x using the Taylor series

For the function, exp(x) we can use the corresponding Taylor series for exp(x) as defined by:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \qquad (26)$$

We eliminate x< 0 by using the identity: $e^{-x} = \frac{1}{e^x}$ meaning we first calculate e^x and then do the inverse of $\frac{1}{e^x}$.

Unfortunately, this series does not converge very fast and will require many Taylor terms to complete.

Example 1 of Taylor series for e^x

Using x=1 we get after 17 Taylor series the result of exp(1)= 2.718281828459

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**1**	
Argument reductions=		**0**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.00E+00	1.000000000000	1.000000000000	1.72E+00
2	1.00E+00	2.000000000000	2.000000000000	7.18E-01
3	5.00E-01	2.500000000000	2.500000000000	2.18E-01
4	1.67E-01	2.666666666667	2.666666666667	5.16E-02
5	4.17E-02	2.708333333333	2.708333333333	9.95E-03
6	8.33E-03	2.716666666667	2.716666666667	1.62E-03
7	1.39E-03	2.718055555556	2.718055555556	2.26E-04
8	1.98E-04	2.718253968254	2.718253968254	2.79E-05
9	2.48E-05	2.718278769841	2.718278769841	3.06E-06
10	2.76E-06	2.718281525573	2.718281525573	3.03E-07
11	2.76E-07	2.718281801146	2.718281801146	2.73E-08
12	2.51E-08	2.718281826198	2.718281826198	2.26E-09
13	2.09E-09	2.718281828286	2.718281828286	1.73E-10
14	1.61E-10	2.718281828447	2.718281828447	1.23E-11
15	1.15E-11	2.718281828458	2.718281828458	8.15E-13
16	7.65E-13	2.718281828459	2.718281828459	5.02E-14

17	4.78E-14	2.718281828459	2.718281828459	0.00E+00

That is not too bad, however, if we change the argument to 10 then we need 45 Taylor's terms to get the result and if we use x=0.1 then we only need 10 Taylor terms.
This lead to the observation that the number of Taylor's terms needed depends heavily on the argument to exp(x).

Argument Reduction for e^x

We prefer to have our x < 1 to ensure that the Taylor series converges more quickly. We can accomplish that using a technique called *argument reduction* to work with a smaller number to get a faster converging to e^x using fewer *terms* of the Taylor series.
We can use the identity: $e^x = (e^{\frac{x}{2}})^2$ to reduce the argument with a factor of two and then after the Taylor iterations we can square the result to find the correct value of e^x.
Or more generally we can reduce the argument x for some k where:

$$e^x = (e^{\frac{x}{2^k}})^{2^k} \tag{27}$$

Iterate through the Taylor terms of the reduced argument $\frac{x}{2^k}$ and then Square the result k times after the Taylor iterations. This makes sense since for each Taylor term you need to divide with the factorial and that is many times more time-consuming than squaring the result k times after the Taylor iterations.

Example 2: Taylor series for e^x using argument reduction

If using the previous example 1 and reducing the argument twice from one to 0.25 we only need 12 Taylor terms to get the same result as before, saving five Taylor terms but gaining two squaring at the end. However, overall huge savings since we have avoided five time-consuming divisions in Taylor's terms.

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**0.25**	
Argument reductions=		**2**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.00E+00	1.000000000000	1.000000000000	2.84E-01
2	2.50E-01	1.250000000000	2.441406250000	3.40E-02
3	3.13E-02	1.281250000000	2.694855690002	2.78E-03
4	2.60E-03	1.283854166667	2.716831973351	1.71E-04
5	1.63E-04	1.284016927083	2.718209939201	8.49E-06
6	8.14E-06	1.284025065104	2.718278851251	3.52E-07
7	3.39E-07	1.284025404188	2.718281722614	1.25E-08
8	1.21E-08	1.284025416299	2.718281825163	3.89E-10
9	3.78E-10	1.284025416677	2.718281828368	1.08E-11
10	1.05E-11	1.284025416687	2.718281828457	2.69E-13
11	2.63E-13	1.284025416688	2.718281828459	5.77E-15
12	5.97E-15	1.284025416688	2.718281828459	0.00E+00

If we use an eight-times reduction we get the same results after just six Taylors terms.

Exp(x)		**Original**	**X Reduced**	

x=		1	0.00390625	
Argument reductions=		8		
Terms	Term value	Term Sum	Exp(x)	Error
1	1.00E+00	1.000000000000	1.000000000000	3.91E-03
2	3.91E-03	1.003906250000	2.712991624253	7.64E-06
3	7.63E-06	1.003913879395	2.718274935741	9.94E-09
4	9.93E-09	1.003913889329	2.718281821729	9.71E-12
5	9.70E-12	1.003913889338	2.718281828454	7.33E-15
6	7.58E-15	1.003913889338	2.718281828459	0.00E+00

These examples demonstrate the efficiency of using argument reduction.

The issue with arbitrary precision for e^x

17 Taylor's terms to reach a result do not seem so bad at a first glance. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. In Yacas book of algorithms [6] they found a bound for the number of Taylor terms n needed as a function of the number of precision in digits P assuming $|x|< 1$:

$$n = \frac{P \cdot \ln(10)}{\ln(P)} - 1 \qquad (28)$$

For P = 1,000 digits you get n=332 Taylor terms are needed. For 10,000 digits, n=2,499, and 100,000 digits you get a whopping n=19,999 Taylor terms and 1M digits, n=166,666 terms. With that amount of Taylor terms, it will take a long time to evaluate exp(x) for high numbers of digits, see table below.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Taylor terms	9	49	332	2,499	19,999	166,666	1.43M	12.5M	111M

Now to see the effect of argument reduction on improving the Taylor series we have recorded the amount of Taylor terms needed for various argument reductions from 1 to 128 on a random floating-point number between 1.xxx and 9.xxx. From the table, we see that the reduction in the number of Taylor terms varies more than 10-fold between 1 as the reduction factors to a reduction factor of 2^{128}

The Auto reduction is the number of Taylor terms when we automatically find a reasonable reduction factor. Most of the time it varies between 32 to 64 reductions.

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	5	12	75	516	4,393
1 Pred.	17	96	435	3,861	25,197
2 Red.	15	81	393	3,510	23,580
4 Red.	11	60	327	2,962	20,877
8 Red.	8	40	243	2,244	16,941
16 Red.	5	24	159	1,497	12,241
32 Red.	4	13	94	889	7,820
64 Red.	3	8	52	487	4,510
128 Red.	3	5	28	255	2,430

The total number of operations going from one Taylor term to the next is:

$$\frac{x^n}{n!} \rightarrow \frac{x \cdot x^n}{(n+1) \cdot n!} \quad (29)$$

Is two multiplication and one division. The n+1 can be handled using the native C++ types and does not count for the workload for arbitrary precision.

Now doing k reduction will require k multiplication before the Taylor iterations start and k multiplication at the back-end or 2k multiplication. The front operation multiplication for a normalized arbitrary precision number is not performed as a real multiplication (of 0.5) but handle by just subtracting one from the exponent (which is the same as dividing by two or multiply by 0.5). This does not amount to anything that counts towards the workload and can be ignored. On the back-end, it will still require k multiplication. As an example, we can calculate the total workload for a 10,000 digits number using one reduction versus two reductions.

1-reduction workload = 3,861*(2*multiplication+1 division)+1*multiplication=7,723*multiplication and 3,86*1division.

16-reduction workload: 1,497*(2*multiplication+1*division)+2*multiplication= 2996*multiplication and 1,497 division

Assuming division is 10 times slower than multiplication, you get a total workload of multiplication equivalence of 7,723+10*3,861=46,333 for 1 reduction and 17,966 or 40% reduction in workload.

Finding a reasonable reductions factor for e^x

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? Yacas book [6] states that at least x should be lower to $|x|<10^{-M}$ and M should be:

$$|x| < 10^{-M} \; and \; M > \frac{\ln(P)}{\ln(10)} \quad (30)$$

Where P is the precision in decimal digits.

Measuring the performance indicates that this is not the most optimal selection. Therefore, we multiply the M found above with a constant eight to get a more reasonable reduction factor and then adjust for the magnitude of |x| itself.

The adjustment for the magnitude of |x| is simply the number exponent (power of 2 exponent) to ensure that the number will be well below one. This works well for small magnitude |x| and for high magnitude |x|. By just adding the exponent (positive or negative) to the reduction factor.

The performance table below shows the effect of using increasingly higher reduction factors.

All measures are in milliseconds

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	0.11	0.53	17	5,596	291,871
1 Pred.	0.24	2.50	59	39,812	1,810,970
2 Red.	0.20	2.00	67	38,736	1,286,680
4 Red.	0.13	1.57	50	32,372	1,104,910
8 Red.	0.09	1.11	57	24,334	898,426
16 Red.	0.08	0.71	34	16,026	652,547
32 Red.	0.10	0.53	22	9,425	413,501
64 Red.	0.24	0.71	15	5,309	241,661
128 Red.	0.59	0.59	17	3,330	131,452

As you can see for higher precision, you will benefit even more from increasing the reduction factor.

Brent enhancement

To avoid loss of precision we do not do a repeated number of squaring at the back end. Instead of just squaring for every number of reductions performed.

$$e^x = (e^{\frac{x}{2}})^2 \tag{31}$$

We use the identity as suggested by Brent [6]:

$$e^x - 1 = \left(e^{\frac{x}{2}} - 1\right)\left(e^{\frac{x}{2}} + 1\right) =>$$
$$e^x - 1 = 2\left(e^{\frac{x}{2}} - 1\right) + \left(e^{\frac{x}{2}} - 1\right)^2 \tag{32}$$

Guard Digits for e^x

When summarizing a Taylor series as exp(x) you need quite a lot of summarizing and that will produce round-off errors. In Yacas [6] they estimate the round-off to be approx. per term involving one multiplication, one division, and one addition to be:

$$digits\ lost = \frac{3\ln(n)}{\ln(10)}\ where\ n\ is\ the\ number\ of\ Taylor\ terms \tag{33}$$

.

Lost digits as a function of Taylor terms

Taylor Terms	10	100	1,000	10,000	1,000,000
Lost digits.	3	6	9	12	15

Lost digits adjusted for actual Taylor's terms versus reduction factor

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	2.1	3.2	5.6	8.1	10.9
1 Pred.	3.7	5.9	7.9	10.8	13.2
2 Red.	3.5	5.7	7.8	10.6	13.1
4 Red.	3.1	5.3	7.5	10.4	13.0
8 Red.	2.7	4.8	7.2	10.1	12.7

16 Red.	2.1	4.1	6.6	9.5	12.3
32 Red.	1.8	3.3	5.9	8.8	11.7
64 Red.	1.4	2.7	5.1	8.1	11.0
128 Red.	1.4	2.1	4.3	7.2	10.2

As can be seen the maximum different only account for 3-4 digits between no reduction and a high reduction factor where a higher reduction factor means less loss of digits.

For our e^x function, we use a simple guard digits calculation that we add

2+ceil(log10(digits)) as extra guard digits.

Further Improvement of the Taylor series for e^x?

There is not a lot of things you can do to improve the exp(x) algorithm. However, consider the Taylor series expansion of exp(x):

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (34)$$

The issue is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient scaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\ldots \frac{x^n}{n!} + \frac{x^{n+1}}{(n+1)!} \ldots$$

Moreover, group them:

$$\ldots \frac{(n+1)x^n}{(n+1)n!} + \frac{x^{n+1}}{(n+1)!} \ldots \Rightarrow$$

$$\ldots \frac{(n+1)x^n + x^{n+1}}{(n+1)!} \ldots$$

Then you have replaced one division with an extra multiplication. The (n+1) can be done using a 32-bit or 64-bit integer since you never get to do that many Taylor terms in real life. There is no need to stop at just grouping two terms together you can do that for three terms:

$$\ldots \frac{(n+1)(n+2)x^n + (n+2)x^{n+1} + x^{n+2}}{(n+2)!} \ldots \Rightarrow$$

$$\ldots \frac{x^n(x^2 + (n+2)x + n^2 + 3n + 2)}{(n+2)!} \ldots$$

Saving two divisions, however, gaining a few more addition and multiplications.

In general, you can add a *g* group together:

$$\frac{\sum_{n}^{n+g}(\prod_{i=1}^{g}(n+i))x^{n+i-1}}{(n+g)!} \quad (35)$$

Because arbitrary precision division is, much more time-consuming to calculate it will be highly advantages to implement this grouping of Taylor terms. With four to five terms grouped, you get a speed up of 2-3 times compare to not grouping terms together.

e^x using Sine Hyperbolic function

Less use but the fastest way to calculate exp(x) is using the Sine Hyperbolic function using the identity:

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (36)$$

Where the sinh(x) can be found with the Taylor series (see later section of Hyperbolic functions):

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (37)$$

The sinh(x) Taylor series looks familiar to the Taylor series for exp(x) (every second term is removed):

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (38)$$

Except that, for each term, we go faster towards zero with the sinh(x) and we should expect that we would need fewer Taylor terms for a given precision compare to the exp(x) Taylor series.

Example: e^x using Sinh

Using no argument reduction. We need 9 Taylor terms to get the result compared to 17 for exp(x) using the Taylor series.

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**1**	
Argument reductions=		**0**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.00E+00	1.00000000000	2.4142135624	3.04E-01
2	1.67E-01	1.16666666667	2.7032574095	1.50E-02
3	8.33E-03	1.17500000000	2.7179274124	3.54E-04
4	1.98E-04	1.17519841270	2.7182769296	4.90E-06
5	2.76E-06	1.17520116843	2.7182817840	4.44E-08
6	2.51E-08	1.17520119348	2.7182818282	2.84E-10
7	1.61E-10	1.17520119364	2.7182818285	1.35E-12
8	7.65E-13	1.17520119364	2.7182818285	4.88E-15
9	2.81E-15	1.17520119364	2.7182818285	0.00E+00

Argument Reduction in e^x using Sine Hyperbolic

As for the regular Taylor, series for exp(x), it is clear that we prefer to have our |x| < 1 to ensure that the Taylor series converge more quickly. We again use *argument reduction* to work with a smaller number to get a faster converging to e^x using fewer *terms* of the Taylor series.

We can use the trisection identity: $\sinh(3x) = \mathrm{Sinh}(x)(3 + 4\mathrm{Sinh}(x)^2)$ to reduce the argument with a factor of three and then after the Taylor iterations we restore and find the correct value for sinh(x) by applying this formula the same number of times we did when reducing the argument.

Example: e^x using Sine Hyperbolic with argument reduction

With two reductions, you get the result after only five Taylor terms compare to 12

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**0.111111111**	
Taylor reductions=		**2**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.11E-01	0.11111111111	2.7127251898	5.56E-03
2	2.29E-04	0.11133973480	2.7182783961	3.43E-06
3	1.41E-07	0.11133987592	2.7182818275	1.01E-09
4	4.15E-11	0.11133987596	2.7182818285	1.73E-13
5	7.11E-15	0.11133987596	2.7182818285	0.00E+00

With 8 times reduction you get the result after two 2 Taylor terms compare to 6 using standard exp(x) Taylor series.

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**0.000152416**	
Taylor reductions=		**8**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.52E-04	0.00015241579	2.7182818179	1.05E-08
2	5.90E-13	0.00015241579	2.7182818285	0.00E+00

Granted it is not fair to compare it this way since the standard exp(x) argument reduction is the only factor of two per reduction compare to a factor of three using the sinh(x) trisection identity.

Further Improvement of e^x using Sine Hyperbolic?

The same technique for coefficient scaling (grouping of Taylor terms) can be applied here as well. Consider the Taylor series for sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \tag{39}$$

The issue again clearly is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient scaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\dots\frac{x^n}{n!}+\frac{x^{n+2}}{(n+2)!}\dots$$

Moreover, group them:

$$\dots\frac{(n+1)(n+2)x^n}{(n+1)(n+2)n!}+\frac{x^{n+2}}{(n+2)!}\dots =>$$

$$\dots\frac{(n+1)(n+2)x^n+x^{n+2}}{(n+2)!}\dots$$

Then you have replaced one division with two extra multiplication. The (n+1)(n+2) can be done using 64-bit integer arithmetic since you never get to do some many Taylor terms in real life that it will overflow. There is no need to stop at just grouping two terms together you can do that for three terms or more terms:

For grouping three Taylor terms, you get:

$$\dots\frac{(n+1)(n+2)(n+3)(n+4)x^n+(n+3)(n+4)x^{n+2}+x^{n+4}}{(n+4)!}\dots =>$$

$$\dots\frac{(n+3)(n+4)(\,(n+1)(n+2)x^n+x^{n+2})+x^{n+4}}{(n+4)!}\dots$$

e^x using the binary splitting method

This method expands on the same method for calculating e, see the section on constants.

It used the Taylor series for exp(x):

$$\exp(x)=1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}+\frac{x^5}{5!}+\cdots \qquad (40)$$

However, instead of calculating the series as above we implement it using the binary splitting method.
The binary splitting methods (see [12]) equate the Taylor series terms with two variables *p* and *q* and then it is just a matter of dividing *p* with *q* to get the approximation for e.

$$e=\sum_{k=0}^{\infty}\frac{1}{k!}=1+\frac{x}{1}+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\cdots=\frac{P(0,k)}{Q(0,k)} \qquad (41)$$

Here Q(0,k) is an integer, but P(0,k) is a *float_precision* variable. (Since x can be any real value). The notation P(0,k)/Q(0,k) represents the first k terms of the above series. For any given value of a & b, we can compute P(a,b) and Q(a,b) as follows using the binary splitting method. (a and b are integers and a<b) following the recursion:

Algorithm: Binary splitting method for e

$m=\frac{a+b}{2}\ integer\ division$

P(a,b)=P(a,m)Q(m,b)+P(m,b)

Q(a,b)=Q(a,m)Q(m,b)
And P(b-1,b)=x^b; Q(b-1,b)=b;

Algorithm 6

You continue this recursive breakdown until a+1=b and you set P(a,b)=x^b and Q(a,b)=b and let the formula reverse bottom up.

Argument reduction for e^x for the binary splitting method

Now to make the algorithm efficient we need to ensure that |x| <1. That can be done easily by just using argument reduction as previously describe under exp(x) using the Taylor series. We expect that if |x| << 1 then the Taylor series will converge faster.

To calculate how many Taylor terms we need as a function of required decimal digits of e. We resort to the Stirling approximation formula for !. We notice that to get P decimal precision of e^x and the number of Taylor terms is k we need it to satisfy the equation that:

$$\frac{x^k}{k!} < 10^{-P} \tag{42}$$

Where we use the Stirling approximation for k!:

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\ approximation \tag{43}$$

This yield:

$$\frac{x^k}{\left(\frac{k}{e}\right)^k \sqrt{2\pi k}} < 10^{-P} \tag{44}$$

Taking log() on both sides you get:

$$-k \cdot \log(x) + k \cdot (\log(k) - 1) + \frac{1}{2}\log(2\pi k) > P \cdot log(10) \tag{45}$$

To solve this for k, we can use Newton's methods that find a solution within a few iterations. Notice we only need to find the next higher integral number for k.

Taylor terms needed as a function of x

Digits	10	100	1,000	10,000	100,000	1,000,000
x						
1	14	70	450	3,249	25,206	205,022
10^{-1}	7	45	325	2,521	20,502	172,350
10^{-2}	5	33	252	2,050	17,235	148,429
10^{-3}	4	25	205	1,724	14,843	130,202
10^{-4}	3	21	173	1,484	13,020	115.878
10^{-5}	2	18	149	1,302	11,588	104,339
10^{-6}	2	15	130	1,159	10,434	94,852
10^{-7}	2	13	116	1,044	9,485	86,920
10^{-8}	2	12	105	949	8,692	80,194
10^{-9}	2	11	95	869	8,020	74,419

The above table clearly shows the effect of using the argument reduction technic in the binary splitting method. We can apply the same argument reduction formula already established at the start of the explanation of e^x.

Finding a reasonable reductions factor for e^x

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? Yacas book [6] states that at least x should be lower to $|x|<10^{-M}$ and M should be:

$$|x| < 10^{-M} \; and \; M > \frac{\ln(P)}{\ln(10)} \tag{46}$$

Where P is the precision in decimal digits.

Measuring the performance indicates that this is not the most optimal selection. Therefore, we multiply the M found above with a constant eight to get a more reasonable reduction factor and then adjust for the magnitude of |x| itself.

The adjustment for the magnitude of |x| is simply the number exponent (power of 2 exponent) to ensure that the number will be well below one. This works well for small magnitude |x| and for high magnitude |x|. By just adding the exponent (positive or negative) to the reduction factor.

The precision needed, to avoid loss of accuracy.

Looking at the algorithm we can see for P(a,b):

$$P(a,b)=P(a,m)Q(m,b)+P(m,b) \tag{47}$$

We multiply each P(a,m) with Q(m,b) where Q is the factorial. This will create a pretty big number as we increase the number of terms we need. To see how big we can again use the Stirling approximation for !.

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\; approximation \tag{48}$$

Using log10(k!) We find the number of decimal digits as the size of k!

$$\log\ 10(k!) \approx log10\left(\left(\frac{k}{e}\right)^k \sqrt{2\pi k}\right) =>$$

$$\mathrm{k}\cdot log10(k) - k + \frac{1}{2} log10(2\pi k) \approx \mathrm{k}\cdot log10(k) - k, for\; large\; k \tag{49}$$

Digits	10	100	1,000	10,000	100,000	1,000,000
Size of k! in decimal digits	9	100	2,000	30,000	400,000	5,000,000

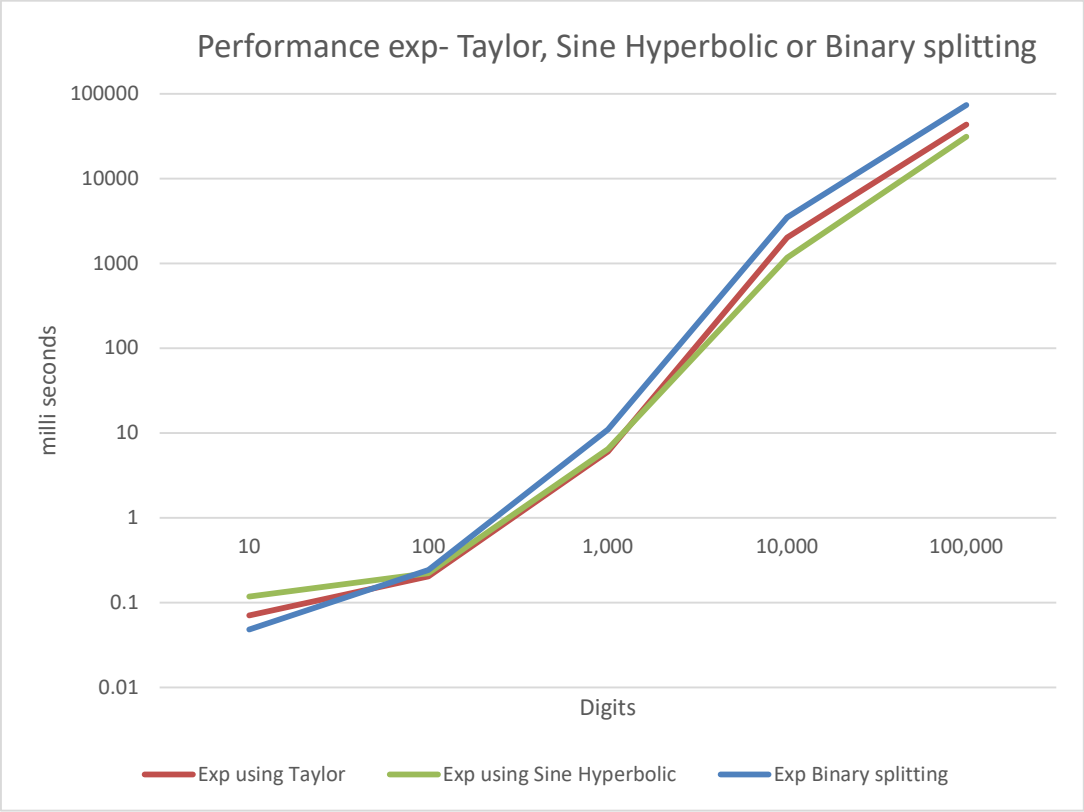
Table of the decimal size of various values for !.

As expected, !. Is a powerful factor where we need to adjust upward the needed accuracy or precision when we calculate ex at some precision. The adjustment amount is much larger

than we are used to dealing with using regular methods for e^x. However, if we use argument reduction it counteracts the need to handle calculation with a significantly higher number of digits.

Which method to use for e^x?

By measuring the performance, we get clear advantages of using the sine hyperbolic function to calculate e^x, particularly with an increasing number of digits. The use of the Binary splitting method is interesting but lacks the performance of the two other methods.



Time in milliseconds between the three methods for evaluating e^x

Logarithmic functions:

There are quite a few ways you can calculate log(x) in arbitrary precision. Traditional Taylor series expansion has been used however, another method involving AGM (Arithmetic-Geometric Mean) has shown to be an efficient method of calculating log(x): This chapter will examine this.

1. Log(x) using Taylor series, argument reduction, and coefficient scaling.
2. Using Newton 2nd order method to calculate log(x)
3. Using Halley 3rd order method to calculate log(x)
4. Using AGM algorithm to calculate log(x)

The most common one for arbitrary precision libraries is the standard Taylor series expansion method but as will be shown this is not the preferred choice if you want performance. When we say log(x) with mean the natural logarithm is denoted as ln(x). For other bases, we will explicitly refer them to $\log_{10}(x)$ or $\log_2(x)$ to avoid any confusion.

Log(x) using the Taylor series

For the function, log(x) or the natural logarithm ln(x) we could use the corresponding Taylor series for ln(x) as defined by:

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots \quad (50)$$

Which is valid for $0<x\leq 2$. The limit range is usually not a problem since we can use argument reduction to get x within the limit. The series however converge slowly to ln(x) and is not suitable for arbitrary precision. Instead, most implementations use the inverse hyperbolic tangent function:

$$\ln(x) = 2 \cdot \operatorname{artanh}\left(\frac{x-1}{x+1}\right) = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right) \quad (51)$$

Which is valid for any real number x>0.

These series converge with reasonable speed if x is small.

Example 1. Ln(x) using Taylor series

Using x=2 we get after 15 Taylor series the result of ln(2)= 0.693147180559945

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**2**	
Taylor reductions=		**0**		
Terms	**z**	**Term Sum**	**Ln(x)**	**Error**
1	3.3333E-01	0.333333333333333	0.666666666666667	2.65E-02
2	3.7037E-02	0.345679012345679	0.691358024691358	1.79E-03
3	4.1152E-03	0.346502057613169	0.693004115226337	1.43E-04
4	4.5725E-04	0.346567378666144	0.693134757332288	1.24E-05
5	5.0805E-05	0.346573023695414	0.693146047390827	1.13E-06

6	5.6450E-06	0.346573536879893	0.693147073759785	1.07E-07
7	6.2723E-07	0.346573585128006	0.693147170256012	1.03E-08
8	6.9692E-08	0.346573589774121	0.693147179548241	1.01E-09
9	7.7435E-09	0.346573590229622	0.693147180459244	1.01E-10
10	8.6039E-10	0.346573590274906	0.693147180549812	1.01E-11
11	9.5599E-11	0.346573590279458	0.693147180558916	1.03E-12
12	1.0622E-11	0.346573590279920	0.693147180559840	1.05E-13
13	1.1802E-12	0.346573590279967	0.693147180559934	1.10E-14
14	1.3114E-13	0.346573590279972	0.693147180559944	1.33E-15
15	1.4571E-14	0.346573590279972	0.693147180559945	0.00E+00

That is not too bad, however, if we change the argument to 10 then we need 75 Taylor's terms to get the result and if we use x=0.1 then we also need 75 Taylor terms. With x=1.1 you only need six Taylor Terms.
This lead to the observation that the number of Taylor's terms needed depends heavily on the argument to ln(x) and how close it is to one.

Argument Reduction

We prefer to have our x in a small neighborhood around one to ensure that the Taylor series converges more quickly. We can accomplish that using a technique called *argument reduction* to work with a smaller number to get a faster converging to ln(x) using fewer *terms* of the Taylor series.
We can use the identity:

$$\ln(x) = \ln\left(\left(\sqrt{x}\right)^2\right) = 2 \cdot \ln\left(\sqrt{x}\right) \qquad (52)$$

to reduce the argument by repeating take the square root of x until it gets closer to 1. If we take k square roots, reducing $x => x^{\frac{1}{2^k}}$ and gets closer to one we can then after the Taylor iterations multiply the result with 2^k to find the correct value of ln(x).

This makes sense to reduce the need for Taylor terms since each Taylor terms involve a division, which is very time-consuming in arbitrary precision arithmetic.

Example 2: ln(x) using Taylor series with argument reduction

If using the previous example 1 and reducing the argument twice from two to 1.1892… we only need 7 Taylor terms to get the same result as before, saving eight Taylor terms but gaining two squaring and multiplication of 2^2 =4 at the end. However, overall huge savings since we have avoided eight time-consuming divisions in Taylor's terms.

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**1.189207115**	
Taylor reductions=		**2**		
Terms	**z**	**Term Sum**	**Ln(x)**	**Error**
1	8.6427E-02	0.086427233725890	0.691417869807118	1.73E-03
2	6.4558E-04	0.086642427936652	0.693139423493214	7.76E-06
3	4.8223E-06	0.086643392394074	0.693147139152589	4.14E-08
4	3.6021E-08	0.086643397539913	0.693147180319306	2.41E-10
5	2.6906E-10	0.086643397569809	0.693147180558474	1.47E-12

6	2.0098E-12	0.086643397569992	0.693147180559936	9.33E-15
7	1.5013E-14	0.086643397569993	0.693147180559945	0.00E+00

If we use an eight-times reduction we get the same results after just four Taylors terms.

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**1.002711275**	
Taylor reductions=		**8**		
Terms	**z**	**Term Sum**	**Ln(x)**	**Error**
1	1.3538E-03	0.001353802259956	0.693146757097522	4.23E-07
2	2.4812E-09	0.001353803087030	0.693147180559489	4.56E-13
3	4.5475E-15	0.001353803087031	0.693147180559955	-9.21E-15
4	8.3346E-21	0.001353803087031	0.693147180559955	-9.21E-15

The issue with arbitrary precision for ln(x)

15 Taylor's terms to reach a result do not seem so bad at a first glance. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result.

Now it would come in very handy if we could estimate the needed number of Taylor terms for a given argument so we can optimize the use of argument reduction. Luckily, this can be estimated for ln(x). The n^{th} -Taylor term for ln(x) is given by:

$$2 \cdot \frac{z^{2n-1}}{2n-1}, where\ z = \frac{x-1}{x+1} \tag{53}$$

Generally, we can stop the iteration when $2 \cdot \frac{z^{2n-1}}{2n-1} < 10^{-P}$ Where P is the decimal precision. Now taking ln on both sides, rearranging and reducing we get:

$$\ln\left(2\frac{z^{2n-1}}{2n-1}\right) = \ln(10^{-P}) => (2n-1)\ln(z) - \ln(n) + \ln(2) = -P \cdot \ln(10) =>$$

$$\ln(n)\ and\ \ln(2)\ can\ be\ ignored\ for\ large\ p\ \approx (2n-1)\ln(z) = -P \cdot \ln(10) =>$$

$$n = \frac{1}{2}\left(\frac{-P \cdot \ln(10)}{\ln(z)} + 1\right) \tag{54}$$

If we use the example of x=2 we get the following estimated Taylor's terms as a function of precision without argument reduction.

Taylor terms needed:							
x/precision	**10**	**16**	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
2	11	17	105	1,048	10,480	104,796	1,047,952

Now to see the effect of argument reduction on improving the Taylor series we have recorded the amount of Taylor terms needed for various argument reductions from 1 to 8 on a random floating-point number between 1.xxx and 1.999. From the table, we see that the reduction in

the number of Taylor terms varies more than 8-10 fold between 0 as the reduction factor to a reduction factor of 8.
The Auto reduction is the number of Taylor terms when we automatically find a reasonable reduction factor. Most of the time it varies between 8-10 reductions.

The number of Taylor Terms.

Digits	10	100	1,000	10,000	100,000
Auto Red.	4	16	151	1,416	14,397
0 Red	17	65	747	9,283	104,166
1 Red.	12	48	519	6,024	65,054
2 Red.	9	38	397	4,431	46,887
3 Red.	7	32	321	3,500	36,587
4 Red.	6	27	270	2,892	29,987
5 Red.	5	24	233	2,464	25,403
6 Red.	5	21	205	2,146	22,034
7 Red.	4	19	183	1,901	19,454
8 Red.	4	18	151	1,706	15,762

Finding a reasonable reduction factor for ln(x).

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? That at least x should be reduced to some arbitrary number. I use 1.001 as the target for ref [1]

First, eliminate the exponent of x reducing it to a number $\geq 1\ x < 2$.

$$\text{Solve } x^{\frac{1}{2^k}} < \text{limit} =>$$

$$\ln\left(x^{\frac{1}{2^k}}\right) < \ln(\text{limit}) => \frac{1}{2^k} \cdot \ln(x) < \ln(\text{limit}) =>$$

$$\frac{\ln(x)}{\ln(\text{limit})} < 2^k => \ln\left(\frac{\ln(x)}{\ln(\text{limit})}\right)/\ln(2) < k \qquad (55)$$

A reasonable number for the limit is 1.001 If x=2 then you would need to perform 10 reductions before summing the Taylor terms. After summarizing the Taylor terms, you would need to multiply that number by 2^{k+1} to get the correct value for ln(x).

The performance table below shows the effect of using increasingly higher reduction factors.

All measures are in milliseconds

Digits	100	1,000	10,000	1,000,000
Auto Red.	1.57	26	14,625	739,917
0 Red.	3.67	113	93,300	5719,740
1 Red.	2.75	99	62,262	3180,440
2 Red.	2.4	59	47,735	2316,740
3 Red.	1.25	48	36,048	2045,750
4 Red.	1	43	29,021	1,500,050
5 Red.	0.91	36	24,548	1,309,860
6 Red.	1	34	21,391	1,148,780

7 Red.	0.91	31	19,182	957,246
8 Red.	0.91	29	16,657	864,200

As you can see for the higher number of precision, you will benefit even with increasing the reduction factor.

Guard Digits for ln(x) calculation

When summarizing a Taylor series as ln(x) you need quite a lot of summarizing and that will produce round-off errors.

For our ln(x) function, we use a simple guard digits calculation that we add

2+ceil(log10(digits)) as extra guard digits.

Further Improvement of the methods for ln(x)?

There is not a lot of things you can do to improve the ln(x) algorithm. However, consider the Taylor series expansion of ln(x):

$$\ln(x) = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right) \quad (56)$$

If we use $z = \frac{x-1}{x+1}$ we get:

$$\ln(x) = 2\left(z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \cdots\right) \quad (57)$$

As was the case when we discuss this in the exponential function paper, the issue is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient scaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\ldots \frac{x^n}{n} + \frac{x^{n+2}}{n+2} \ldots$$

Moreover, group them:

$$\ldots \frac{(n+2)x^n}{(n+2)n} + \frac{n \cdot x^{n+2}}{n(n+2)} \ldots =>$$

$$\ldots \frac{(n+2)x^n + n \cdot x^{n+1}}{n(n+2)} \ldots$$

Then you have replaced one division with three extra multiplication. The (n+2) can be done using a 32-bit or 64-bit integer since you never get to do many Taylor terms in real life. There is no need to stop at just grouping two terms together you can do that for three terms:

$$\ldots \frac{(n+2)(n+4)x^n + n(n+4)x^{n+1} + n(n+2)x^{n+2}}{n(n+2)(n+4)} \ldots$$

Saving two divisions, however, gaining a few more addition and multiplications.

Because arbitrary precision division is, much more time-consuming to calculate it will be highly advantageous to implement this grouping of Taylor terms. With four to five terms grouped, you get a speedup of 2-3 times compared to not grouping terms together.

Log(x) using the Newton method

This method is only relevant if you have a very fast way to compute e^x. This usually is the case since exp(x) is faster to calculate than ln(x) when using arbitrary precision. The method solves the equation x=ln(y) by taking the exp() of both sides: exp(x) = y and then solving it using the Newton method, which yields the iteration:

$$x_{n+1} = x_n - 1 + \frac{y}{e^{x_n}} \quad (58)$$

Algorithm 7

Unfortunately, it will require a division; however, e^x is more time-consuming to calculate than a division so it does not matter in the big picture. The Newton method has a quadratic convergence rate doubling the number of correct digits for each iteration. For precision, less than 10,000 digits the Taylor series from the previous chapter is faster but above 10,000 digits the Newton method exceeds the performance of the Taylor series. At 100,000 digits Newton's method is approximately 40% faster than the Taylor series.

Log(x) using the Halley method

Since the Newton method is faster than the Taylor series for precision above 10,000 digits it is interesting to check if the cubic convergence Halley method is even faster. The Halley method with cubic convergence is:

$$x_{n+1} = x_n + 2\frac{y - e^{x_n}}{y + e^{x_n}} \quad (59)$$

Algorithm 8

The benefit is that you triple the number of correct digits per iteration versus Newton double per iteration. The Halley method is indeed faster exceeding the Newton method around a 1,000 digits precision and is approximately 8-10% faster than the Newton Method.

Log(x) using the AGM method

The AGM method is the method that has the best asymptotic performance of all the methods. It was found around 1975 and is described in the Yacas book [6]:

$$\ln(x) = \pi \cdot x \frac{1 + \frac{4}{x^2}\left(1 - \frac{1}{\ln(x)}\right)}{2 \cdot AGM(x,4)} \quad (60)$$

It looks more complex than any of the other methods but the trick is to observe that if x is "large enough" then the numerator is one. For a given precision "large enough" mean that $\frac{4}{x^2} < 10^{-P}$, where P is the wanted precision. In case x is not "large enough" we need to multiply it with 2^s. (Which is argument expansion and not argument reduction as we are used to) Since we expand the argument with a factor of 2^s we would need to subtract it after the AGM method with $s \cdot \ln(2)$:

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) \quad (61)$$

For a given precision, P, s is found below:

$$s = P\frac{\ln(10)}{2 \cdot \ln(2)} + 1 - \frac{\ln(x)}{\ln(2)} \quad (62)$$

With all components in place, we can now devise our AGM algorithm:

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) = \frac{\pi \cdot x^{s-2}}{2 \cdot AGM(x^{s-2},1)} - s \cdot \ln(2), \text{for } x > 1 \quad (63)$$

If x<1 then we use the identity $\ln(x) = -\ln(\frac{1}{x})$ and use the AGM algorithm with 1/x.
Even though we are using two arbitrary precision constants, π and ln(2) that needs to be calculated to the same precision, P and we need to perform approximately $2\frac{\ln(P)}{\ln(2)}$ iterations to calculate the AGM value the method outperformed any of the other methods presented here for precision exceeding approximately 4,000 digits. See the log(x) performance chart.

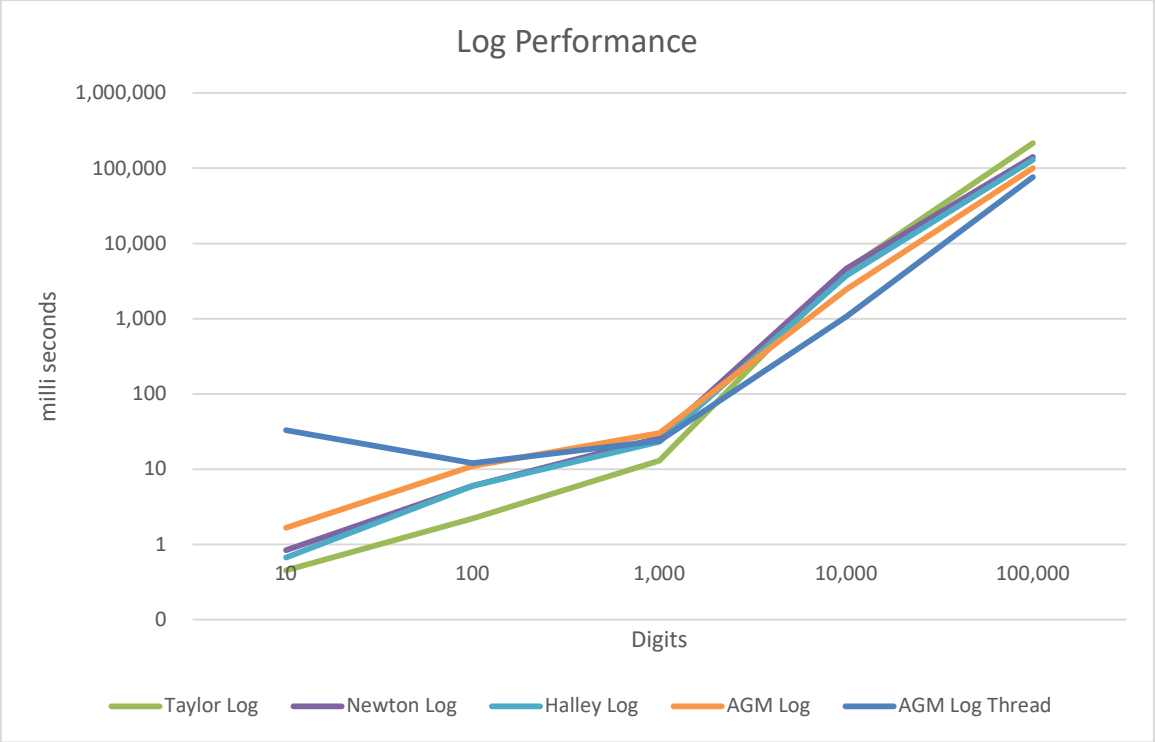
AGM Algorithm

The arithmetic-geometric mean algorithm is defined as two positive numbers x & y by the following algorithm AGM(x,y)= $\lim_{n\to\infty} x_n = \lim_{n\to\infty} y_n$.

AGM(x,y)
- a_o=x
- g_0=y
- iterate:
  - $a_{n+1} = \frac{1}{2}(a_n + g_n)$
  - $g_{n+1} = \sqrt{a_n g_n}$
- until a_{n+1}=g_{n+1}
- return a_{n+1}

Algorithm 9

Log(x) performance



Log Performance.

Based on the performance chart Taylor series log is the fastest up to approximately 4,000 digits whereas after the log AGM becomes the fastest in both the threaded and non-threaded versions. Above 10,000 digits, the Newton and Halley method also exceeds the performance of the Taylor series version.

Log(x) using the AGM method and multiple threads

The AGM method lends itself to being implemented using threads. There are three basic components of the AGM method.

- Calculating the constant π
- Calculating the constant ln(2)
- Calculating the AGM value

These three calculations can run in parallel in separate threads with a few simple changes to the source code using C++ lambda functions.

Recommendation for calculating log(x)

Based on the performance measure of the various ln() methods recommend:

- Ln(x) using Taylor series with argument reduction and coefficient scaling for precision up to approx. 4,000 digits.
- If the AGM method is available then use it above 4,000 digits.
- Moreover, use AGM in multi-threaded versions to increase performance.
- If the AGM method is not available then use either the Newton method or the better Halley method when precision exceeds 10,000 digits.

- Always use argument reduction to increase performance
- Coefficient scaling (or grouping of terms) can speed up calculation by a factor of two-three and is therefore recommended.

$Log_{10}(x)$:

To calculate $Log_{10}(x)$ we use the equation: $Log_{10}(x)=Log_e(x)/Log_e(10)$ and $log_e(x)$ has been handled previously. $Log_e(10)$ is a constant, see a later section of this document.

x to the power of y

To calculate x^y we use a combination of the exponential and logarithm function using the equation:

$$x^y = e^{y \cdot \ln(x)} \quad (64)$$

Exp() and Log() is a time-consuming functions for arbitrary precision. There is nothing much you can do about that unless *y* is an integer in which case we use the algorithm for integer power listed below: if y is an integer then we can rewrite the equation of x^y:

$$x^y = (x)^{\frac{y}{2}+\frac{y}{2}} = (x)^{\frac{y}{2}}(x)^{\frac{y}{2}} = (x \cdot x)^{\frac{y}{2}} \quad (65)$$

Algorithm for x^y when y is an integer

```
function ipower(x,y)
        r=1
        while(y>0)
                if(y is odd)
                        r=r*x
                x=x*x
                y=y/2
        return r
```

Algorithm 10

If y is < 0 we use $x^{-y} = \frac{1}{x^y}$ and return $\frac{1}{r}$ instead of r in the algorithm above. Now if we happen to come across x as a true power of 2 and y is an integer then we can make further optimization, by noticing that:

$$x = 2^n \; and \; n \; and \; y \; is \; an \; integer \Rightarrow (2^n)^y = 2^{n \cdot y} \quad (66)$$

Constants: e, $Log_e(2)$, $Log_e(10)$ & π

Constants are not constants since it depends on the actual precision we need. We need the following constants: e, $Log_e(2)$, $Log_e(10)$, and π available for the actual precision of the operations. To avoid repeated calculations of the same constant we store the constant and reuse it the next time we need one of these constants. e.g. let's assume we need one of the constants with 100,000 digits precisions. We then calculate this constant only once. The next time we need the constant with equal or less precision (≤100,000 digits) we then used the stored constant and round it to the precision needed ≤ 100,000 digits). If on the other hand, we need the constant with higher precision we then discard the constant stored and recalculate the constant with the higher precision and used that as the new stored constant.

The constant e

The transcendental constant e (same as exp(1)) can be more beneficial calculated by other methods than the ones presented in the previous section. There is a Spigot-like algorithm from the computer Journal 1968 (A H J Sale) that I have modified to serve the purpose of use in the arbitrary precision library. The algorithm is a magnitude faster than using the Taylor series for calculating exp(1) even with the enhancement presented in this paper. Please ref to [11] for further details. However this method is not the fastest one, see the binary splitting method for e.

AHJ Sale algorithm for e

The algorithm was presented in [11] back in the sixties and accompanied by an Algol 60 version. The original code has been ported to the C++ environment with a few additional improvements. The result of the calculation is delivered as a decimal string see the source code below. The function is called with the wanted number of digits for e. Based on this the needed number of Taylor Terms is calculated and then the main loop delivers one decimal number per loop.

Binary splitting method for e

The binary splitting methods (see [12]) equate the Taylor series terms with two integers p and q and then it is just a matter of dividing p with q to get the approximation for e.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \cdots = \frac{P(0,k)}{Q(0,k)} \qquad (67)$$

The notation P(0,k)/Q(0,k) represents the first k terms of the above series. For any given value of a & b, we can compute P(a,b) and Q(a,b) as follows using the binary splitting method. (a and b are integers and a<b) following the recursion:

Algorithm: Binary splitting method for e

$m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)
Q(a,b)=Q(a,m)Q(m,b)

And P(b-1,b)=1; Q(b-1,b)=b;

Algorithm 11

You continue this recursive breakdown until a+1=b and you set P(a,b)=1 and Q(a,b)=b and let the formula reverse bottom up.

Notice if you need more than the first 19 Taylor Terms you will need more than 64-bit variables to hold p and q. You would need to switch to arbitrary integer precision. This is done using the type *int_precision* (instead of e.g. uintmax_t for 64-bit environment) from the author's arbitrary precision packages.

To calculate how many Taylor terms we need as a function of required decimal digits of e. We resort to the Stirling approximation formula for! We notice that to get to P decimal precision of e and the number of Taylor terms is k we need it to satisfy the equation that $k!>10^P$.

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\ approximation$$

$$\left(\frac{k}{e}\right)^k \sqrt{2\pi k} > 10^P$$

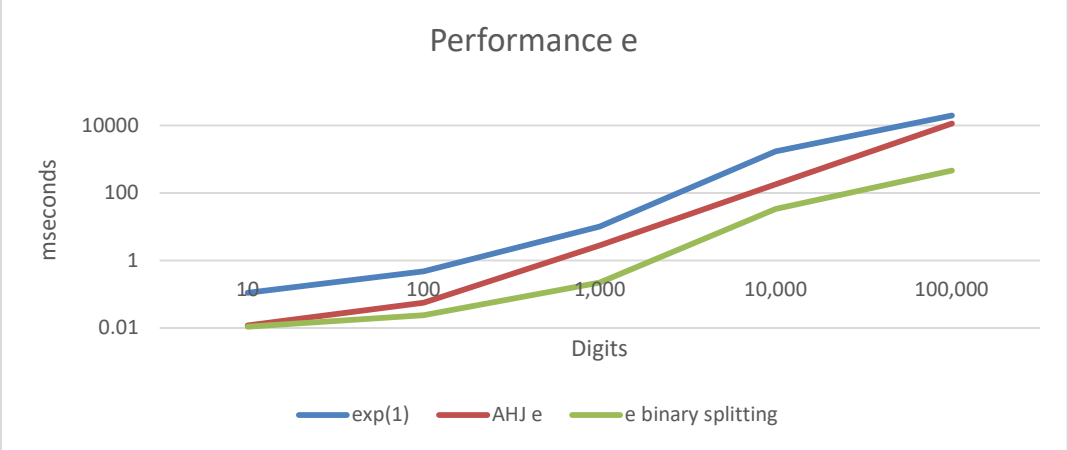
Taking log() on both sides you get:

$$k \cdot (\log(k) - 1) + \frac{1}{2}\log(2\pi k) > P \cdot log(10) \qquad (68)$$

To solve this for k, we can use Newton's methods that find a solution within a few iterations. Notice we only need to find the next higher integral number for k.

To reduce the number of recursive calls and increase the performance you would not have to wait until a+1=b before setting p, q for the first time. We can use a pre-calculated formula that calculates p and q directly when a+2=b, a+3=b, and a+4=b to reduce the number of recursive calls we make and increase the performance.

Performance for various e algorithm



Recommendation for calculating e

Use the binary splitting method, which is approx. 20 times faster than the AHJ Sale method when calculating e with 100,000 digits. The binary splitting method is approx. 40 times faster than using the Taylor series for exp(1) describe in the previous chapter.

The constant $Log_e(2)$

Is calculated the same way as for $log_e(x)$ with the exception that we can determine a fixed number of argument reductions using square root. Since the argument is 2 we know that squaring it twice will reduce the number below 1.19 and thereby make the Taylor series converge rapidly. After we have found the result to the given precision we then multiply the result by 2^3 to compensate for the argument reduction. However, in our newer version, we have applied the dynamic argument reduction to further speed up the calculation as discussed under **$Log_e(x)$.** However, as for e, there exist other methods that are faster. [13]

The constant $Log_e(10)$

Is calculated the same as for $Log_e(2)$ except that we square 10 four times in the argument reduction to get the number below 1.16 and then use our Taylor series to quickly find the result of $Log_e(10)$ and in the back end we multiply with 2^5 instead of 2^3
However, in our newer version, we have applied the dynamic argument reduction to further speed up the calculation as discussed under **$Log_e(x)$.** Again as for loge(x), there exist other algorithms that are faster. See [13].

The constant π

This is another interesting constant that there has been devoted much attention to for the last many thousand years. For a very good walkthrough of different algorithms to calculate π we recommend you read [5] Borwein, "PI and the AGM". However [12]
For our implementation, we can use one of the many algorithms for calculating π that can be found in [5][3][14][15][16]. In [17] we walk through many of these algorithms for π with practical examples of implementations.

Borwein π

Algorithm for Borwein π

$Set\ x_0 = \sqrt{2}, \pi_0 = 2 + \sqrt{2}, y_0 = \sqrt[4]{2}$

The repeat for i=1,2,3,… until sufficient accuracy has been obtained.

$$x_{i+1} = \frac{1}{2}\left(\sqrt{x_i} + \frac{1}{\sqrt{x_i}}\right)$$

$$\pi_{i+1} = \pi_i\left(\frac{x_{i+1}+1}{y_i+1}\right)$$

$$y_{i+1} = \frac{y_i\sqrt{x_{i+1}} + \frac{1}{\sqrt{x_{i+1}}}}{y_i+1}$$

Algorithm 12

Until sufficient precision has been obtained for π. The nice part of this algorithm is that we only use basic operations like +,*, /, and then the square root function.

To see how the algorithm works let's calculate π.
As we can see after 3 iterations we have found π to the limit of IEEE754 arithmetic

Iteration	x	π	y	Error
0	1.414214	3.41421356237309	1.189207	0.272621
1	1.015052	3.14260675394162	1.000673	0.001014
2	1.000028	3.14159266096604	1	7.38E-09

3	1	3.14159265358979	1	0

Brent-Salamin π

Another algorithm and slightly better than the Borwein algorithm is the Gauss-Legendre and the deviation is also known as Bent-Salamin.

Algorithm for Brent-Salamin π

$$Set\ a_0 = 1, b_0 = \frac{1}{\sqrt{2}}, c_0 = 0.5$$

Then repeat for n=0,1,2… until sufficient accuracy has been obtained.

$$a_{n+1} = \frac{1}{2}(a_n + b_n)$$
$$b_{n+1} = \sqrt{a_n b_n}$$
$$c_{n+1} = c_n - 2^{n+1}(a_{n+1} - b_n)^2$$
$$\pi_{n+1} = 2\frac{a_{n+1}^2}{c_{n+1}}$$

Algorithm 13

Brent-Salamin

π

Iteration	a	b	C	Π	Error
0	1	0.707107	0.5		8.58E-01
1	0.853553	0.840896	0.457107	3.18767264271211	4.61E-02
2	0.847225	0.847201	0.456947	3.14168029329765	8.76E-05
3	0.847213	0.847213	0.456947	3.14159265389545	3.06E-10
4	0.847213	0.847213	0.456947	3.14159265358979	8.88E-16

As for the Borwein algorithm, we get quadratic convergence doubling the number of correct digits for each iteration. After 10 iterations we have more than 1,000 digits and after 20 iterations more than 1 million digits. Borwein also showed several higher-order convergence rate algorithms for finding π, with a convergence rate for each iteration that multiplies the number of correct digits with a factor of 3, 4, 5, and 9. However, these algorithm requires a lot more work to be done per iteration and is usually not worth implementing compare to the current one with quadratic convergence.

Binary splitting of the Chudnovsky infinite series

There is one method that beats all the classic methods as outlined in [17] and that is the Chudnovsky method using binary splitting.

Instead of adding each series of terms we instead try to find two integers, P & Q that equate to the first k terms of the series.

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(13591409 + 545140134n)}{(3n)!\,(n!)^3 640320^{3n+3/2}} =>$$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(13591409 + 545140134n)}{(3n)!\,(n!)^3 640320^{3n}} =>$$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\frac{P}{Q}$$

Given the first k terms of the series, you get (see [15]):

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\frac{P(0,k) + 13591409Q(0,k)}{Q(0,k)} =>$$

$$\pi = \frac{4270934400 \cdot Q(0,k)}{P(0,k)+13591409\ \ (0,k)}\frac{1}{\sqrt{10005}} + O(151931373056000^{-k}) \quad (69)$$

Where k, is found to satisfy the precision of the number. E.g. for precision P we have equality:

$$10^{-P} < 151931373056000^{-k} => k > \frac{P \cdot log(10)}{\log(151931373056000)}$$

We take k as the ceiling of:

$$k = \left\lceil \frac{P \cdot log(10)}{\log(151931373056000} \right\rceil \quad (70)$$

Algorithm for Chudnovsky method for π using binary splitting

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b:

$m = \frac{a+b}{2}\ integer\ division$

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)

Q(a,b)=Q(a,m)Q(m,b)

R(a,b)=R(a,m)R(m,b)

Where:

P(b+1,b)=(13591409+545140134b)(2b-1)(6b-5)(6b-1)(-1)^b

Q(b-1,b)=10939058860032000b^3

R(b-1,b)= (2b-1)(6b-5)(6b-1)

Algorithm 14

Recommendation for the Infinite series for π

I recommend always using the binary spitting algorithm for Chudnovsky, which has the fastest performance of them all. It is no surprise that it is the Chudnovsky binary splitting method that is used in the record-breaking calculation of π with 100 Trillion digits (2022).

Trigonometric functions:

There are quite a few ways you can calculate trigonometric functions with arbitrary precision. Traditional Taylor series expansion has been used, however. This section will examine:

- Sin(x) using Taylor series, argument reduction, and coefficient scaling.
- Cos(x) using Taylor series, argument reduction, and coefficient scaling.
- Tan(x) using various methods.
- Arcsin(x) using Taylor series, argument reduction, and coefficient scaling
- Arccos(x) using arcsin(x)
- Arctan(x) using Taylor series, argument reduction, and coefficient scaling.
- Arctan(x) using other methods.

The most common one for arbitrary precision libraries is the standard Taylor series expansion method.

Sin(x) using Taylor Series

The standard way of calculating sin(x) using the Taylor Series. Sin(x) can be found with the Taylor series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (71)$$

Where the similarity to the sine hyperbolic functions is obvious, which Taylor series is:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (72)$$

Where the only difference is the alternating sign between the Taylor Terms. Sin(x) is defined for any real number.

However, before we start the Taylor series we first reduce the argument x. We will do that in four steps.

Step 1: We notice that sin(x) is cyclic with a period of 2π so we can easily reduce any argument > 2π so it falls between zero and 2π by simply taking x modulo 2π.

Step 2: We can further reduce x so it is between 0..π using the identity:
sin(x)=-sin(x-π) for x≥π.

Step 3: We reduce it further by using the symmetry around $\frac{\pi}{2}$ $to\ the\ range\ 0..\frac{\pi}{2}$:

$$sin(x) = sin\left(x - \frac{\pi}{2}\right) for\ x \geq \frac{\pi}{2}$$

If π is 'expensive' to calculate (which is usually the case with arbitrary precision) we can omit step 3 since we have a different way to obtain the same thing by just increasing the argument reduction factor. See the section on finding a reasonable reduction factor.

Step 4: Finally we reduced the argument *k* number of times using the trisection identity:
$\sin(3x)=3\sin(x)-4\sin^3(x)$

until x is below a certain threshold. It is obvious from the sin(x) Taylor series that the smaller x is the fewer terms we would need.

This argument reduction is done to reduce the number of Taylor iterations and to minimize the round-off errors and calculation time.

After the Taylor series has converged, we use the trisection identity reverse *k* numbers of times to find our result for sin(x).

Example 1. Sin(x) using the Taylor series

To see how this algorithm works let us find the sin(0.7). After the 8th Taylor term, the error is zero and the result is ~ 0.6442176872.

sin(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.7**	
Taylor reductions=		**0**		
Terms	**Term value**	**Term Sum**	**sin(x)**	**Error**
1	7.00E-01	0.70000000000	0.7000000000	-5.58E-02
2	5.72E-02	0.642833333	0.6428333333	1.38E-03
3	1.40E-03	0.64423391667	0.6442339167	-1.62E-05
4	1.63E-05	0.64421757653	0.6442175765	1.11E-07
5	1.11E-07	0.64421768773	0.6442176877	-4.94E-10
6	4.95E-10	0.64421768724	0.6442176872	1.55E-12
7	1.56E-12	0.64421768724	0.6442176872	-3.66E-15
8	3.63E-15	0.64421768724	0.6442176872	0.00E+00

Example 2. Sin(x) using Taylor series and argument reduction

We can see the effect in Step 4 by increasing the number of argument reductions. E.g. for two reductions you get the same result after only five iterations. The argument is reduced twice from 0.7 to 0.077…

sin(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.077777778**	
Taylor reductions=		**2**		
Terms	**Term value**	**Term Sum**	**sin(x)**	**Error**
1	7.78E-02	0.07777777778	0.6447587967	-5.41E-04
2	7.84E-05	0.07769936	0.6442175235	1.64E-07
3	2.37E-08	0.07769938357	0.6442176873	-2.36E-11
4	3.42E-12	0.07769938357	0.6442176872	2.00E-15
5	2.87E-16	0.07769938357	0.6442176872	0.00E+00

If we do four argument reductions in step 4, we get the result after only three iterations

sin(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.008641975**	
Taylor reductions=		**4**		
Terms	**Term value**	**Term Sum**	**sin(x)**	**Error**

1	8.64E-03	0.00864197531	0.6442243516	-6.66E-06
2	1.08E-07	0.008641868	0.6442176872	2.49E-11
3	4.02E-13	0.00864186774	0.6442176872	0.00E+00

Again, we notice that using argument reduction can seriously cut down the number of Taylor terms needed and thereby increase the performance of calculating sin(x).

The issue with arbitrary precision for sin(x)

The Number of Taylor terms to reach a result does not seem so bad at a first glance. In the previous examples, we were only using approx. 15 decimal digits. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. In Yacas book of algorithms [5] they found a bound for the number of Taylor terms, n needed for the sin(x) as a function of the number of precision in digits P and the magnitude, M of the argument $x=10^M$:

$$2(n+1) \approx \frac{(P-M)\cdot\ln(10)}{\ln(P-M)-1-M\cdot\ln(10)} =>$$

$$n \approx \frac{1}{2}\frac{(P-M)\cdot\ln(10)}{\ln(P-M)-1-M\cdot\ln(10)} - 1 \qquad (73)$$

The number of Taylor terms needed for sin(x) as a function of precision and argument magnitude.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
x								
10^1	(11)	88	319	1,948	14,022	109,512	898,358	7,615,327
10^0	8	31	194	1,402	10,951	89,835	761,532	6,608,768
10^{-1}	3	19	140	1,095	8,983	76,153	660,876	5,837,230
10^{-2}	2	14	109	898	7,615	66,087	583,723	5,227,006
10^{-3}	1	11	90	761	6,608	58,372	522,700	4,732,291
10^{-4}	1	9	76	661	5,837	52,270	473,229	4,323,125
10^{-5}	1	7	66	584	5,227	47,323	432,312	3,979,084
10^{-6}	1	6	58	522	4,732	43,231	397,908	3,685,765
10^{-7}	1	6	52	473	4,323	39,791	368,576	3,432,721
10^{-8}	1	5	47	432	3,979	36,857	343,272	3,212,190
10^{-9}	(1)	5	43	398	3,686	34,327	321,219	3,018,284

The table above is quite interesting. E.g., the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of 1 in magnitude down to the argument of 10^{-9} in magnitude. For a precision of 100,000 digits, the factor is only around three and for 100M digits, it is around 2.2. The lesson here is that argument reduction is more efficient for smaller precision than for higher precision. However, overall argument reduction is beneficial at any precision. There is another approximation for n based on the actual value of x not just the magnitude. It usually gives a little bit less amount of needed Taylor terms. This formula can be quite useful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (74)$$

Finding a reasonable reductions factor for sin(x)

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? The argument reduction on the front end is a division per reduction. In the back end, you do this as many times as you did the reduction on the front end. $\sin(3x)=3\sin(x)-4(\sin^3(x))$ taking sin(x) out as a factor you get this: $\sin(3x)=\sin(x)(3-4(\sin^2(x)))$ or one subtraction and three multiplication. Using

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (75)$$

At a starting point of x=1, you get for P=1,00digits that the needed Taylor term is 24. Doing three reductions you get $x=1/3^3 = 0.037$. Using the above formula we expect we would only need 14 Taylor terms. Each Taylor term requires one addition/subtraction, 1 division, and multiplication which yield a total saving of 10 subtraction, 10 division, and 10 multiplication. Compared to three reductions on the front-end is three divisions and on the backend three subtraction and nine multiplication a total saving of seven subtraction/addition, one multiplication, and seven division. Since division is a magnitude slower than multiplication and addition/subtraction, we can give a rough saving equivalent with seven divisions. For higher precisions, the saving becomes larger.

We automatically calculate the reduction factor as $k = 8\left\lceil \frac{2}{3}\ln(2) * \ln(P) \right\rceil$ for higher precisions, and then we adjusted the magnitude of x. After Step 2, we know that x is in the range of $[0..\pi]$ this is equivalent that the exponent of our number (in base 2) being in the range $[-\infty..1]$. We add the exponent to the reduction factor. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply do not perform any argument reductions at all. E.g. for P=100 you get 24 and for P=10,000 you get 40. To compensate for the inaccuracy when adding the front and back end calculation, we increase the precision by a quarter of the k factor. The increased precision only generates a small performance penalty compared to the extra saving in Taylor's terms of the overall calculation.

Guard Digits for sin(x)

When summarizing a Taylor series as sin(x) you need quite a lot of summarizing and that will produce round-off errors.

For our sin (x) function, we use a simple guard digits calculation that we add

2+ceil(log10(precision)) as extra guard digits as the working precision.

Further Improvement of the methods for sin(x)?

There is not a lot of things you can do to improve the sin(x) algorithm. However, consider the Taylor series expansion of sin(x):

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (76)$$

The issue is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as

coefficient scaling) and reduce the number of divisions. Consider the n'th and the n+1 term assuming the n'th term is the negative part (for the moment):

$$\ldots - \frac{x^n}{n!} + \frac{x^{n+2}}{(n+2)!} \ldots$$

Moreover, group them:

$$\ldots \frac{-(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \ldots =>$$

$$\ldots \frac{-(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \ldots$$

If the n'th term is not the one starting with the minus sign you can simply just flip the sign in the above equation, yielding:

$$\ldots \frac{+(n+1)(n+2)x^n - x^{n+2}}{(n+2)!} \ldots$$

Then you have replaced one division for two multiplication. The (n+1)(n+2) can be done using a 32-bit or 64-bit integer since you never get to do so many Taylor terms in real life. There is no need to stop at just grouping two terms together you can do that for three terms or more:

$$\ldots \frac{-(n+1)(n+2)(n+3)(n+4)x^n + (n+3)(n+4)x^{n+2} - x^{n+4}}{(n+4)!} \ldots$$

Saving two divisions, however, gaining a few more addition and multiplications.

It is very easy to determine when we need to start with a negative sign by just testing if n'th term divided by 2 is an odd number (start with a minus sign) or an even number starting with plus sign and then alternative the sign thereafter.

Recommendation for calculating sin(x)

We can be based on the performance measure of the various sin() methods recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.\pi]$
- It is unnecessary to reduce it down to the range $[0\ldots\frac{\pi}{2}]$ using symmetry avoiding another calculation of π.
- Use Taylor for sin(x) using an aggressive reduction factor to speed up the Taylor term calculation.
- Use Coefficient scaling to increase performance

Cos(x) using Taylor series:

For cos(x) we again use a Taylor series until any additional addition does not change the result for the given precision of the number.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \ldots for\ any\ real\ value\ x \quad (77)$$

We can use the equivalent four steps produced for cos(x), mapping it into the interval $[0\ldots\frac{\pi}{2}]$.

Step 1: We notice that cos(x) is cyclic with a period of 2π so we can easily reduce any argument > 2π so it falls between 0 and 2π by simply taking x modulo 2π.

Step 2: We can further reduce x so it is between 0..π using the identity:
cos(2π-x)=cos(x) for x≥π.

Step 3: We reduce it further by using the symmetry around $\frac{\pi}{2}$ $to\ the\ range\ 0..\frac{\pi}{2}$:
$cos(x) = -cos\left(x - \frac{\pi}{2}\right) for\ \mathrm{x} \geq \frac{\pi}{2}$.
If π is 'expensive' to calculate (which is usually the case with arbitrary precision) we can omit step 3 since we have a different way to obtain the same thing by just increasing the argument reduction factor. See the section on finding a reasonable reduction factor.

Step 4: Finally we reduced the argument *k* number of times using the trisection identity:
$\cos(3x)=-3\cos(x)+4\cos^3(x)$
until x is below a certain threshold. It is obvious from the cos(x) Taylor series that the smaller x the fewer terms we would need. We could also use the double-angle identity:
$\cos(2x) = 2cos^2(x) - 1$
Although the trisection identity serves us well for calculating sin(x) it turns out that there is a much higher loss of precision using the trisection identity over the double angle formula. See later.

This argument reduction is done to reduce the number of Taylor iterations and to minimize the round-off errors and calculation time.

After the Taylor series has converged, we use the trisection or double angle identity reverse *k* number of times to find our result for cos(x).

Example 1. Cos(x) using the Taylor series

To see how this algorithm works let us find the cos(0.7). After the 8th Taylor term, the error is zero and the result is ~ 0.7648421873.

cos(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.7**	
Taylor reductions=		**0**		
Terms	**Term value**		**cos(x)**	**Error**
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01
2	2.45E-01	0.7550000000	0.7550000000	9.84E-03
3	1.00E-02	0.7650041667	0.7650041667	-1.62E-04
4	1.63E-04	0.7648407653	0.7648407653	1.42E-06
5	1.43E-06	0.7648421950	0.7648421950	-7.76E-09
6	7.78E-09	0.7648421873	0.7648421873	2.88E-11
7	2.89E-11	0.7648421873	0.7648421873	-7.76E-14
8	7.78E-14	0.7648421873	0.7648421873	0.00E+00

Example 2. Cos(x) using Taylor series and argument reduction

We can see the effect in Step 4 by increasing the number of argument reductions. E.g. for two reductions you get the same result after only five iterations. The argument is reduced twice from 0.7 to 0.077…

cos(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.077777778**	
Taylor reductions=		**2**		
Terms	**Term value**		**cos(x)**	**Error**
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01
2	3.02E-03	0.9969753086	0.7647284320	1.14E-04
3	1.52E-06	0.9969768334	0.7648422102	-2.29E-08
4	3.07E-10	0.9969768331	0.7648421873	2.48E-12
5	3.32E-14	0.9969768331	0.7648421873	2.78E-15

If we do four argument reductions in step 4, we get the result after only four iterations

cos(x)		**Original**	**X Reduced**	
x=		**0.7**	**0.008641975**	
Taylor reductions=		**4**		
Terms	**Term value**		**cos(x)**	**Error**
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01
2	3.73E-05	0.9999626581	0.7648407840	1.40E-06
3	2.32E-10	0.9999626584	0.7648421873	-3.63E-12
4	5.79E-16	0.9999626584	0.7648421873	-2.81E-13

Again, we notice that using argument reduction can seriously cut down the number of Taylor terms needed and thereby increase the performance in calculating cos(x).

We notice that the error has increased and we cannot find an answer better than an absolute error or ~1E-13. The higher the reduction factor the worse it gets. It has to be noticed that this issue arises only from the use of a reduction factor and not from the use of the Taylor series.

Although many of the same arguments used in the calculation of sin(x) also apply for cos(x), including aggressive use of argument reduction, coefficients scaling, etc. We have to be careful how aggressive our argument reduction can be.

Cos(x) using double angle reduction

Argument reduction reduces x to a much smaller value that is much more sensitive to round-off errors for cos(x) than its counterpart for sin(x). It is therefore better to use the double-angle formula:

$$\cos(2x) = 2cos^2(x) - 1 \tag{78}$$

Alternatively, even better written as:

$$\cos(2x) = 2(1 - \cos(x))^2 - 4(1 - \cos(x)) + 1 \tag{79}$$

Although it does not prevent round-off errors it is less sensitive that the trisection formula. We calculate the reduction factor for cos(x) as $k = 2\lceil \ln(2) * \ln(P) \rceil$ for higher precisions, and then we adjusted the magnitude of x. After Step 2, we know that x is in the range of [0..π] this is equivalent that the exponent of our number (in base 2) being in the range [-∞...1]. We add the exponent to the reduction factor. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply do not perform any argument reductions at all.

Cos(x) using sin(x)

Since we have a very fast and robust implementation of sin(x) that does not suffer from the same issue of using a high reduction factor compare to cos(x) it could be interesting to calculate cos(x) using sin(x):

$$\cos(x) = \sqrt{1 - sin^2(x)} \tag{80}$$

It turns out that this increases the performance by a factor of 2 times the traditional way of calculating cos(x) directly and is therefore recommended. There is another alternative to using the identity: $\cos(x) = \sin(\frac{\pi}{2} - x)$. If you have a fast generation π you will experience a similar performance as the $\cos(x) = \sqrt{1 - sin^2(x)}$ but in my opinion, it will be safer to rely on the faster sqrt(x) function.

Recommendation for calculating cos(x)

Based on the performance measure of the various cos(x) methods recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.\pi]$
- It is unnecessary to reduce it down to the range $[0..\frac{\pi}{2}]$ using symmetry avoiding another calculation of π.
- Use the double angle formula for argument reduction instead of the trisection formula.
- Do not use the Taylor series for cos(x) with an aggressive reductions factor to speed up the Taylor term calculation. If you do it anyway then use it with coefficient scaling to increase performance.
- Use $\cos(x) = \sqrt{1 - sin^2(x)}$ is recommended for calculating cos(x) which is two times faster than the other cos(x) methods.

Tan(x):

We could use a Taylor series for tan(x) however since we have an efficient implementation of sin(x) it is better to use the identity:

$$tan(x) = \frac{\sin(x)}{\sqrt{1 - si^2(x)}} \tag{81}$$

However, before we start the calculation we first reduce the argument x so it falls between 0 and 2π and then call Sin(x) (see above).

Alternatively, we could use the Taylor series for tan(x):

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^5}{315} + \frac{62x^9}{2835} + \cdots \frac{2^{2n}(2^{2n}-1)B_n x^{2n-1}}{(2n)!} + \cdots \tag{82}$$

Where B_n is the Bernoulli number. However, since we don't know how many Bernoulli numbers we need this will require it to be calculated on the fly and therefore way more complicated to implement than the identity for tan(x) using sin(x).

Arcsin(x):

We have a few options. Either we can find arcsin (x) using the Newton method or we can do it using a Taylor series for arcsin (x).

Arcsin using the Newton method

To find arcSin(x) it is very popular to resort to a Newton iteration when solving the equation arcSin(a)=x => a=sin(x).

Restating the problem as f(a)=sin(x)-a=0 and applying the Newton method we get:
Where f(x)=sin(x)-a and f'(x)=cos(x).

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\cos(x_n)} \quad (83)$$

We stop when $x_n=x_{n-1}$ for any given precision of the number. We do not want to calculate both sin(x) and cos(x) so we replace cos(x) with the identity:

$$\cos(x) = \sqrt{1 - sin^2(x)} \quad (84)$$

Yields:

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\sqrt{1-sin^2(x_n)}} \quad (85)$$

To speed up the iteration and to ensure convergence we repeatedly reduced the argument x, to a small value using the identity:

$$Arcsin(x) = 2 \cdot ArcSin(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}) \quad (86)$$

Now the x argument will always per definition between $-1 \leq x \geq 1$, so we will only need a maximum of two argument reductions to get below 0.5.

You can obtain k, numbers of reduction by repeatedly doing below recurrence k number of times. Set $x_0=x$ and k is the number of reductions:

$$x_k = \frac{x_{k-1}}{\sqrt{2}\sqrt{1+\sqrt{1-x_{k-1}^2}}} \quad (87)$$

Until x_m is sufficiently low. Now we can start with an initial guess of arcsin(x) using standard IEEE754. This gives us a starting guess for the Newton iteration with a least 15 significant digits and the Newton iteration will converge quickly with a convergence rate of 2 meaning the number of correct digits doubles per iteration. After we find the new x_n we will need to multiply the result with $x = x_n \cdot 2^k$ to reverse the argument reduction we did before the Newton iteration.

Example 1. ArcSin(x) using the Taylor series

To see how this algorithm works let us find the arcSin(0.3). After only three iterations the error is zero and the result is ~ 0.304693.

ArcSin(x)	Newton	Original	X Reduced
x=		0.3	0.3
No Reduction		0	
Iteration	x	ArcSin(x)	Error
1	0.304689231	0.304689230851802	3.42E-06
2	0.304692654	0.304692654013555	1.84E-12
3	0.304692654	0.304692654015398	0.00E+00

Now assuming for a moment we did not do any argument reduction we will see a much slower convergence when x get near 1. See below.

ArcSin(x)	Newton	Original	X Reduced
x=		1	1
No Reduction		0	
Iteration	x	ArcSin(x)	Error
1	1.293407993	1.293407993026020	2.77E-01
2	1.432998367	1.432998366665080	1.38E-01
3	1.502006577	1.502006576891840	6.88E-02
4	1.536415021	1.536415021395350	3.44E-02
5	1.553607368	1.553607367680850	1.72E-02
6	1.562202059	1.562202058854760	8.59E-03
7	1.566499219	1.566499219274400	4.30E-03
8	1.568647776	1.568647776340770	2.15E-03
9	1.569722052	1.569722051981120	1.07E-03
10	1.570259189	1.570259189439680	5.37E-04
11	1.570527758	1.570527758123650	2.69E-04
12	1.570662042	1.570662042459940	1.34E-04
13	1.570729185	1.570729184627400	6.71E-05
14	1.570762756	1.570762755710630	3.36E-05
15	1.570779541	1.570779541251150	1.68E-05
16	1.570787934	1.570787934020610	8.39E-06
17	1.57079213	1.570792130414050	4.20E-06
18	1.570794229	1.570794228613920	2.10E-06
19	1.570795278	1.570795277678890	1.05E-06
20	1.570795802	1.570795802251530	5.25E-07
21	1.570796064	1.570796064492250	2.62E-07
22	1.570796196	1.570796195702950	1.31E-07
23	1.570796261	1.570796260914560	6.59E-08
24	1.570796295	1.570796294618790	3.22E-08
25	1.570796312	1.570796311871080	1.49E-08
26	1.570796319	1.570796319310360	7.48E-09

Even after 26 iterations, we only get a decent result with an error margin of 7.48E-9, while with two argument reductions, we have the result with only three iterations.

Example 2. ArcSin(x) using Taylor series and argument reduction

ArcSin(x)	Newton	Original	X Reduced
x=		1	0.382683432
No Reduction		2	
Iteration	**x**	**ArcSin(x)**	**Error**
1	0.392678725	1.570714899985370	2.04E-05
2	0.392699082	1.570796326451610	8.58E-11
3	0.392699082	1.570796326794900	0.00E+00

This example demonstrates the benefit of using argument reduction before applying the Newton iterations.

Using Newton's iteration gives the result in relatively few iterations however still not very fast compared to the direct approach using the Taylor series, see next section.

Arcsin(x) using Taylor series and argument reduction

Instead of the Newton method, we can use the Taylor Series for arcsin(x) given by:

$$Arcsin(x) = x + \frac{x^3}{2\cdot 3} + \frac{3x^5}{2\cdot 4\cdot 5} + \frac{3\cdot 5x^7}{2\cdot 4\cdot 6\cdot 7} + \frac{3\cdot 5\cdot 7x^9}{2\cdot 4\cdot 6\cdot 8\cdot 9} \ldots \quad (88)$$

This gives us a more direct approach to arcsin(x) and applied together with the dynamic argument reductions we see a speed up in the calculation in the range of two. As the precision rise, this method will become increasingly faster than the Newton version.

The Taylor series seems a little hard to digest. If we denote the n'th Taylor term, r we can go from one Taylor term to the next using the following recurrence:

$$r_1 = x$$

$$r_n = r_{n-1}\frac{(2n-3)^2 \cdot x^2}{(2n-1)(2n-2)} \; for\; n = 2,3,\ldots,m$$

We calculate the reducing factor, k as $2 \cdot \lceil \ln(2) * \ln(precision) \rceil$ and adjust the reduction factor downwards if x is small to avoid unnecessary reductions. We should be careful not to be too aggressive because of the reduction of identity:

$$Arcsin(x) = 2 \cdot ArcSin\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (89)$$

Require one division and two square roots ($\sqrt{2}$ is a constant that can be calculated before the reduction), two multiplication, and two addition/subtracting. The benefit of using reduction slows down and becomes counterproductive when the reduction factor exceeds 30-40.

Example 3. ArcSin(x) using the Taylor series

Below is an example of using the Taylor Series for calculating arcSin(x) with x=0.3.

ArcSin(x)	Taylor	Original	X Reduced

x=		0.3	0.3	
No Reduction		0		
Terms	Term Value	Taylor sum	Arcsin(x)	Error
1	3.00E-01	0.300000000000000	0.300000000000000	4.69E-03
2	4.50E-03	0.304500000000000	0.304500000000000	1.93E-04
3	1.82E-04	0.304682250000000	0.304682250000000	1.04E-05
4	9.76E-06	0.304692013392857	0.304692013392857	6.41E-07
5	5.98E-07	0.304692611400670	0.304692611400670	4.26E-08
6	3.96E-08	0.304692651032278	0.304692651032278	2.98E-09
7	2.77E-09	0.304692653798869	0.304692653798869	2.17E-10
8	2.00E-10	0.304692653999250	0.304692653999250	1.61E-11
9	1.49E-11	0.304692654014168	0.304692654014168	1.23E-12
10	1.13E-12	0.304692654015302	0.304692654015302	9.53E-14
11	8.78E-14	0.304692654015390	0.304692654015390	7.55E-15
12	6.88E-15	0.304692654015397	0.304692654015397	6.66E-16

After 12 Taylor terms, we have the result with 15-16 decimal digits. If we run it with a reduction factor of, two we get:

Example 4. Sin(x) using Taylor series and argument reduction

ArcSin(x)	Taylor	Original	X Reduced	
x=		0.3	0.076099521	
No Reduction		2		
Terms	Term Value	Taylor sum	Arcsin(x)	Error
1	7.61E-02	0.076099520968904	0.304398083875615	2.95E-04
2	7.35E-05	0.076172971428661	0.304691885714644	7.68E-07
3	1.91E-07	0.076173162841418	0.304692651365671	2.65E-09
4	6.60E-10	0.076173163501238	0.304692654004951	1.04E-11
5	2.60E-12	0.076173163503838	0.304692654015353	4.47E-14
6	1.11E-14	0.076173163503849	0.304692654015397	3.89E-16

The same result is achieved after only six iterations. This again demonstrates that argument reduction can reduce the workload significantly.

Arcsin with coefficient scaling

We have seen that we can gain typically 2-3 times better performance if we implement coefficient scaling. If we try to group two Taylor terms to avoid a division, we get from the Taylor terms listed above:

$$r_1 = x, \quad r_n = r_{n-1}\frac{(2n-3)^2 \cdot x^2}{(2n-1)(2n-2)} \; for\; n = 2,3,\dots,m$$

If we denoted for simplicity $u_1 = (2n-3)^2, l_1 = (2n-1)(2n-2)$ and the following term u_2 and l_2 we get from the above recurrence when grouping two terms together:

$$Two\ Taylor\ terms = r_{n-1}\frac{u_1 \cdot x^2}{l_1} + r_{n-1}\frac{u_1 \cdot x^2}{l_1}\cdot\frac{u_2 \cdot x^2}{l_2} =>$$

$$r_{n-1}x^2\left(\frac{u_1}{l_1}+\frac{u_1}{l_1}\cdot\frac{u_2\cdot x^2}{l_2}\right) =>$$

$$r_{n-1}x^2\left(\frac{u_1l_2}{l_1l_2}+\frac{u_1u_2\cdot x^2}{l_1l_2}\right) =>$$

$$r_{n-1}x^2\left(\frac{u_1l_2+u_1u_2\cdot x^2}{l_1l_2}\right)$$

The new recurrence for r, grouping two Taylor terms together is given by:

$$r_1 = x, \quad r_{n+1} = r_{n-1}x^4\frac{u_1u_2}{l_1l_2}$$

Continue one by grouping three Taylor terms together you get.

$$r_{n-1}x^2\left(\frac{u_1l_2l_3+u_1u_2l_3\cdot x^2+u_1u_2u_3\cdot x^4}{l_1l_2l_3}\right)$$

The new r_{n+2} is given by:

$$r_1 = x, \quad r_{n+2} = r_{n-1}x^6\frac{u_1u_2u_3}{l_1l_2l_3}$$

You can continue on this path. In the current implementation, we use a grouping of five Taylor terms and scale the coefficients accordingly.

Recommendation for calculating Arcsin(x)

Based on the performance measure of the various arcsin(x) methods recommend:

- The preferred method is to use the Taylor series for arcsin(), together with argument reduction and coefficient scaling.
- Arcsin() using the Newton method does not perform as well as the Taylor series method. The performance issue gets worse with increasing precision.
- Only use a moderate number of argument reductions since it is very time-consuming to calculate. (Involving a division and two square roots calculation).

Arccos(x):

To find Arccos(x) we used the identity:

$$Arccos(x) = \frac{\pi}{2} - Arcsin(x) \qquad (90)$$

It is not much else you can do.

Arctan(x):

There are two interesting methods to use. One is the standard Taylor series and the other one is contributed to Euler which is considered faster than the Taylor series (at least fewer terms are needed).

Arctan(x) using the Taylor series

For arctan(x) we can use a Taylor series until any additional addition does not change the result for the given precision of the number:

$$Arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots, where\ |x| \leq 1 \quad (91)$$

However, before we start the Taylor series we first need to reduce the argument x to a smaller value that will make the Taylor series run faster by using fewer Taylor terms. We use the identity:

$$Arctan(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (92)$$

k number of times until x is sufficiently low.

This argument reduction is done to reduce the number of Taylor steps and minimize the round-off errors and calculation time and of course, ensure that our Taylor series is stable.

We calculate the reducing factor, k as $2 \cdot \lceil \ln(2) * \ln(precision) \rceil$ and adjust the reduction factor downwards if x is small to avoid unnecessary reductions. We should be careful not to be too aggressive because of the reduction of identity:

$$Arctan(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (93)$$

Require one division and one square root, two addition. The benefit of using reduction slows down and becomes counterproductive when the reduction factor exceeds 30-40.

After the Taylor series has converged, we multiply the result with 2^k to find our result for arctan(x). Now looking closer at the argument reduction, you will notice that we never need more than one argument reduction to reduce x>1 to x<1. The first reduction will give us a max of ±1 since:

$$\lim_{x\to\infty}\left(\frac{x}{1+\sqrt{1+x^2}}\right) = 1 \quad (94)$$

or

$$\lim_{x\to-\infty}\left(\frac{x}{1+\sqrt{1+x^2}}\right) = -1 \quad (95)$$

Example 1. ArcTan(x) using the Taylor series

To see how this algorithm works let us find the arctan(0.3). After the 13th Taylor Terms the errors do not get lower and the result is ~ 0.291456794477867.

ArcTan(x)	Taylor	Original	X Reduced

x=		0.3	0.3	
No Reduction		0		
Terms	Term Value	Taylor sum	Arctan(x)	Error
1	3.00E-01	0.300000000000000	0.300000000000000	8.54E-03
2	9.00E-03	0.291000000000000	0.291000000000000	4.57E-04
3	4.86E-04	0.291486000000000	0.291486000000000	2.92E-05
4	3.12E-05	0.291454757142857	0.291454757142857	2.04E-06
5	2.19E-06	0.291456944142857	0.291456944142857	1.50E-07
6	1.61E-07	0.291456783100130	0.291456783100130	1.14E-08
7	1.23E-08	0.291456795364153	0.291456795364153	8.86E-10
8	9.57E-10	0.291456794407559	0.291456794407559	7.03E-11
9	7.60E-11	0.291456794483524	0.291456794483524	5.66E-12
10	6.12E-12	0.291456794477407	0.291456794477407	4.60E-13
11	4.98E-13	0.291456794477905	0.291456794477905	3.77E-14
12	4.09E-14	0.291456794477864	0.291456794477864	3.16E-15
13	3.39E-15	0.291456794477867	0.291456794477867	2.22E-16

Example 2. ArcTan(x) using Taylor series and argument reduction

Now if we take two-argument reduction we reduced the number of Taylor terms taken. E.g., arctan(0.3) gives the result after only six Taylor terms.

ArcTan(x)	Taylor	Original	X Reduced	
x=		0.3	0.072993423	
No Reduction		2		
Terms	Term Value	Taylor sum	Arctan(x)	Error
1	7.30E-02	0.072993423050513	0.291973692202050	5.17E-04
2	1.30E-04	0.072863785762585	0.291455143050342	1.65E-06
3	4.14E-07	0.072864200190164	0.291456800760656	6.28E-09
4	1.58E-09	0.072864198612959	0.291456794451837	2.60E-11
5	6.54E-12	0.072864198619495	0.291456794477981	1.13E-13
6	2.85E-14	0.072864198619467	0.291456794477867	5.55E-16

If we do four argument reductions, we only need four Taylor terms to get the result. As we have seen before, argument reduction is crucial to lowering the number of Taylor terms needed when precision is increased.

The issue with arbitrary precision for Arctan

The Number of Taylor terms to reach a result does not seem so bad at a first glance. In the previous examples, we were only using approx. 15 decimal digits. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. You can find the approximate value for the number of Taylor Terms *n* by:

$$\frac{x^{2n-1}}{2n-1} < 10^{-P} \tag{96}$$

Where P is the precision in decimal digits and $|x| < 1$. The terms we dropped are the 2^{n+1} terms. Given

$$\frac{x^{2n+1}}{2n+1} = 10^{-P} =>$$

$$(2n+1)\ln(x) - \ln(2n+1) = -P \cdot \ln(10)$$

-ln(2n+1) is small compare to (2n+1)ln(x) so we drop it and get:

$$(2n+1)\ln(x) \approx -P \cdot \ln(10) =>$$

$$(2n+1) \approx \frac{-P \cdot \ln(10)}{\ln(x)} =>$$

$$n \approx \frac{-P \cdot \ln(10) - \ln(x)}{2 \cdot \ln(x)} \quad (97)$$

Now if we use $x=10^M$ where M is the magnitude of the number we can further simplify it:

$$n \approx \frac{-P-M}{2 \cdot M} \quad (98)$$

The number of Taylor terms needed for arctan(x) as a function of precision and argument magnitude.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
x								
10^{-1}	5	50	500	5,000	50,000	500,000	5,000,000	50,000,000
10^{-2}	2	25	250	2,500	25,000	250,000	2,500,000	25,000,000
10^{-3}	1	16	166	1,666	16,666	166,666	1,666,666	16,666,666
10^{-4}	1	12	125	1,250	12,500	125,000	1,250,000	12,500,000
10^{-5}	1	10	100	1,000	10,000	100,000	1,000,000	10,000,000
10^{-6}	0	8	83	833	8,333	83,333	833,333	8,333,333
10^{-7}	0	7	71	714	7,142	71,428	714,285	7,142,857
10^{-8}	0	6	62	625	6,250	62,500	625,000	6,250,000
10^{-9}	0	5	55	555	5,555	55,555	555,555	5,555,555

This table indicates the usefulness of argument reduction.

The table above is quite interesting. E.g., the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of -1 in magnitude down to an argument of 10^{-9} in magnitude is around a factor of 10 times fewer Taylor Terms. However overall argument reduction is beneficial at any precision.

Arctan(x) using coefficient scaling

We have seen that we can gain typically 2-3 times better performance if we implement coefficient scaling. If we try to group two Taylor terms to avoid a division, we get from the Taylor series for arctan where *n* denoted the n'th Taylor term for arctan if term *n* is even we start with a minus sign otherwise +, and then we alternate the sign for each Taylor term going forward:

$$Two\ Taylor\ terms:\ -\frac{x^{n-1}}{2n-1}+\frac{x^{n+1}}{2n+1} =>$$

$$-\frac{(2n+1)x^{n-1}+(2n-1)x^{n+1}}{(2n-1)(2n+1)} =>$$

$$x^{n-1}\cdot\frac{-(2n+1)+(2n-1)x^2}{(2n-1)(2n+1)}$$

If we group three Taylor terms, we get:

$$x^{n-1}\cdot\frac{-(2n+1)(2n+3)+(2n-1)(2n+3)x^2-(2n-1)(2n+1)x^4}{(2n-1)(2n+1)(2n+3)}$$

We can continue grouping Taylor terms. From a practical point of view, grouping five Taylor terms together is a reasonable amount as it will double the performance compared to not doing it.

Arctan(x) using the Euler method

Euler devised another series for arctan that supposedly converges more quickly than the Taylor series. The series can be expressed (alternatively) as:

$$Arctan(x)=\sum_{n=0}^{\infty}\frac{2^{2n}(n!)^2}{(2n+1)!}\frac{z^{2n+1}}{(1+x^2)^{n+1}} \quad (99)$$

For x>0.4 required fewer Terms than the equivalent Taylor series, e.g. arctan(0.6) requires 25 terms to get the result. While using the Taylor series requires 30 Taylor terms. As x increased, get worse. However, for x<0.4 the Taylor series and the Euler series require approximately the same number of terms.

Example 1. ArcTan(x) using Euler's method

ArcTan(x)	Euler	Original	X Reduced	
x=		0.6	0.6	
No Reduction		0		
Terms	**Term Value**	**Euler sum**	**Arctan(x)**	**Error**
1	4.41E-01	0.441176470588235	0.441176470588235	9.92E-02
2	7.79E-02	0.519031141868512	0.519031141868512	2.14E-02
3	1.65E-02	0.535518013433747	0.535518013433747	4.90E-03
4	3.74E-03	0.539258732192246	0.539258732192246	1.16E-03
5	8.80E-04	0.540138901311893	0.540138901311893	2.81E-04
6	2.12E-04	0.540350706715016	0.540350706715016	6.88E-05
7	5.18E-05	0.540402460071436	0.540402460071436	1.70E-05
8	1.28E-05	0.540415246194786	0.540415246194786	4.25E-06
9	3.19E-06	0.540418431664964	0.540418431664964	1.07E-06
10	7.99E-07	0.540419230498042	0.540419230498042	2.70E-07
11	2.01E-07	0.540419431884533	0.540419431884533	6.84E-08
12	5.10E-08	0.540419482874974	0.540419482874974	1.74E-08
13	1.30E-08	0.540419495832545	0.540419495832545	4.44E-09

14	3.30E-09	0.540419499135455	0.540419499135455	1.14E-09
15	8.44E-10	0.540419499979607	0.540419499979607	2.91E-10
16	2.16E-10	0.540419500195850	0.540419500195850	7.47E-11
17	5.55E-11	0.540419500251357	0.540419500251357	1.92E-11
18	1.43E-11	0.540419500265630	0.540419500265630	4.95E-12
19	3.68E-12	0.540419500269306	0.540419500269306	1.28E-12
20	9.48E-13	0.540419500270254	0.540419500270254	3.30E-13
21	2.45E-13	0.540419500270499	0.540419500270499	8.54E-14
22	6.33E-14	0.540419500270562	0.540419500270562	2.21E-14
23	1.64E-14	0.540419500270579	0.540419500270579	5.66E-15
24	4.24E-15	0.540419500270583	0.540419500270583	1.44E-15
25	1.10E-15	0.540419500270584	0.540419500270584	3.33E-16

Example 2. ArcTan(x) using Euler's method and argument reduction

As with the Taylor series using argument reduction greatly reduced the number of terms needed. E.g. arctan(0.6) using a reduction factor of four requires only 5 terms (same as for the Taylor series).

ArcTan(x)	**Euler**	**Original**	**X Reduced**	
x=		**0.6**	**0.033789069**	
No Reduction		**4**		
Terms	**Term Value**	**Euler sum**	**Arctan(x)**	**Error**
1	3.38E-02	0.033750535945282	0.540008575124517	4.11E-04
2	2.57E-05	0.033776195334449	0.540419125351177	3.75E-07
3	2.34E-08	0.033776218744006	0.540419499904093	3.66E-10
4	2.29E-11	0.033776218766888	0.540419500270213	3.71E-13
5	2.32E-14	0.033776218766912	0.540419500270584	3.33E-16

Another drawback is that each Euler term requires more computational power than the corresponding Taylor series. Overall it is not worth using the Euler version of arctan(x) over the Taylor series version.

Arctan(x) using Arcsin()

It could be interesting to use the identity:

$$Arctan(x) = \text{Arcsin}\left(\frac{x}{\sqrt{1+x^2}}\right) \qquad (100)$$

Particularly if you want to reduce the size of your code and reuse existing code for arcsin(). However, the performance is slightly slower (20%-30%) than using the Taylor series for arctan().

Recommendation for calculating Arctan(x)

Based on the performance measure of the various arctan(x) methods recommend:

- The preferred method is to use the Taylor series for arctan(x), together with argument reduction and coefficient scaling.
- Arctan(x) using the Euler series has no advantages over the Taylor series for argument < 0.4. For argument x>0.4, it is more beneficial to stick with the Taylor

series and use the recommended argument reduction and coefficient scaling for increased performance.

- Arctan(x) using arcsin(x) is an alternative that is slower but can be used to simplify and reduce code size.
- Only use a moderate number of argument reductions since it is very time-consuming to calculate. (Involving a division and a square root calculation).

Hyperbolic functions:

Usually, you use a Taylor series to calculate the Hyperbolic functions for sinh(x) and cosh(x) and some simple hyperbolic identity to calculate tanh(x), arcsinh(x), arccosh(x), and arctanh(x). This chapter will examine:

- Sinh(x) using exp(x)
- Sinh(x) using Taylor series, argument reduction, and coefficient scaling.
- Cosh(x) using Taylor series, argument reduction, and coefficient scaling.
- Tanh(x) using a simple identity.
- Arcsinh(x) using a simple identity.
- Arccosh(x) using a simple identity.
- Arctanh(x) using a simple identity.

The most common one for arbitrary precision libraries is the standard Taylor series expansion method.

Sinh(x) using Exp(x)

It is tempting to use the definition of sinh(x):

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right) \quad (101)$$

Were we only need to calculate e^x once. Particularly if you have a fast implementation of e^x, you can use the above to calculate sinh(x) and save some code. However, recall ref [9] where the recommended method for calculating e^x is to use the sine hyperbolic function:

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (102)$$

If you have implemented the above method for exp(x) then you will experience a little bit slower performance using exp(x) to calculate sinh(x). Usually, sinh(x) is faster to calculate than exp(x).

Sinh(x) using the Taylor series

We have already seen that using the sinh(x) Taylor series for calculating e^x is faster than the e^x using the Taylor series. See ref [9]. We will repeat the finding from ref [9] below.

Sinh(x) is found with the Taylor series:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (103)$$

Example sinh(1):

Calculating sinh(1) using no argument reduction. We need seven Taylor terms to get the result using the Taylor series.

Sinh(x)	Original	X Reduced

x=		1	1	
Taylor reductions=		0		
Terms	Term value	Term Sum	Sinh(x)	Error
1	1.00E+00	1.00000000000	1.0000000000	1.75E-01
2	1.67E-01	1.16666666667	1.1666666667	8.53E-03
3	8.33E-03	1.17500000000	1.1750000000	2.01E-04
4	1.98E-04	1.17519841270	1.1751984127	2.78E-06
5	2.76E-06	1.17520116843	1.1752011684	2.52E-08
6	2.51E-08	1.17520119348	1.1752011935	1.61E-10
7	1.61E-10	1.17520119364	1.1752011936	7.67E-13

The issue with arbitrary precision for sinh(x)

The number of Taylor terms to reach a result does not seem too bad at a first glance. In the previous examples, we were only using approx. 15 decimal digits. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. In Yaca's book of algorithms [5] they found a bound for the number of Taylor terms, n needed for the sin(x) as a function of the number of precision in digits P and the magnitude, M of the argument $x=10^M$. You can use the same rationale as they used for sin(x) to get a bound for the number of Taylor terms for sinh(x):

$$2(n+1) \approx \frac{(P-M)\cdot \ln(10)}{\ln(P-M)-1-M\cdot\ln(10)} =>$$

$$n \approx \frac{1}{2}\frac{(P-M)\cdot\ln(10)}{\ln(P-M)-1-M\cdot\ln(10)} - 1 \qquad (104)$$

The number of Taylor terms needed for sinh(x) as a function of precision and argument magnitude.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
x								
$\mathbf{10^1}$	(11)	88	319	1,948	14,022	109,512	898,358	7,615,327
$\mathbf{10^0}$	8	31	194	1,402	10,951	89,835	761,532	6,608,768
$\mathbf{10^{-1}}$	3	19	140	1,095	8,983	76,153	660,876	5,837,230
$\mathbf{10^{-2}}$	2	14	109	898	7,615	66,087	583,723	5,227,006
$\mathbf{10^{-3}}$	1	11	90	761	6,608	58,372	522,700	4,732,291
$\mathbf{10^{-4}}$	1	9	76	661	5,837	52,270	473,229	4,323,125
$\mathbf{10^{-5}}$	1	7	66	584	5,227	47,323	432,312	3,979,084
$\mathbf{10^{-6}}$	1	6	58	522	4,732	43,231	397,908	3,685,765
$\mathbf{10^{-7}}$	1	6	52	473	4,323	39,791	368,576	3,432,721
$\mathbf{10^{-8}}$	1	5	47	432	3,979	36,857	343,272	3,212,190
$\mathbf{10^{-9}}$	(1)	5	43	398	3,686	34,327	321,219	3,018,284

The table above is quite interesting. E.g., the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of 1 in magnitude down to an argument of 10^{-9} in magnitude. For a precision of 100,000 digits, the

factor is only around three and for 100M digits, it is around 2.2. The lesson here is that argument reduction is more efficient for smaller precision than for higher precision. However overall argument reduction is beneficial at any precision. There is another approximation for *n* based on the actual value of x not just the magnitude. It usually gives a little bit less amount of needed Taylor terms. This formula can be quite useful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \qquad (105)$$

Argument Reduction for sinh(x)

It is clear looking at the Taylor series for sinh(x) that we prefer to have our |x| < 1 to ensure that the Taylor series converge more quickly. As we have seen before we can use *argument reduction* to work with a smaller number to get a faster converging of sinh(x) using fewer *Taylor terms* of the Taylor series.

We can use the trisection identity: $\sinh(3x) = \sinh(x)(3 + 4\sinh^2(x))$ to reduce the argument with a factor of three and then after the Taylor iterations we restore and find the correct value for sinh(x) by applying this formula the same number of times we did when reducing the argument.

Example – Two-argument reduction:

Using the same example as before for sinh(1) and using two argument reductions, you get the result after only four Taylor terms compare to seven with no argument reductions.

Sinh(x)		**Original**	**X Reduced**	
x=		**1**	**0.111111111**	
Taylor reductions=		**2**		
Terms	**Term value**	**Term Sum**	**Sinh(x)**	**Error**
1	1.11E-01	0.11111111111	1.1720460995	3.16E-03
2	2.29E-04	0.11133973480	1.1751992452	1.95E-06
3	1.41E-07	0.11133987592	1.1752011931	5.73E-10
4	4.15E-11	0.11133987596	1.1752011936	9.84E-14

Example – Eight-argument reductions:

With 8 times argument reduction, you get the result after two Taylor terms compare to four using two argument reductions.

Sinh(x)		**Original**	**X Reduced**	
x=		**1**	**0.000152416**	
Taylor reductions=		**8**		
Terms	**Term value**	**Term Sum**	**Sinh(x)**	**Error**
1	1.52E-04	0.00015241579	1.1752011877	5.97E-09
2	5.90E-13	0.00015241579	1.1752011936	0.00E+00

As of no surprise, using argument reduction greatly reduced the number of Taylor terms needed and will result in faster performance.

Finding a reasonable argument reductions factor for sinh(x)

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? The argument reduction on the front end is a division per reduction. In the back end, you do this as many times as you did the reductions on the front end. $\text{Sinh}(3x)=3\text{Sinh}(x)-4(\sinh^3(x))$ taking sinh(x) out as a factor you get this: $\text{Sinh}(3x)=\text{Sinh}(x)(3-4(\sinh^2(x)))$ or one subtraction and three multiplication. Using

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P)-\ln(x))} - 1 \qquad (106)$$

At a starting point of x=1, you get for P=1,00digits that the needed Taylor terms are 24. Doing three reductions you get $x=1/3^3 = 0.037$. Using the above formula we expect we would only need 14 Taylor terms. Each Taylor term requires one addition/subtraction, 1 division, and one multiplication which yields a total saving of 10 subtraction, 10 division, and 10 multiplication. Compared to three reductions on the front-end is three divisions and on the backend 3 subtraction and nine multiplication a total saving of seven subtraction/addition, one multiplication, and seven division. Since division is a magnitude slower than multiplication and addition/subtraction, we can give a rough saving equivalent with seven divisions. For higher precisions, the saving becomes larger.

We automatically calculate the reduction factor as $k = 8\left\lceil \frac{2}{3}\ln(2) * \ln(P) \right\rceil$ for higher precisions, and then we adjusted the magnitude of x. We add the exponent to the reduction factor. If x is large then we do more argument reductions and if x is small, we reduced the number of reductions. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply do not perform any argument reductions at all. E.g. for P=100 you get 24 and for P=10,000 you get 40. To compensate for the inaccuracy when adding the front and back end calculation, we increase the precision by the reduction factor, k/4. The increased precision only generates a small performance penalty compared to the extra saving in Taylor's terms of the overall calculation.

Now to calculate a reasonable reductions factor we make it a function of the wanted precision and the magnitude of the argument x. E.g. argument reduction increased as a log function of the wanted precision and argument reduction increased with a large magnitude of the number and decreased for a smaller magnitude of the argument x.

Guard Digits for sinh(x)

When summarizing a Taylor series as sinh(x) you need quite a lot of summarizing and that will produce round-off errors.

For our sinh(x) function, we use a simple guard digits calculation that we add

2+ceil(log10(precision)) as extra guard digits as the working precision.

Further improvements of the method for sinh(x)?

The same technique for coefficient scaling (grouping of Taylor terms) can be applied here as well. Consider the Taylor series for sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (107)$$

The issue again clearly is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient scaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\ldots \frac{x^n}{n!} + \frac{x^{n+2}}{(n+2)!} \ldots$$

Moreover, group them:

$$\ldots \frac{(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \ldots =>$$

$$\ldots \frac{(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \ldots$$

Then you have replaced one division with two extra multiplication. The (n+1)(n+2) can be done using 64-bit integer arithmetic since you never get to do so many Taylor terms in real life that it will overflow. There is no need to stop at just grouping two terms together you can do that for three terms:

For grouping three Taylor terms, you get:

$$\ldots \frac{(n+1)(n+2)(n+3)(n+4)x^n + (n+3)(n+4)x^{n+2} + x^{n+4}}{(n+4)!} \ldots =>$$

$$\ldots \frac{(n+3)(n+4)(\,(n+1)(n+2)x^n + x^{n+2}) + x^{n+4}}{(n+4)!} \ldots$$

Alternatively, even higher.

Recommendation for calculating sinh(x)

Based on the performance measure of the various sinh(x) methods recommend:

- Use standard Taylor series for sinh(x)
- Use an aggressive reductions factor to speed up the Taylor terms calculation.
- Use coefficient scaling to increase performance.

Cosh(x) using Exp(x)

It is tempting to use the definition of cosh(x):

$$\cosh(x) = \frac{1}{2}(e^x + e^{-x}) = \frac{1}{2}\left(e^x + \frac{1}{e^x}\right) \quad (108)$$

Were only need to calculate e^x once. Particularly if you have a fast implementation of e^x, you can use the above to calculate cosh(x) and save some code. However, the Taylor series for cosh(x) is faster to calculate than using the Taylor series for exp(x).

Cosh(x) using the Taylor series

For cosh(x), we again use a Taylor series until any additional addition does not change the result for the given precision of the number.

$$\cos h(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} \ldots \text{for any real value x} \qquad (109)$$

Example cosh(1):

Calculating cosh(1) using no argument reduction. We need nine Taylor terms to get the result using the Taylor series.

Cosh(x)		**Original**	**X Reduced**	
x=		**1**	**1**	
Taylor reductions=		**0**		
Terms	**Term value**	**Taylor sum**	**Cosh(x)**	**Error**
1	1.00E+00	1	1.00000000000000	5.43E-01
2	5.00E-01	1.5	1.50000000000000	4.31E-02
3	4.17E-02	1.54166667	1.54166666666667	1.41E-03
4	1.39E-03	1.54305556	1.54305555555556	2.51E-05
5	2.48E-05	1.54308036	1.54308035714286	2.78E-07
6	2.76E-07	1.54308063	1.54308063271605	2.10E-09
7	2.09E-09	1.54308063	1.54308063480373	1.15E-11
8	1.15E-11	1.54308063	1.54308063481520	4.77E-14
9	4.78E-14	1.54308063	1.54308063481524	0.00E+00

The vast majority of the issues arising in arbitrary precision for sinh(x) also apply to cosh(x).

Argument Reduction for cosh(x)

It is clear looking at the Taylor series for cosh(x) that we prefer to have our $|x| < 1$ to ensure that the Taylor series converge more quickly. As we have seen before we can use *argument reduction* to work with a smaller number to get a faster converging of cosh(x) using fewer *Taylor terms* of the Taylor series.

We can use the trisection identity: $\cosh(3x) = \cosh(x)(4cosh^2(x) - 3)$ to reduce the argument with a factor of three and then after the Taylor iterations we restore and find the correct value for cosh(x) by applying this formula the same number of times we did when reducing the argument.

Example – Two-argument reduction:

Using the same example as before for cosh(1) and using two argument reductions, you get the result after only five Taylor terms compare to nine with no argument reductions.

Cosh(x)	**Original**	**X Reduced**
x=	**1**	**0.111111111**
Taylor reductions=	**2**	

Terms	Term value	Taylor sum	Cosh(x)	Error
1	1.00E+00	1	1.00000000000000	5.43E-01
2	6.17E-03	1.00617284	1.54247714909996	6.03E-04
3	6.35E-06	1.00617919	1.54308038649498	2.48E-07
4	2.61E-09	1.00617919	1.54308063476051	5.47E-11
5	5.76E-13	1.00617919	1.54308063481524	2.00E-15

Example – Eight-argument reductions:

With 8 times argument reduction, you get the result after two Taylor terms compare to five using two argument reductions. However, the error is considerably higher (less accurate) than the equivalent calculation for sinh(1).

Cosh(x)		Original	X Reduced	
x=		1	0.000152416	
Taylor reductions=		8		
Terms	**Term value**	**Taylor sum**	**Cosh(x)**	**Error**
1	1.00E+00	1	1.00000000000000	5.43E-01
2	1.16E-08	1.00000001	1.54308063305537	1.76E-09

As of no surprise, using argument reduction greatly reduced the number of Taylor terms needed and will result in faster performance. However aggressive reductions of argument result in a significant reduction in accuracy. This is due to the trisection identity and the Taylor sum is approaching one for a very small argument resulting in a higher loss of accuracy unless you take precautions. To avoid inaccuracy in the result we increase the precision with the reduction, k (instead of k/4 as for sinh(x)).

Cosh(x) using double angle reduction

Argument reduction reduces x to a much smaller value that is much more sensitive to round-off errors for cosh(x) than its counterpart for sinh(x). It is therefore potentially better to use the double-angle formula:

$$\cosh(2x) = \cosh^2(x) - 1 \qquad (110)$$

Alternatively, even better written as:

$$\cosh(2x) = 2(1 - \cosh(x))^2 - 4(1 - \cosh(x)) + 1 \qquad (111)$$

Although it does not prevent round-off errors it is less sensitive that the trisection formula. We calculate the reduction factor for cosh(x) as $k = 8\lceil \ln(2) * \ln(P) \rceil$ for higher precisions, and then we adjusted the magnitude of x. We add the exponent to the reduction factor. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply do not perform any argument reductions at all.
Since we are a little bit less sensitive using the double angle formula versus the trisection formula we only increase the precision with $\frac{3}{4}k$.

The performance is similar to the cosh(x) using the trisection as a reduction factor. Although you can use a little bit less precision it does not change the performance observed compared to the trisection formula of cosh(x).

Recommendation for calculating cosh(x)

Based on the performance measure of the various cosh(x) methods recommend:

- It is a matter of taste if you should use the cosh(x) using double-angle formula or trisection formula since the performance is equivalent.
- Do not use the Taylor series for cosh(x) with to aggressive reductions factor to speed up the Taylor term calculation.
- Also, use coefficient scaling to increase performance

Tanh(x)

Tanh(x) is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{112}$$

Which seems to be the most effective way of calculating tanh(x) using one call for exp(x).

We could also use the Taylor series for tanh(x):

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \cdots \frac{(-1)^{n-1} \cdot 2^{2n}\left(2^{2n} - 1\right)\beta_n x^{2n-1}}{(2n)!} + \cdots \tag{113}$$

Where B_n is the Bernoulli number. However, since we do not know how many Bernoulli numbers we need this will require us to calculate Bernoulli numbers on the fly and therefore much more complicated to implement than just a call to the e^{2x} function.

Recommendation for calculating tanh(x)

Based on the performance measure of the various tanh(x) methods recommend:

- Use the definition of tanh(x) using exp(x) in favor of using the Taylor series for tanh(x).

Arcsinh(x)

There are two methods. The direct method or the method using the Taylor series.

Arcsinh(x) direct method

Arcsinh(x) is equal to:

$$\text{Arcsinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right) \tag{114}$$

Ln(x) is relatively fast to calculate and the same goes for the square root. This direct method is the preferred way of calculating Arcsinh(x).

Arcsinh(x) using the Taylor series

Arcsinh(x) also has a Taylor series equivalence based on two Taylor series. This first Taylor series is for $|x| < 1$:

$$\text{Arcsinh}(x) = x - \frac{1}{2}\frac{x^3}{3} + \frac{1\cdot3}{2\cdot4}\frac{x^5}{5} - \frac{1\cdot3\cdot5}{2\cdot4\cdot6}\frac{x^7}{7} + \frac{1\cdot3\cdot5\cdot7}{2\cdot4\cdot6\cdot8}\frac{x^9}{9} - \cdots \quad (115)$$

The second Taylor series is for |x|>=1:

$$\text{Arcsinh}(x) = \pm\ln(2x) + \frac{1}{2}\frac{1}{2x^2} - \frac{1\cdot3}{2\cdot4}\frac{1}{4x^4} + \frac{1\cdot3\cdot5}{2\cdot4\cdot6}\frac{1}{6x^6} - \frac{1\cdot3\cdot5\cdot7}{2\cdot4\cdot6\cdot8}\frac{1}{8x^8} - \cdots \quad (116)$$

Where the + applies for x>=1 and – for x<=-1.

Example: Arcsinh(0.1)

A result is found using only seven Taylor terms

ArcSinh(x)		Original		|x|<1
	x=	0.1		
Terms	Term value	Term Sum	ArcSinh(x)	Error
1	1.00E-01	0.1000000000000	0.100000000000000	-1.66E-04
2	1.67E-04	0.0998333333333	0.099833333333333	7.46E-07
3	7.50E-07	0.0998340833333	0.099834083333333	-4.43E-09
4	4.46E-09	0.0998340788690	0.099834078869048	3.02E-11
5	3.04E-11	0.0998340788994	0.099834078899430	-2.22E-13
6	2.24E-13	0.0998340788992	0.099834078899206	1.73E-15
7	1.74E-15	0.0998340788992	0.099834078899208	0.00E+00

Example: Arcsinh(0.7)

A result is found after 27 Taylor Terms, but the result is not very accurate (error ~10^{-12})

ArcSinh(x)		Original		|x|<1
	x=	0.7		
Terms	Term value	Term Sum	ArcSinh(x)	Error
1	7.00E-01	0.7000000000000	0.700000000000000	-4.73E-02
2	5.72E-02	0.6428333333333	0.642833333333333	9.83E-03
3	1.26E-02	0.6554385833333	0.655438583333333	-2.77E-03
4	3.68E-03	0.6517620520833	0.651762052083333	9.05E-04
5	1.23E-03	0.6529880731293	0.652988073129340	-3.22E-04
6	4.42E-04	0.6525457024446	0.652545702444649	1.21E-04
7	1.68E-04	0.6527138316619	0.652713831661927	-4.73E-05
8	6.63E-05	0.6526475327072	0.652647532707247	1.90E-05
9	2.69E-05	0.6526744057210	0.652674405721047	-7.84E-06
10	1.11E-05	0.6526632785647	0.652663278564660	3.29E-06
11	4.69E-06	0.6526679649520	0.652667964952025	-1.40E-06
12	2.00E-06	0.6526659636053	0.652665963605294	6.02E-07
13	8.65E-07	0.6526668282204	0.652666828220438	-2.62E-07
14	3.77E-07	0.6526664510290	0.652666451029002	1.15E-07
15	1.66E-07	0.6526666169607	0.652666616960717	-5.09E-08
16	7.35E-08	0.6526665434351	0.652666543435125	2.26E-08
17	3.28E-08	0.6526665762216	0.652666576221552	-1.01E-08
18	1.47E-08	0.6526665615197	0.652666561519732	4.56E-09
19	6.63E-09	0.6526665681449	0.652666568144933	-2.06E-09
20	3.00E-09	0.6526665651461	0.652666565146113	9.36E-10
21	1.36E-09	0.6526665665089	0.652666566508912	-4.27E-10

22	6.22E-10	0.6526665658874	0.652666565887360	1.95E-10
23	2.84E-10	0.6526665661718	0.652666566171770	-8.94E-11
24	1.31E-10	0.6526665660412	0.652666566041240	4.11E-11
25	6.01E-11	0.6526665661013	0.652666566101311	-1.90E-11
26	2.77E-11	0.6526665660736	0.652666566073596	8.76E-12
27	1.28E-11	0.6526665660864	0.652666566086412	-4.06E-12

There are several issues with the Taylor series approach.

- First, the Taylor series convergence slowly when x is approaching one from either side.
- Secondly, you need to calculate the ln(2x) which takes approximately the same amount of time as the direct method.
- Thirdly, you cannot overcome the issue of slow convergence since there is no argument reduction available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series approach method is not recommended.

Recommendation for calculating Arcsinh(x)

Based on the performance measure of the various arcsinh(x) methods recommend:

- Use the Direct method: $\mathrm{Arcsinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$

Arccosh(x)

Again, there are two methods. The direct method or the method using the Taylor series.

Arccosh(x) direct method

Arccosh(x) is equal to:

$$\mathrm{Arccosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right), \text{ where } x \geq 1 \qquad (117)$$

Ln(x) is relatively fast to calculate and the same goes for the square root. This direct method is the preferred way of calculating Arccosh(x).

.

Arccosh(x) using the Taylor series

Arccosh(x) also have a Taylor series equivalence for |x|>=1:

$$\mathrm{Arccosh}(x) = \ln(2x) - \left(\frac{1}{2}\frac{1}{2x^2} + \frac{1\cdot 3}{2\cdot 4}\frac{1}{4x^4} + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\frac{1}{6x^6} + \frac{1\cdot 3\cdot 5\cdot 7}{2\cdot 4\cdot 6\cdot 8}\frac{1}{8x^8} - \cdots\right) \qquad (118)$$

However, it converges just as slowly as the Arcsinh(x).

Example: Arccosh(5) with argument reduction

A result is found using nine Taylor terms.

ArcCosh(x)	Original	x>=1

	x=	5		
Terms	**Term value**	**Term Sum**	**ArcCosh(x)**	**Error**
1	2.30E+00	2.30258509299405	2.30258509299405	-1.02E-02
2	1.00E-02	2.29258509299405	2.29258509299405	-1.53E-04
3	1.50E-04	2.29243509299405	2.29243509299405	-3.42E-06
4	3.33E-06	2.29243175966071	2.29243175966071	-9.01E-08
5	8.75E-08	2.29243167216071	2.29243167216071	-2.60E-09
6	2.52E-09	2.29243166964071	2.29243166964071	-7.95E-11
7	7.70E-11	2.29243166956371	2.29243166956371	-2.53E-12
8	2.45E-12	2.29243166956126	2.29243166956126	-8.30E-14
9	8.04E-14	2.29243166956118	2.29243166956118	0.00E+00

Example: Arccosh(2) with argument reduction

A result is found using twenty-one Taylor terms

ArcCosh(x)		**Original**		**x>=1**
	x=	**2**		
Terms	**Term value**	**Term Sum**	**ArcCosh(x)**	**Error**
1	1.39E+00	1.38629436111989	1.38629436111989	-6.93E-02
2	6.25E-02	1.32379436111989	1.32379436111989	-6.84E-03
3	5.86E-03	1.31793498611989	1.31793498611989	-9.77E-04
4	8.14E-04	1.31712118403656	1.31712118403656	-1.63E-04
5	1.34E-04	1.31698766963226	1.31698766963226	-2.98E-05
6	2.40E-05	1.31696363703949	1.31696363703949	-5.74E-06
7	4.59E-06	1.31695904748184	1.31695904748184	-1.15E-06
8	9.13E-07	1.31695813425353	1.31695813425353	-2.37E-07
9	1.87E-07	1.31695794697038	1.31695794697038	-5.00E-08
10	3.93E-08	1.31695790766404	1.31695790766404	-1.07E-08
11	8.40E-09	1.31695789926231	1.31695789926231	-2.34E-09
12	1.82E-09	1.31695789743962	1.31695789743962	-5.15E-10
13	4.00E-10	1.31695789703933	1.31695789703933	-1.15E-10
14	8.88E-11	1.31695789695050	1.31695789695050	-2.57E-11
15	1.99E-11	1.31695789693062	1.31695789693062	-5.81E-12
16	4.48E-12	1.31695789692614	1.31695789692614	-1.32E-12
17	1.02E-12	1.31695789692512	1.31695789692512	-3.02E-13
18	2.33E-13	1.31695789692489	1.31695789692489	-6.95E-14
19	5.34E-14	1.31695789692483	1.31695789692483	-1.62E-14
20	1.23E-14	1.31695789692482	1.31695789692482	-4.00E-15
21	2.85E-15	1.31695789692482	1.31695789692482	0.00E+00

Arccosh(x) suffers from the same deficit as the Taylor series for Arcsinh(x).

- First, the Taylor series convergence slowly when x is approaching one.
- Secondly, you need to calculate the ln(2x) which takes approximately the same amount of time as the direct method.
- Thirdly, you cannot overcome the issue of slow convergence since there is no argument reduction available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series approach method is not recommended.

Recommendation for calculating Arccosh(x)

Based on the performance measure of the various arcsinh(x) methods recommend:

- Use the Direct method: $\text{Arccosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$

Arctanh(x)

There are two interesting methods to use. One is the standard Taylor series and the other one is the direct method.

Arctanh(x) direct method

Arctanh(x) is equal to:

$$\text{Arctanh}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right), \text{ where } |x| < 1 \quad (119)$$

Ln(x) is relatively fast. This direct method is the preferred way of calculating Arctanh(x).
.

Arctanh(x) using the Taylor series

For arctanh(x) we can use a Taylor series until any additional addition does not change the result for the given precision of the number:

$$\text{Arctanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} - \cdots, \text{where } |x| < 1 \quad (120)$$

Notice the similarity with the arctan(x) Taylor series. Arctanh(x) does not use any alternating signs between Taylor terms as the arctan(x) Taylor series does.

Arctanh(x) suffers from the same weakness as the other hyperbolic function, that there is no argument reduction formula to lower the argument, x, and increase the performance of the Taylor series.

The Arctanh(x) Taylor series converges slowly and particularly close to 1 or -1.

Example Arctanh(0.1)

We need only seven Taylor terms to get the result.

ArcTanh(x)		**Original**		**|x|<1**
	x=	**0.1**		
Terms	Term value	Term Sum	ArcCosh(x)	Error
1	1.00E-01	0.100000000000000	0.100000000000000	3.35E-04
2	3.33E-04	0.100333333333333	0.100333333333333	2.01E-06
3	2.00E-06	0.100335333333333	0.100335333333333	1.44E-08
4	1.43E-08	0.100335347619048	0.100335347619048	1.12E-10
5	1.11E-10	0.100335347730159	0.100335347730159	9.17E-13
6	9.09E-13	0.100335347731068	0.100335347731068	7.79E-15
7	7.69E-15	0.100335347731076	0.100335347731076	0.00E+00

Example Arctanh(0.5)

Now we need 23 Taylor terms to get the result. More than three times as many as for arctanh(0.1).

ArcTanh(x)	x=	Original 0.5		|x|<1
Terms	Term value	Term Sum	ArcTanh(x)	Error
1	5.00E-01	0.500000000000000	0.500000000000000	4.93E-02
2	4.17E-02	0.541666666666667	0.541666666666667	7.64E-03
3	6.25E-03	0.547916666666667	0.547916666666667	1.39E-03
4	1.12E-03	0.549032738095238	0.549032738095238	2.73E-04
5	2.17E-04	0.549249751984127	0.549249751984127	5.64E-05
6	4.44E-05	0.549294141188672	0.549294141188672	1.20E-05
7	9.39E-06	0.549303531212711	0.549303531212711	2.61E-06
8	2.03E-06	0.549305565717919	0.549305565717919	5.79E-07
9	4.49E-07	0.549306014505833	0.549306014505833	1.30E-07
10	1.00E-07	0.549306114892603	0.549306114892603	2.94E-08
11	2.27E-08	0.549306137599134	0.549306137599134	6.73E-09
12	5.18E-09	0.549306142782147	0.549306142782147	1.55E-09
13	1.19E-09	0.549306143974239	0.549306143974239	3.60E-10
14	2.76E-10	0.549306144250187	0.549306144250187	8.39E-11
15	6.42E-11	0.549306144314416	0.549306144314416	1.96E-11
16	1.50E-11	0.549306144329437	0.549306144329437	4.62E-12
17	3.53E-12	0.549306144332965	0.549306144332965	1.09E-12
18	8.32E-13	0.549306144333797	0.549306144333797	2.58E-13
19	1.97E-13	0.549306144333993	0.549306144333993	6.16E-14
20	4.66E-14	0.549306144334040	0.549306144334040	1.50E-14
21	1.11E-14	0.549306144334051	0.549306144334051	3.89E-15
22	2.64E-15	0.549306144334054	0.549306144334054	1.22E-15
23	6.32E-16	0.549306144334054	0.549306144334054	0.00E+00

You can add coefficient scaling to speed things up. However, the Taylor series method is still many magnitudes slower than the direct method.

It suffers from the same deficit as the Taylor series for Arcsinh(x) and Arccosh(x) but not as bad from a performance perspective.

- First, the Taylor series converges slowly when x is approaching one.
- Secondly, you cannot overcome the issue of slow convergence since there is no argument reduction available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series approach method is not recommended.

Recommendation for calculating Arctanh(x)

Based on the performance measure of the various arctan(x) methods recommend:

Use the direct method. $\text{Arctanh}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$, where $|x| < 1$

(121)

Overall Recommendation for calculating Hyperbolic functions

- Use the Taylor series for calculating sinh(x) using argument reductions and coefficient scaling
- Use the Taylor series for calculating cosh(x) using argument reductions and coefficient scaling
- Use $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ for calculating tanh(x)
- Use $\text{Arcsinh}(x) = \ln\left(x + \sqrt{x^2+1}\right)$ for calculating arcsinh(x)
- Use $\text{Arccosh}(x) = \ln\left(x + \sqrt{x^2-1}\right)$ for calculating arccosh(x)
- Use $\text{Arctanh}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$ for calculating arctanh(x)

Gamma function

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:

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{122}$$

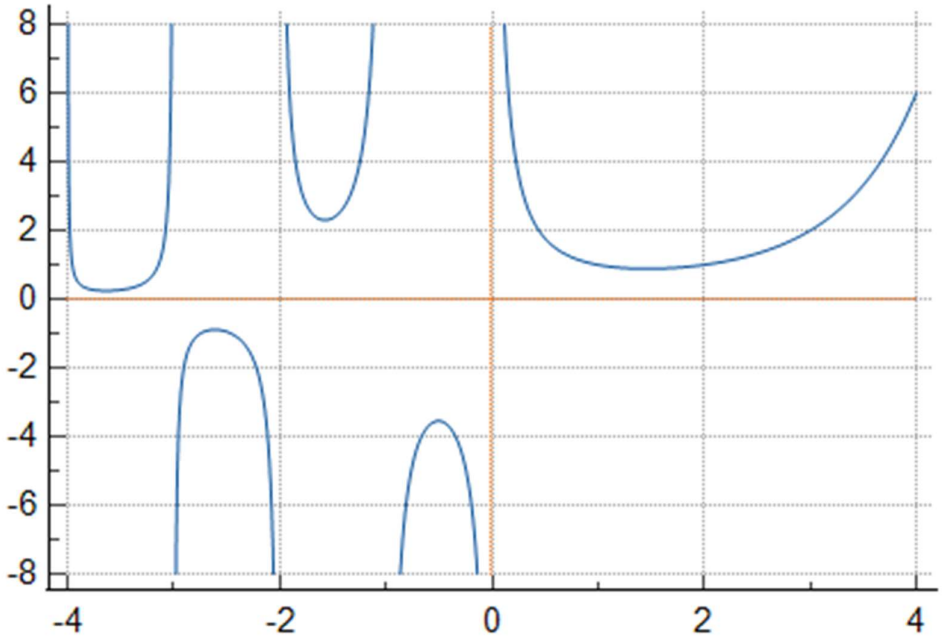


Figure 1. Gamma function in the interval [-4:+4]

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

$$\Gamma(z+1) = z\Gamma(z) \tag{123}$$

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:

$$\Gamma(n) = (n-1)! \tag{124}$$

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

$$\Gamma\left(\frac{1}{2} + n\right) = \frac{(2n)!}{2^{2n} n!}\sqrt{\pi}\ for\ n \geq 0 \tag{125}$$

Or in the negative half of the real axis:

$$\Gamma\left(\frac{1}{2} - n\right) = (-1)^n \frac{2^{2n} n!}{(2n)!}\sqrt{\pi}\ for\ n > 0 \tag{126}$$

Another important equation is Euler's reflection formula:

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(z\pi)} => \tag{127}$$

$$\Gamma(z) = \frac{\pi}{\Gamma(1-z)\sin(z\pi)} \tag{128}$$

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 (125) and (126).
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 15

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 precision arithmetic.

$$\Gamma(x) = \frac{(x+a)^{x+\frac{1}{2}}}{e^{x+a}}\left(c_0 + \sum_{k=1}^{a-1}\frac{c_k}{x+k}\right) \tag{129}$$

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

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 Yacas [6] they found that the lowest value to compute *P* correct digits in the calculation above was estimated to be:

$$a = \left\lceil \left(P - \frac{\ln(P)}{\ln(10)}\right)\frac{\ln(10)}{\ln(2\pi)} - \frac{1}{2} \right\rceil \quad (130)$$

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 [6] 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 [21] instead of finding a, from the wanted precision they give instead that the error is bounded by:

$$e_a(x) = \frac{1}{a^{0.5}(2\pi)^{a+0.5}} \quad (131)$$

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	

Stirling asymptotic series method

Stirling asymptotic series for Gamma function is given by:

$$\ln(\Gamma(x)) \sim \left(x - \frac{1}{2}\right)\ln(x) - x + \frac{1}{2}\ln(2\pi) + \sum_{n=1}^{\infty}\frac{B_{2n}}{2n(2n-1)x^{2n-1}} \quad (132)$$

Where B_{2n} is the Bernoulli number. In Yacas [6] they find that the optimal value for n in the summation is given by:

$$n_{optimal} \sim \pi|x| + 2 \quad (133)$$

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

$$(2\pi - 1)x + (x+1)ln(x) + 3.9 > P_{max} \cdot ln(10) => \quad (134)$$

$$P_{max} = \left\lfloor \frac{(2\pi-1)x + (x+1)ln(x) + 3.9}{\ln(10)} \right\rfloor \quad (135)$$

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 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:

$$\Gamma(x) = \frac{\Gamma(x+10)}{x(x+1)(x+2)(x+3)(x+4)(x+5)(x+6)(x+7)(x+8)(x+9)} \tag{136}$$

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

$$\Gamma(x) = \frac{\Gamma(x+M)}{\prod_{m=0}^{M-1}(x+m)} \tag{137}$$

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 [22] that states the number of shifts needed as follows.

$$|x + shifts| = P * \frac{\ln(10)}{\ln(2)} * 0.11038 => \tag{138}$$

$$shifts = P * \frac{\ln(10)}{\ln(2)} * 0.11038 - |x| \tag{139}$$

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 ∑ (also reducing the number of Bernoulli numbers we need to compute).

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 √
	Need to compute Bernoulli numbers
	Need to shift gamma value for small magnitude |x|
	Slow Computation

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:

$$\Gamma(x) = \int_0^M t^x e^{-t} \frac{dt}{t} + \int_M^\infty t^x e^{-t} \frac{dt}{t} \tag{140}$$

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:

$$\Gamma(x) \approx \int_0^M t^x e^{-t} \frac{dt}{t} = M^x e^{-M} \sum_{n=0}^{\infty} \frac{M^n}{\prod_{k=0}^{n}(x+k)} \quad (141)$$

Now the question is how to choose an appropriate M. in Yacas [6] they find the condition to be that:

$$M > (P + \ln(P))\ln(10) \quad (142)$$

The only thing missing now is how many terms (N_{max}) of the series you need to calculate. Again in [6] they find that to be:

$$N_{max} = P \frac{\ln(10)}{W(\frac{1}{e})}, \text{ where W is Lambert's function } W\left(\frac{1}{e}\right) \approx 0.2785 \quad (143)$$

With that we have the final formula:

$$\Gamma(x) \approx M^x e^{-M} \sum_{n=0}^{N_{max}} \frac{M^n}{\prod_{k=0}^{n}(x+k)} \quad (144)$$

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.

$$\Gamma(x) = \prod_{m=0}^{k-1}(x-m)\Gamma(x-k) \quad (145)$$

E.g. if 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]

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.

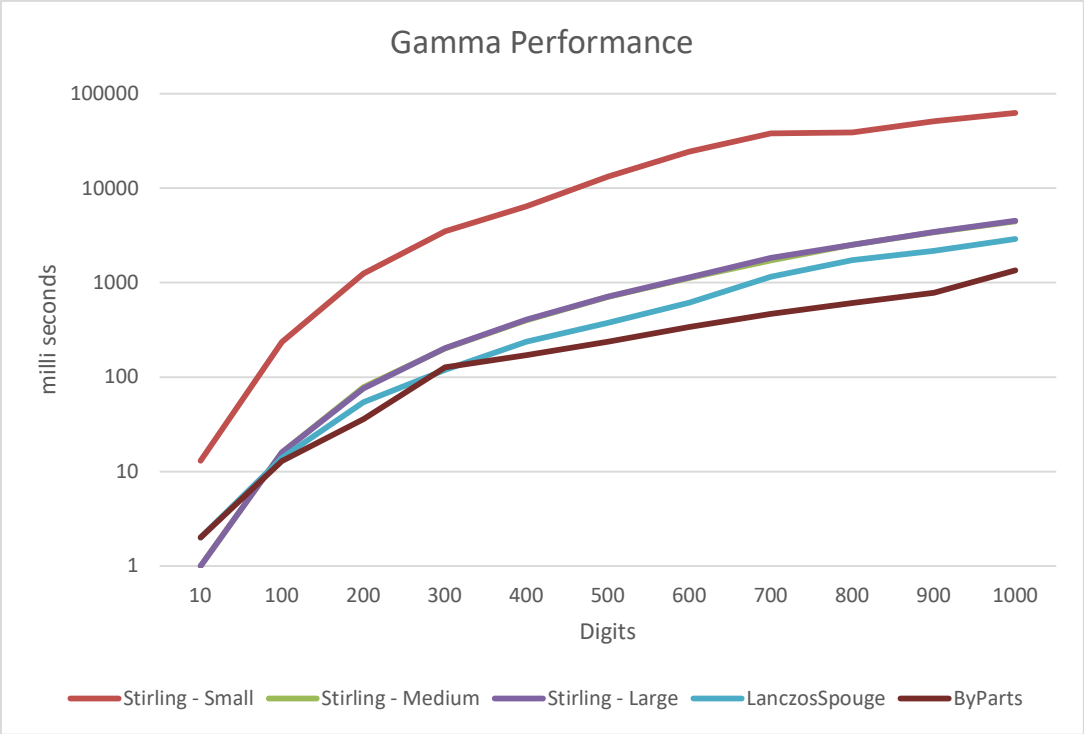


Figure 2. Gamma Performance

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 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:

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}dt \tag{146}$$

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:

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{147}$$

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)

The Error function

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

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{148}$$

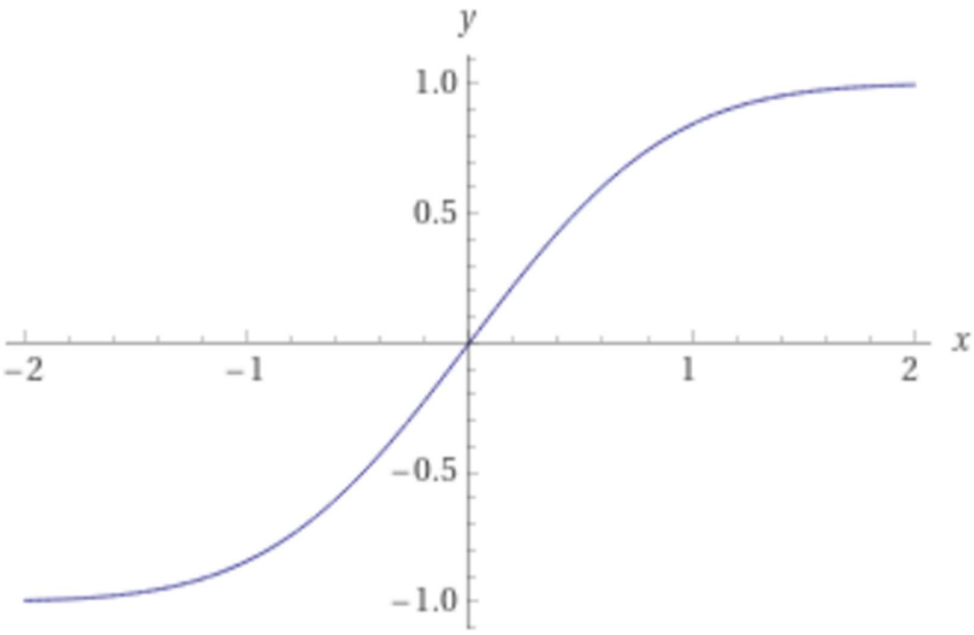


Figure 3. Error function in the interval [-2:+2]

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

The complementary error function is defined as:

$$\text{erfc(x)} = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt = 1 - \text{erf}\,(x) \tag{149}$$

For the numerical computation of the error function with arbitrary precision arithmetic there are three formulas suited for this job [24]:

Formula 1: $$\text{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n+1)n!} \tag{150}$$

Formula 2: $$\text{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}, !!\ is\ the\ double\ factorial \tag{151}$$

Formula 3: $$\text{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n-1)!!}{(2x^2)^n}, !!\ is\ the\ double\ factorial \tag{152}$$

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 accuracy if not

controlled. In [24] 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 x and then terminate the summation. In [6] they recommend only using formula one for |x|≤1 and give the following number of terms needed to archive an accuracy for P decimal digits:

$$n > 1 + e^{\left(1+W\left(\frac{P\cdot\ln(10)}{e}\right)\right)},\ W\ is\ the\ Lambert's\ W\ function \tag{153}$$

In [24] they also give an error bound for each formula and the readers kindly refer to [24] for the details. Furthermore, in [24] they also recommend using concurrent series summation instead of the straightforward way. This is a little bit more complicated to implement but [24] 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: 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 [24] they found that the achievable accuracy for P decimal digits was:

$$P \sim e^{2\ln(x)}\frac{\ln(2)}{\ln(10)} \tag{154}$$

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.

Figure 4. Performance of the Error function

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.

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.

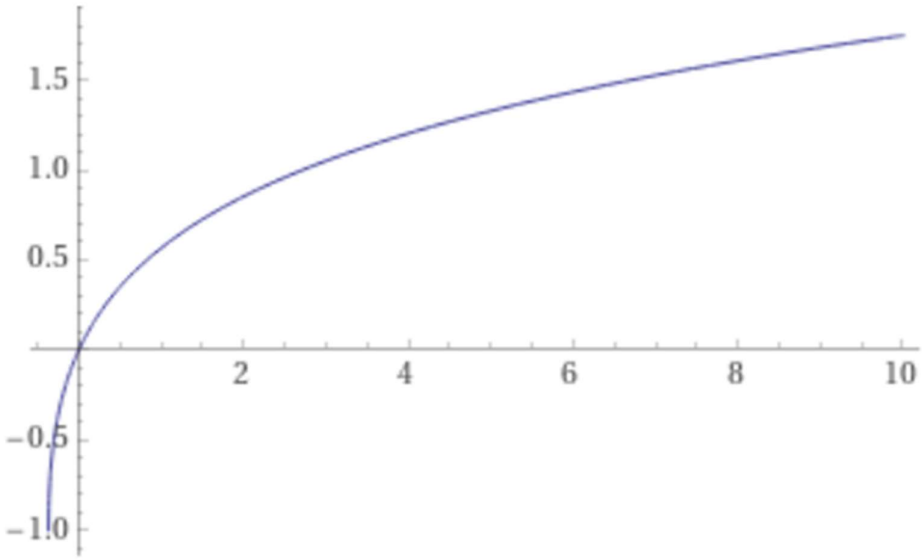


Figure 5 Lambert W function w0(x) in the interval]-1/e:10]

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 built-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.

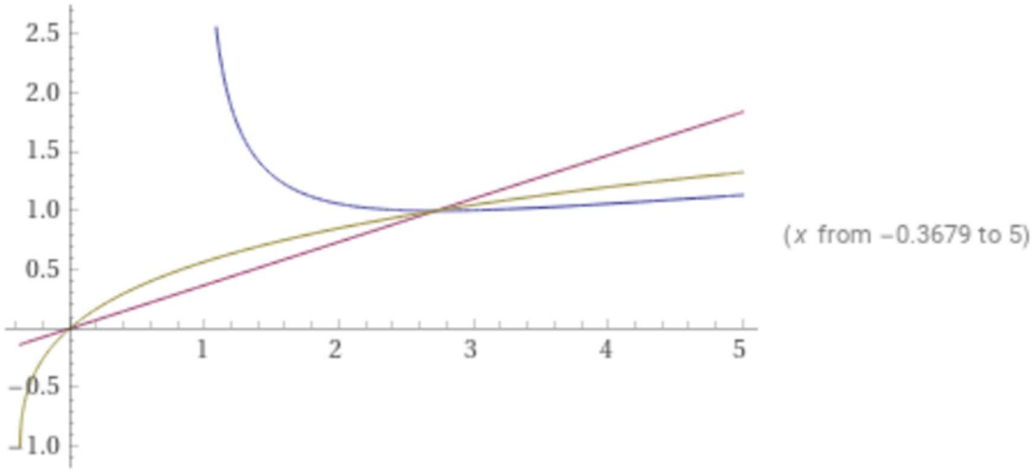


Figure 6 amber color is $W_0(x)$

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 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:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n} - x}{e^{w_n} + w_n e^{w_n}} \quad (155)$$

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:

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n} - x}{e^{w_n(w_n+1)} + \frac{(w_n+2)(w_n \cdot e^{w_n} - x)}{2(w_n+2)}} \quad (156)$$

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

Boyd quadratic method

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

$$w_{n+1} = \frac{w_n}{1+w_n}\left(1 + \ln\left(\frac{x}{w_n}\right)\right) \quad (157)$$

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 has previously been described in this paper. 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 graduate 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.

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.

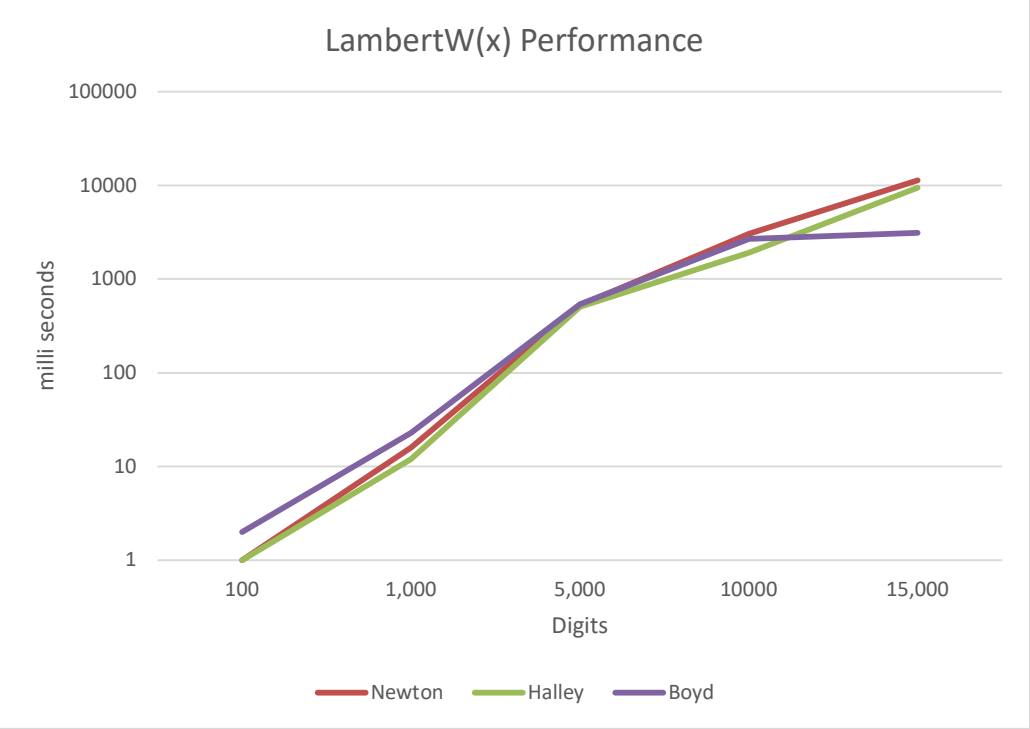


Figure 7. Performance of Lambert W(x) function

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:

$$\zeta(z) = \sum_{n=0}^{\infty} \frac{1}{n^z} \tag{158}$$

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

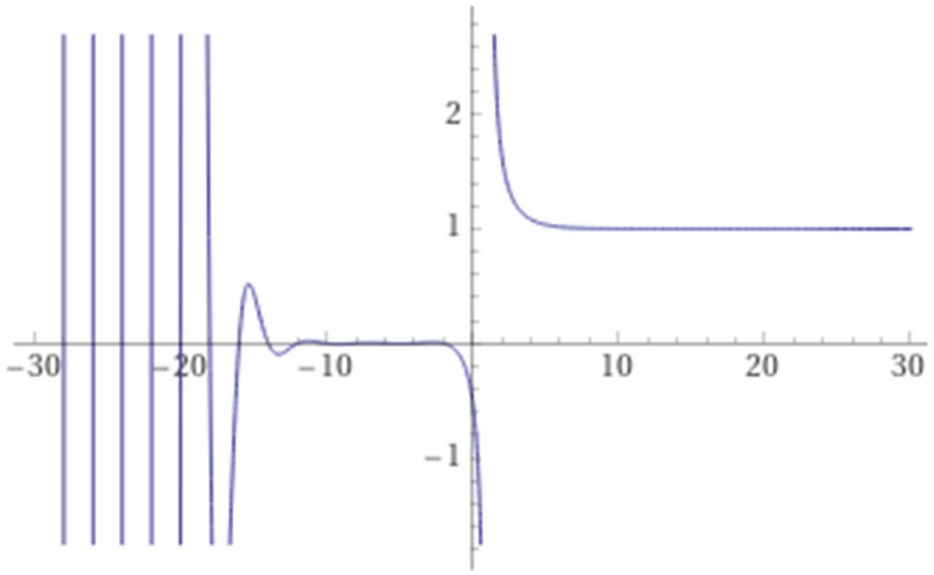


Figure 8 Zeta(x) in the interval [-30:+30]

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.

$$\zeta(1-s) = \frac{2\Gamma(s)}{(2\pi)^s}\cos\left(\frac{\pi \cdot s}{2}\right)\zeta(s) \tag{159}$$

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 [25] and in particular, his algorithm 3 is of interest dues to its simplicity. The formula ([25]) below is valid for any real s>-(n-1).

$$\zeta(s) = \frac{-1}{2^n(1-2^{1-s})}\sum_{j=0}^{2n-1}\frac{e_j}{(j+1)^s} \tag{160}$$

Where e_j is defined as:

$$e_j = (-1)^j\left(\left(\sum_{k=0}^{j-n}\frac{n!}{k!(n-k)!}\right) - 2^n\right) \tag{161}$$

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:

$$n = \left\lceil \frac{\ln(10)}{\ln(8)} P \right\rceil \tag{162}$$

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 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

$$s > 1 + P\frac{\ln(10)}{\ln(P)} \tag{163}$$

Where *P,* is the target precision (in decimal digits) is met [6]. We can resort to the below series for faster computation.

$$\zeta(s) \approx \sum_{k=0}^{N} \frac{1}{k^s} \tag{164}$$

And a suitable value for N is:

$$N = \left\lceil 10^{\frac{P}{s-1}} \right\rceil \tag{165}$$

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 the n'th Bernoulli number and n is an integer:

$$\zeta(0) = -\frac{1}{2} \tag{166}$$

$$\zeta(-n) = (-1)^n \frac{B_{n+1}}{n+1} \tag{167}$$

$$\zeta(2n) = (-1)^n \frac{B_{2n}(2\pi)^{2n}}{2(2n)!} \tag{168}$$

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 ζ(s) series. If we look at the computation for e_j in equation (161).

$$e_j = (-1)^j \left(\left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!}\right) - 2^n\right) \tag{169}$$

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:

$$e_j = (-1)^j \left(\left(\sum_{k=0}^{j-n} \binom{n}{k}\right) - 2^n\right) \tag{170}$$

However, the real optimization trick is when we realized that we can replace the ∑ 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:

$$e_j = (-1)^j (sum_j - 2^n) \tag{171}$$

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

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

Algorithm 16 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 (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)
```

Euler-Mascheroni constant

The Euler-Mascheroni constant γ is defined as:

$$\gamma = \lim_{n\to\infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \ln(n) \right) \approx 0.577215664 \tag{172}$$

The above equation (1) converts only slowly and it is not useful for arbitrary precision arithmetic. Instead, there are a few other interesting methods

- Brent-McMillan method
- Brent enhancement
- The binary splitting version of the Brent-McMillan method

Brent-McMillan method

To compute γ you can use the Brent-McMillan decomposition [6].

$$\gamma \approx \frac{S(n)}{V(n)} - \ln(n) \tag{173}$$

Where S(n) and V(n) are some auxiliary functions and n is chosen to ensure high enough precision for the result.

Furthermore the sequence S(n) and V(n) is defined as:

$$S(n) = \sum_{k=0}^{\infty}(\frac{n^k}{k!})^2 \cdot H_k \tag{174}$$

$$V(n) = \sum_{k=0}^{\infty}(\frac{n^k}{k!})^2 \tag{175}$$

And the sequence H_n is defined as the partial sum of the Harmonic series:

$$H_n = \sum_{k=1}^{n}\frac{1}{k} \tag{176}$$

Two questions arise. What is an appropriate value for *n* and when should the k summation stop? Brent estimated the minimum value for *n* as a function for the required precision *P* to be:

$$n = \left\lceil \frac{P \cdot \ln(10) + \ln(\pi)}{4} \right\rceil \tag{177}$$

And the required number of terms k_{max} in the summation as a function of the precision *P* to be:

$$k_{max} \approx 2.07 \cdot P \tag{178}$$

Technically we don't need to know the k_{max} before since we can just terminate the S(n) and V(n) sequence when the individual term value becomes less than the required precision dictate. (Usually, that will require a few more iterations than just using k_{max}).

Brent enhancement

Brent further improves the above formula by using a clever summation trick. Brent defined U(n) as:

$$U(n) = S(n) - V(n) \cdot \ln(n) \tag{179}$$

Then

$$\gamma \approx \frac{U(n)}{V(n)} \tag{180}$$

And

$$U(n) = \sum_{k=0}^{\infty} A_k \tag{181}$$

Where A_k is:

$$A_k = \left(\frac{n^k}{k!}\right)^2 (H_k - \ln(n)) \tag{182}$$

Furthermore, we use B_k as the n^{th} term of the V(n) series.

$$B_k = (\frac{n^k}{k!})^2 \tag{183}$$

We can then compute the A_k and B_k simultaneously using the below recurrence.

Algorithm Brent summation trick

$$A_0 = -\ln(n), B_0 = 1$$

$$B_k = B_{k-1} \cdot \frac{n^2}{k^2}$$

$$A_k = \frac{1}{k}\left(A_{k-1} \cdot \frac{n^2}{k} + B_k\right) = A_{k-1} \cdot \frac{n^2}{k^2} + \frac{B_k}{k}$$

Algorithm 17

Binary splitting method for γ

Lastly, you can use the Binary splitting technic as outlined in [26]

Algorithm: Binary splitting method for γ (7 variables)

$set\ m = \frac{a+b}{2}\ integer\ division$

P(a,b)=P(a,m)Q(m,b)+Q(a,m)P(m,b)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)S(m,b)+T(a,m)R(m,b)
S(a,b)=S(a,m)S(m,b)
T(a,b)=T(a,m)T(m,b)
U(a,b)=U(a,m)V(m,b)+P(a,m)T(a,m)Q(m,b)R(m,b)+Q(a,m)T(a,m)U(m,b)
V(a,b)=V(a,m)V(m,b)

And P(b-1,b)=1; Q(b-1,b)=b; R(b-1,b)=n^2; S(b-1,b)=b^2; T(b-1,b)=n^2; U(b-1,b)=n^2; V(b-1,b)=b^3;

Algorithm 18

You continue this recursive breakdown until a+1=b and you set:
P(a,b)=1 Q(a,b)=b R(a,b)=n^2 S(a,b)=b^2 T(a,b)=n^2 U(a,b)=n^2 V(a,b)=b^3
and let the formula reverse bottom up.

In the end, you find γ by:

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+S(0,i))} - \ln(n) \tag{184}$$

In [26] they found that i=3.5911214766686221366n as the number of needed terms as a function of n. And n can be chosen as in (26).

Now the binary splitting algorithm requires seven variables. You can quite easily reduce the number of variables from 7 to 5 by noting that S(m,b)=$Q(m,b)^2$ and V(m,b)=$Q(m,b)^3$, and you get the reduced variable version.

Algorithm: Binary splitting method for γ (5 variables)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+Q(a,m)P(m,b)
Q(a,b)=Q(a,m)Q(m,b)
$R(a,b)=R(a,m)Q(m,b)^2+T(a,m)R(m,b)$
T(a,b)=T(a,m)T(m,b)
$U(a,b)=U(a,m)Q(m,b)^3+P(a,m)T(a,m)Q(m,b)R(m,b)+Q(a,m)T(a,m)U(m,b)$

And P(b-1,b)=1; Q(b-1,b)=b; $R(b-1,b)=n^2$; $T(b-1,b)=n^2$; $U(b-1,b)=n^2$;

Algorithm 19

At the end you find γ by (now that S(0,i) has been replaced by $Q(0,i)^2$:

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+Q(0,i)^2)} - \ln(n) \tag{185}$$

The 5 variables version does perform better but not impressively better.

It is relatively easy to create a threaded version of the binary splitting algorithm and it has been proven advantageous to increase the performance by just threading the computational task.

Performance of Euler-Mascheroni constant

It is quite clear by looking at the performance chart that the binary splitting method is superior for the computation of the Euler-Mascheroni constant. The traditional Brent-McMillan (Brent 1) and Brent Summation trick (Brent 2) methods can't be recommended for fast computation of the Euler-Mascheroni constant. However Brent Summarization trick is a significant improvement over the standard Brent-McMillan formula.

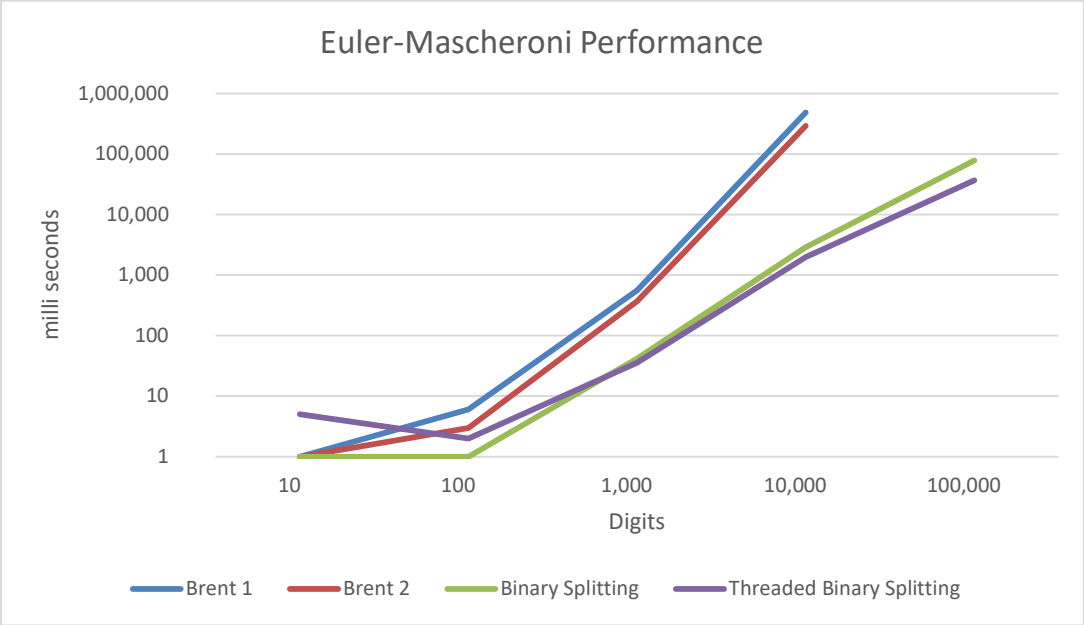


Figure 9 Euler-Mascheroni Performance

Recommendation for the Euler-Mascheroni constant

Based on the performance chart below, I recommend the Binary splitting method and for digits above 1,000 digits the threaded binary splitting version outperforms the non-threaded version.

Catalan's constant G

The Catalan constant G is defined as:

$$G = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2} \tag{186}$$

The Catalan constant is ~0.9159655941772…

This series also convert slowly. However, there are several alternative methods to consider.

- Ramanujan method I
- Ramanujan method II
- Broadhurst series
- Binary splitting method (ref. [4])
  - Lucas(2000)
  - Guillera (2008)
  - Guillera (2019)
  - Pilehrood (2010)

Ramanujan's method I

This is one of the many Ramanujan series for fast calculation of the Catalan constant.

$$G = \frac{\pi}{8} ln\left(2+\sqrt{3}\right) + \frac{3}{8}\sum_{n=0}^{\infty} \frac{(n!)^2}{(2n)!(2n+1)^2} \tag{187}$$

To achieve *P*, decimal precision we need to take $P\frac{\ln(10)}{\ln(4)}$ terms of the series and we can use Horner's schema to efficiently summarize the series. One of the drawbacks of this method is that we need to calculate π, ln(2+$\sqrt{3}$), and $\sqrt{3}$ to arbitrary precision. Horner's schema looks like this:

$$1 + \frac{1}{2\cdot 3}\left(\frac{1}{3} + \frac{2}{2\cdot 5}\left(\frac{1}{5} + \frac{3}{2\cdot 7}\left(\frac{1}{7} + \cdots\right)\right)\right) \tag{188}$$

And the algorithm for the Horner schema sum.

```
Sum=0
For(k=1;k<=n;++k)
        Sum+=1/(2k+1)
        Sum*=k/(2(2k+1))
```

Algorithm 20

Ramanujan's method II

Another of Ramanujan's methods is based on this formula:

$$G = \sum_{k=0}^{\infty} \frac{(k!)^2}{(2k+1)!} \cdot 2^{k-1} \sum_{j=0}^{k} \frac{1}{2j+1} \tag{189}$$

In [23] Free use this formula and Brent summation trick to obtain the following algorithm:

Algorithm for Ramanujan's II

B_0=0.5, C_0=0.5, G_0=0.5
For(k=1;k<=n;++k)
    tmp=k/(2k+1)
    B_k=B_{k-1}*tmp
    C_k=C_{k-1}*tmp+B_k/(2k+1)
    G_k=G_{k-1}+C_k

Algorithm 21

We need a little bit more to achieve *P*, decimal precision compare to the first method. In [6] find that we need to take $P\frac{\ln(10)}{\ln(2)}$ terms to reach the desired precision.

Broadhurst series

Broadhurst series has a faster convergence rate than Ramanujan's series at the expense of higher complexity.

$$G = 3\sum_{k=0}^{\infty} \frac{1}{16^k} \sum_{i=0}^{7} \frac{a_i}{(8k+i)^2} - 2\sum_{k=0}^{\infty} \frac{1}{4096^k} \sum_{i=0}^{7} \frac{b_i}{(8k+i)^2} \tag{190}$$

Where:

a_i=(0,1/2,-1/2,1/4,0,-1/8,1/8,-1/16) and
b_i=(0,1/8,1/16,1/64,0,-1/512,-1/1024,-1/4096).

For a precision *P*, we need only to take $\left\lceil P\frac{\ln(10)}{\ln(16)} \right\rceil$ terms to reach the desired precision of the first series and $\left\lceil P\frac{\ln(10)}{\ln(4096)} \right\rceil$ for the second series. However, each term is also 6 times more complicated than the Ramanujan I series. In [6] they state that due to the extra complexity, it is not worth implementing it. However, I found that an efficient implementation of the Broadhurst series results in higher performance than the Ramanujan series for the Catalan constant.

Lupas Binary Splitting method

Lupas series for the Catalan constant is:

$$G = \frac{1}{64}\sum_{k=1}^{\infty} \frac{(-1)^{k-1}2^{8k}(40k^2-24k+\ )(2k)!^3 k!^2}{k^3(2k-1)(4k)!^2} \tag{191}$$

As we have seen many times before we can transform this series into a binary Splitting method using the below algorithm:

Algorithm: Binary splitting method for Catalan – Lupas (2000)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
$P(b-1,b)=(32(2b-1)b^3)(40b^2+56b+19)(-1)^b$
$Q(b-1,b)=(4b+1)^2(4b+3)^2$
$R(b-1,b)=32(2b-1)b^3$

Algorithm 22

You continue this recursive breakdown until a+1=b and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{P(0,n)+19Q(0,n)}{18Q(0,n)} + O(4^{-n}) \tag{192}$$

For n terms, the error is $O(4^{-n})$ and for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(4)} \right\rceil \tag{193}$$

In [26] they found the linearly convergent cost to be ~11.5, which makes is not as fast as the Guillera or Pilehrood methods.

Guillera Binary Splitting method

Guillera publish two methods back in 2008 and 2019. The first method used:

$$G = \frac{1}{2}\sum_{k=0}^{\infty} \frac{(-8)^k(3k+2)}{(2k+1)^3\binom{2k}{k}^3} \tag{194}$$

Converting into a binary splitting method, they found in [26] that the linearly convergent cost is ~11.5 around the same as for the Lupas binary splitting method. In [26] they rewrote the formula to:

$$G = \frac{1}{2}\sum_{k=0}^{\infty} \frac{(-8)^k(3k+2)k!^6}{(2k+1)!^3} \tag{195}$$

And archive a linearly convergent cost of ~5.7 making it faster than the Lupas method.

Algorithm: Binary splitting method for Catalan – Guillera (2008)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:

P(b-1,b)= $b^3(3b+2)$
Q(b-1,b)=$-(2b+1)^3(2b-1)^3$
R(b-1,b)=$b^3(2b+1)^3$

Algorithm 23

You continue this recursive breakdown until a+1=b and let the formula reverse bottom up.

In the end, you find G by:

$$G = 1 + \frac{1}{2}\frac{P(0,n)}{Q(0,n)} + O(8^{-n}) \tag{196}$$

For n terms the error is $O(8^{-n})$ and for a given precision, P you need:

$$n = \left\lceil P\frac{\ln(10)}{\ln(8)} \right\rceil \tag{197}$$

In 2019 Guillera publish another formula with a higher convergence rate. The formula looks intimidating at first glance:

$$G = -\frac{1}{1024}\sum_{k=1}^{\infty}\frac{(-409\ )^k(45136k^4-57184k^3+21240k^2-3160k+165)}{k^3(2k-1)^3}\left(\frac{(2k)!^6(3k)!^3}{k!^3(6k)!^3}\right) \tag{198}$$

But has a linearly convergent cost of only ~4.2 which is lower than Guillera's formula from 2008.

Algorithm: Binary splitting method for Catalan – Guillera (2019)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
P(b-1,b)= $45136b^4-57184b^3+21240b^2-3160b+165$
Q(b-1,b)=$-27(6b-1)^3(6b-5)^3$
R(b-1,b)=$512b^3(2b-1)^3$

Algorithm 24

You continue this recursive breakdown until a+1=b and let the formula reverse bottom up.

In the end, you find G by:

$$G = -\frac{1}{2}\frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{19683}{64}\right)^{-n}\right) \tag{199}$$

For n terms, the error is $O\left(\left(\frac{19683}{64}\right)^{-n}\right)$ and for a given precision, P you need:

$$n = \left\lceil P\frac{\ln(10)}{\ln\left(\frac{19683}{64}\right)} \right\rceil \tag{200}$$

Pilehrood binary splitting method

Pilehrood publish two formulas in 2010. The short and long formula.

$$G = \frac{1}{64}\sum_{k=1}^{\infty}\frac{256^k\left(580k^2-184k+15\right)}{k^3(2k-1)\binom{6k}{3k}\binom{6k}{4k}\binom{4k}{2k}} \tag{201}$$

When applying the binary splitting method you get a linearly convergent cost of only ~3.1 which is the lowest of all the Catalan binary splitting methods.

Algorithm: Binary splitting method for Catalan – Pilehrood (2010-short)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
P(b-1,b)= $580b^2$-184b+15
Q(b-1,b)=$9(6b-1)^2(6b-5)^2$
R(b-1,b)=$32b^3(2b-1)$

Algorithm 25

You continue this recursive breakdown until a+1=b and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{1}{2}\frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{729}{4}\right)^{-n}\right) \tag{202}$$

For n terms, the error is $O\left(\left(\frac{729}{4}\right)^{-n}\right)$ and for a given precision, P you need:

$$n = \left\lceil P\frac{\ln(10)}{\ln\left(\frac{729}{4}\right)}\right\rceil \tag{203}$$

Pilehrood also has a long version formula:

$$G = -\frac{1}{64}\sum_{k=1}^{\infty}\frac{(-256)^k\left(419840k^6-915456k^5+782848k^4-332800k^3+7325\quad^2-7800k+315\right)}{k^3(2k-1)(4k-1)^2(4k-3)^2\binom{8k}{4k}^2\binom{2k}{k}} \tag{204}$$

Which have a linearly convergent cost of ~4.6 which is higher than the Pilehrood short version.

Algorithm: Binary splitting method for Catalan – Pilehrood (2010-long)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)

Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
$P(b-1,b)=(-1)^b(419840b^6-915456b^5+782848b^4-332800b^3+73256b^2-7800b+315)$
$Q(b-1,b)=(8b-1)^2(8b-3)^2(8b-5)^2(8b-7)^2$
$R(b-1,b)=32b^3(2b-1)(4b-1)^2(4b-3)^2$

Algorithm 26

You continue this recursive breakdown until a+1=b and let the formula reverse bottom up.

In the end, you find G by:

$$G=-\frac{1}{2}\frac{P(0,n)}{Q(0,n)}+O(1024^{-n}) \tag{205}$$

For n terms the error is $O(1024^{-n})$ and for a given precision, P you need:

$$n=\left\lceil P\frac{\ln(10)}{\ln(1024)}\right\rceil \tag{206}$$

Comparison of the Catalan Methods

We have outlined quite a few methods for calculating the Catalan constant. To get an overview of the different methods you can look at the below table that outlines the name, implementation type, error, and precision requirement.

Method	Implementation	Error	N(P), *P=precision*
Ramanujan-I	Series	$O(4^{-n})$	1.661P
Ramanujan-II	Series	$O(2^{-4})$	3.322P
Broadhurst	Series	$O(16^{-n})$	0.830P
Lupas	Binary Splitting	$O(4^{-n})$	1.661P
Guillera-2008	Binary-Splitting	$O(8^{-n})$	1.107P
Guillera-2019	Binary-Splitting	$O((\frac{19683}{64})^{-n})$	0.402P
Pilehrood-short	Binary-Splitting	$O((\frac{729}{4})^{-n})$	0.442P
Pilehrood-long	Binary-Splitting	$O(1024^{-n})$	0.332P

Not surprisingly the performance depends heavily on the convergence speed and implementation type e.g. Series or binary splitting method as shown in the next section.

Catalan Constant Performance

Not surprisingly the linearly convergent cost predicts the performance of the method. The clear winner is the Pilehrood binary splitting method from 2010. It outperforms the others significantly. Furthermore, a two-way multi-threaded version further improves the performance by 30-40%. The Pilehrood method is 40-50% faster than Guillera 2019 method and 90-100% faster than Guillera 2008 method. Comparing Pilehrood and Lucas, Pilehrood is more than 5 times faster. If we compare the Binary splitting method against the classical

series formula the binary splitting version is several magnitudes faster and among the classical series the Broadhurst method is by far the fastest.

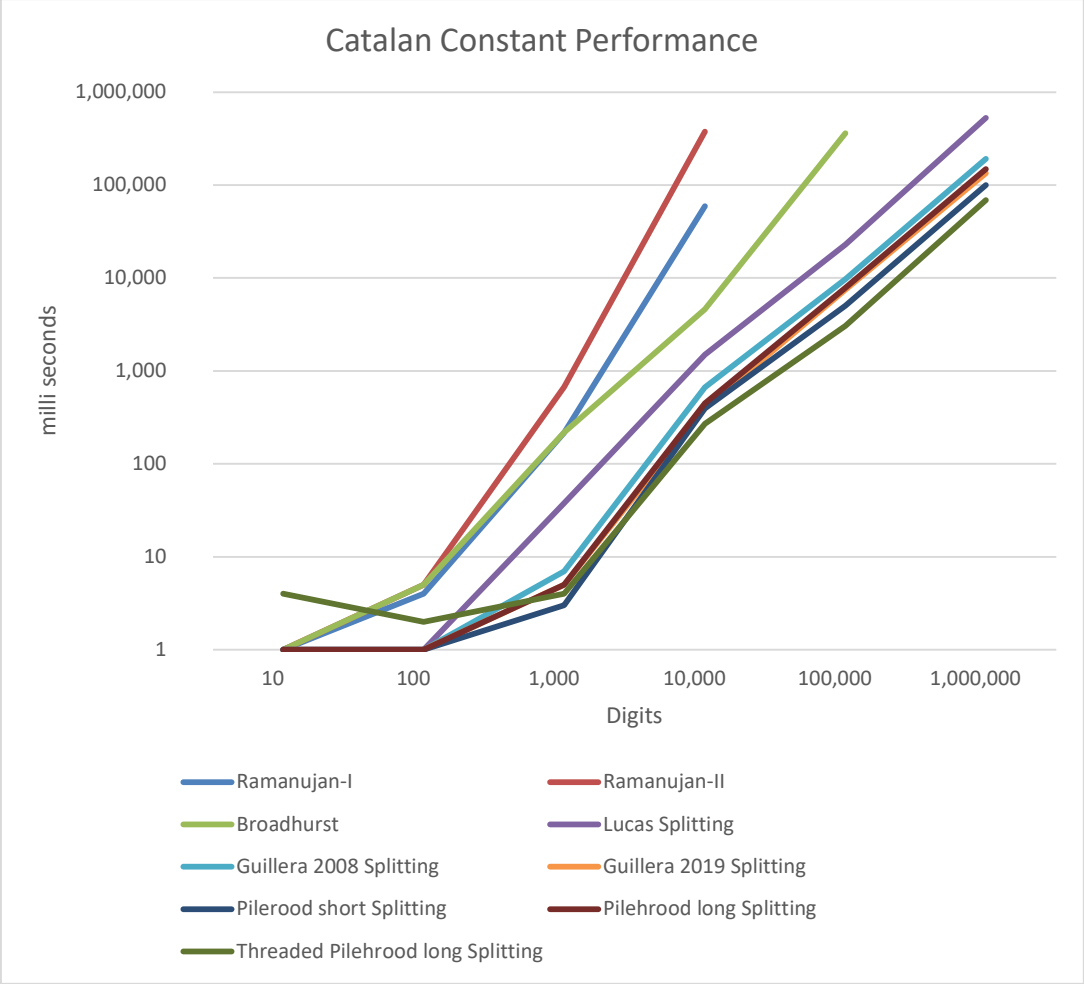


Figure 10 Catalan Constant Performance chart

Recommendation for the Catalan constant

Based on the performance chart and ease of implementation I recommend the Pilehrood 2010 short version as the preferred binary splitting method. Also if performance is required then use or implement a threaded version of the Pilehrood method. It is very easy to create a 2, 3, 4, or more core-threaded version of the binary splitting method. If we only want a classical method then I recommend the Broadhurst method.

Apéry's constant ζ(3)

Is the common short name for the ζ(3) value. This is a specialized formula for the ζ(3) instead of using the more general computation of ζ(s) in [6]. As I mentioned in [6] there has been researched into finding a formula, series, etc. for the odd integer's values of the zeta function. One of them is the value of ζ(3). There is two formula that comes to mind and these are [4]:

- Amdeberhan-Zeilberger series
- Wedeniwski series

Amdeberhan-Zeilberger series

This series is given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3) = \frac{1}{64}\sum_{k=0}^{\infty} \frac{(-1)^k(205k^2+250k+77)(k!)^{10}}{(2k+1)^5} \tag{207}$$

Now by now, we should have learned that the most efficient computation is by using the binary splitting method.

Algorithm: Binary splitting method for ζ(3) (1997)

$set\ m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
$P(b-1,b)=(-1)^b(205b^2+250b+77)b^5$
$Q(b-1,b)=32(2b+1)^5$
$R(b-1,b)=b^5$

Algorithm 27

And then

$$\zeta(3) = \frac{P(0,n)+77Q(0,n)}{64\ \ (0,n)} + O(1024^{-n}) \tag{**208**}$$

Which have a linearly convergent cost of ~2.89 which is slightly higher than the next Wedeniwski method.

For *n* terms the error is $O(1024^{-n})$ and for a given precision, *P* you need:

$$n = \left\lceil P\frac{\ln(10)}{\ln(1024)} \right\rceil \tag{209}$$

Wedeniwski series

This series is given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3)=\frac{1}{24}\sum_{k=0}^{\infty}\frac{(-1)^k(126392k^5+412708k^4+531578k^3+336367k^2+104000k+12643)((2k+1)!(2k)!k!)^3}{(3k+2)!(4k+3)!^3} \quad (210)$$

Algorithm: Wedeniwski Binary splitting method for ζ(3) (1998)

$set\ m=\frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

And:
P(b-1,b)=(-1)^b(126392b^5+412708b^4+531578b^3+336367b^2+104000b+12463)b^5(2b-1)^3
Q(b-1,b)=24(3b+1)(3b+2)(4b+1)^3(4b+3)^3
R(b-1,b)=b^5(2b-1)^3

Algorithm 28

And then

$$\zeta(3)=\frac{P(0,n)+1246\quad(0,n)}{10368\quad(0,n)}+O(110592^{-n}) \quad (211)$$

Which have a linearly convergent cost of ~2.78 which is slightly lower than the Amdeberhan-Zeilberger method. You should expect close to the same performance for both methods.

For n terms the error is $O(110592^{-n})$ and for a given precision, P you need:

$$n=\left\lceil P\frac{\ln(10)}{\ln(110592)}\right\rceil \quad (212)$$

Apéry Constant ζ(3) performance

Both Binary splitting methods outperform the general zeta function implementation. It seems that the Wedeniwski method has a slight edge over the Amdeberhan. This was expected since the linearly convergent cost is 2.78 for Wedeniwski versus 2.89 for Amdeberhan-Zeilberger. Furthermore, Wedeniwski only needs ~0.198*Precision terms versus ~0.332*Precision terms for Amdeberhan-Zeilberger, However, the computation of the P(b-1,b), Q(b-1.b) and R(b-1,b) is more complicated for the Wedeniwski method.

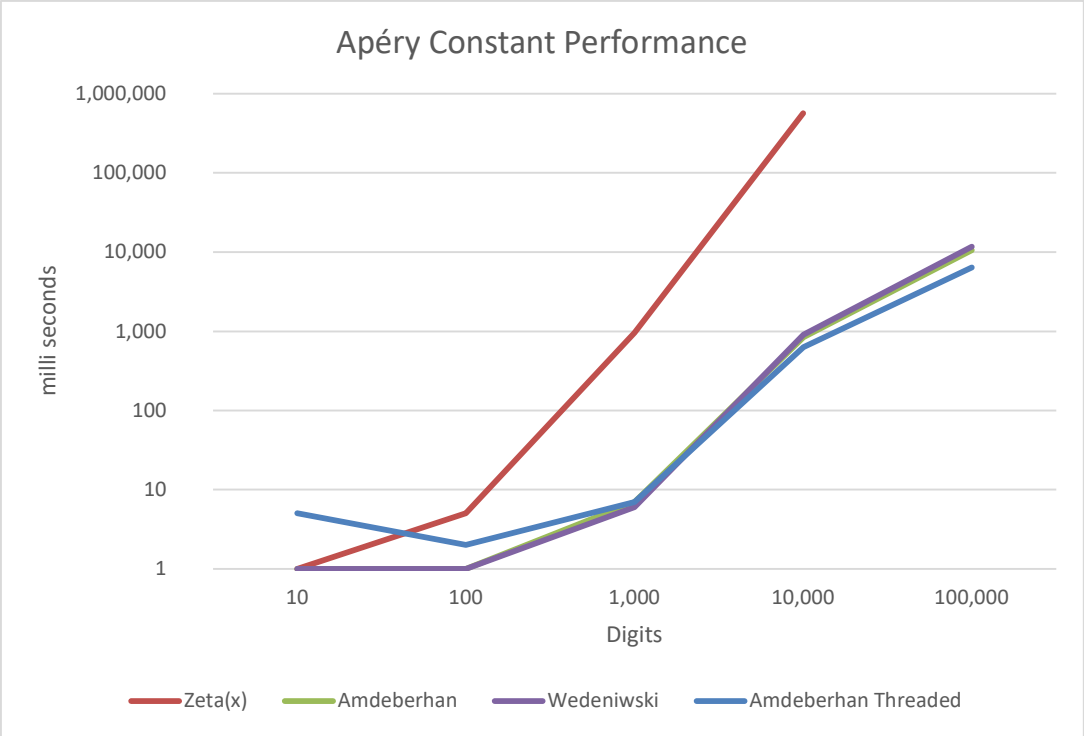


Figure 11 Apéry Constant Performance

Recommendation for the constant ζ(3)

I recommend the following:

1) It is clear that if you are serious you would implement one of the binary splitting methods.
2) The general zeta(s) function is not recommended for the computation of the Apéry constant ζ (3).
3) Wedeniwski is slightly faster but Amdeberhan-Zeilberger is simpler to implement.
4) Implement the threaded version for the binary splitting method if speed is of the essence.

Appendix

A summary table of the preferred method for arbitrary precision arithmetic.

	Prefer method
Integer arithmetic	
Addition	School book addition
Subtraction	School book subtraction
Multiplication	Linear convolution for a smaller number, otherwise FFT
Division	Use floating-point division
Remainder	If $\frac{a_n}{b_m} = c_k$ then $rem = a_n - b_m \cdot c_k$
Floating point arithmetic	
Addition	School book addition
Subtraction	School book subtraction
Multiplication	Linear convolution for smaller numbers otherwise FFT
Division	Newton or Halley iteration
$\sqrt{x}$	Newton or Halley iteration
$\sqrt[n]{x}$	Newton iteration
x^y	$x^y = e^{y \cdot \ln(x)}$ or simpler if the argument allows
Elementary functions	
e^x	Sinh(x) Taylor series with argument reduction and coefficient scaling
Log(x)	For smaller numbers Taylor series for log(x) with argument reduction and coefficient scaling. For larger numbers the AGM method
Trigonometric functions	
Sin(x)	Taylor series with argument reduction and coefficient scaling
Cos(X)	Use $\cos(x) = \sqrt{1 - sin^2(x)}$
Tan(x)	Use $\tan(x) = \frac{\sin(x)}{\sqrt{1-\sin^2(x)}}$
Arcsin(x)	Taylor series with argument reduction and coefficient scaling
Arccos(x)	Use $\text{Arccos}(x) = \frac{\pi}{2} - \text{Arcsin}(x)$
Arctan(x)	Taylor series with argument reduction and coefficient scaling
Hyperbolic functions	
Sinh(x)	Taylor series with argument reduction and coefficient scaling
Cosh(x)	Taylor series with argument reduction and coefficient scaling
Tanh(x)	Use $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
Arcsinh(x)	Use $\text{Arcsinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$
Arccosh(x)	Use $\text{Arccosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$

Arctanh(x)	Use $\mathrm{Arctanh(x)} = \frac{1}{2}\ln\left(\frac{1+\mathrm{x}}{1-\mathrm{x}}\right)$
Special functions	
Gamma function	Integration by parts
Beta function	$B(z,w) = \frac{\Gamma(\mathrm{z})\Gamma(\mathrm{w})}{\Gamma(\mathrm{z}+\mathrm{w})}$
Error Function	$\mathrm{erf}(x)$ $= \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}, !!\, is\ the\ double\ factorial$
Lamber W function	Boyd's iteration: $w_{n+1} = \frac{w_n}{1+w_n}(1 + \ln(\frac{x}{w_n}))$
Zeta function	An optimized version of P. Borwein algorithm 3: $\zeta(s) = \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s}$ Where: $e_j = (-1)^j((\sum_{k=0}^{j-n} \frac{n!}{k!\,(n-k)!}) - 2^n)$
Constants	
e	Binary splitting method
Ln(2)	Spigot algorithm, alternatively you can use log(2)
Ln(10)	Spigot algorithm, alternatively you can use log(10)
π	Chudnovsky Binary splitting method
Special Constants	
Euler-Mascheroni	Binary Splitting Method
Catalan	Pilehrood 2010 short version Binary Splitting Method
Apéry (zeta(3))	Amdeberhan-Zeilberger Binary Splitting Method

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. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ 1963
4. Methods of Computing square roots; May 17-2013; http://en.wikipedia.org/wiki/Methods_of_computing_square_roots
5. Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
6. The Yacas book of algorithms, Version 1.3.3, April 1 2013 by the Yacas team
7. Richard Brent & Paul Zimmermann, Modern Computer Arithmetic, Version 0.5.9 17 October 2010; http://maths-people.anu.edu.au/~brent/pd/mca-cup-0.5.9.pdf
8. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision (hvks.com)
9. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision (hvks.com)
10. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision v2 (hvks.com)
11. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants (hvks.com)
12. Methods of computing binary splitting. Mathematical Constants and computation (free.fr) (direct link) Binary splitting method (free.fr)
13. Practical implementation of spigot algorithm for transcendental constants. HVE Practical implementation of Spigot Algorithms for transcendental constants (hvks.com)
14. The world of π. http://www.pi314.net/eng/goutte.php - Dec 28, 2016
15. D Bailey, A compendium of BBP-Type Formulas for Mathematical Constants. April 29, 2013
16. The World of π. www.pi314.net/eng/salamin.php - Oct 5, 2016
17. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms rev2 (hvks.com)
18. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision (hvks.com)
19. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision (hvks.com)
20. Boost, performance comparison of nrooth algorithm https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/math_toolkit/root_comparison/root_n_comparison.html
21. J.L. Spouge, Computation of the Gamma, Diagamma, and Trigamma functions. SIAM Journal on Numerical Analysis. 31(3): 931-000
22. Frederik Johansson, Arbitrary-precision computation of the gamma function. 2021. Hal-03346642. HAL open science.
23. G.Free, Computation of Catalan's Constant using Ramanujan's formula. 1990 ACM

24. 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
25. P. Borwein, “An efficient Algorithm for the Riemann Zeta function”, Canadian Mathematical Society, Conference paper.
26. Alexander Yee, Binary Splitting Recursion Library