

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

By Henrik Vestermark (hve@hvks.com)

Abstract:

Modern computing provides exceptional speed for fixed-precision arithmetic, yet implementing arbitrary precision operations, capable of handling integers and floating-point numbers with millions of digits, requires returning to fundamental mathematical principles. This paper presents a comprehensive mathematical framework for constructing high-performance arbitrary precision libraries, demonstrating how classical algorithms combine with advanced optimization techniques to achieve practical computational efficiency. Beginning with integer arithmetic foundations (addition, subtraction, multiplication, and division), we progress through increasingly sophisticated methods including Karatsuba, Toom-Cook, FFT, and NTT multiplication algorithms. The paper then extends these principles to floating-point operations, followed by detailed implementations of transcendental functions: square roots, logarithms, exponentials, trigonometric and hyperbolic functions, and specialized functions including Gamma, Beta, Error, Zeta, and Lambert W. Each section explores optimization strategies essential for maintaining performance at extreme precision levels, from hundreds to millions of digits. This 2025 revision significantly expands upon earlier versions, incorporating refined methodologies, detailed analysis of integer division algorithms, mathematical derivations for the Lemniscate constant, and performance comparisons across multiple computational approaches, providing both theoretical foundations and practical implementation guidance for arbitrary precision arithmetic systems.

Introduction:

This paper describes the underlying mathematics behind this package [1] and is a completely updated and expanded version of the original 2013 paper, as well as the revised versions from 2023 and 2025.

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 the formulas and mathematics that underlie the arbitrary precision packages, beginning with arbitrary integer precision and then proceeding to floating-point math. For floating-point math, when a floating-point number is broken down into its base components of <integer>, <fraction>, and <exponent>, it also utilizes the integer precision math library for the calculation. After the basic floating-point operators like addition, subtraction, multiplication, and division, we build upon these functions to implement $\sqrt{\quad}$, 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 techniques to improve performance, particularly when the needed precision exceeds 100 digits and goes into the million-digit precision and higher. This paper has been influenced by the Yacas book of algorithms [6], but many of the references listed are worth reading. E.g. [7], [12], [14], [21], [22], [26], [34].

Change log

10 October 2025. Revised and updated. Clarity has been improved, and several new methods have been introduced throughout the paper. Furthermore, a portion of the paper has been revised, including a more detailed description of various integer divisions, and the mathematical background behind the Lemniscate constant has been added. Typo has also been corrected.

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 inconsistencies 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. 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 Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
2. Fast Square Root & Inverse calculation for arbitrary precision math. [HVE Fast Square Root & inverse calculation for arbitrary precision](#)
3. Fast Exponential calculation for arbitrary precision math. [HVE Fast Exp\(\) calculation for arbitrary precision](#)
4. Fast logarithm calculation for arbitrary precision math. [HVE Fast Log\(\) calculation for arbitrary precision](#)
5. Practical implementation of Spigot Algorithms for Transcendental Constants. [Practical implementation of Spigot Algorithms for transcendental constants](#)
6. Practical implementation of π algorithms. [HVE Practical implementation of PI Algorithms](#)
7. Fast Trigonometric function for arbitrary precision. [HVE Fast Trigonometric calculation for arbitrary precision](#)
8. Fast Hyperbolic functions for arbitrary precision. [HVE Fast Hyperbolic calculation for arbitrary precision](#)
9. Fast conversion from an arbitrary precision number to a string. [HVE Fast conversion from arbitrary precision to string](#)
10. Fast conversion from a decimal string to an arbitrary precision number. [HVE Fast conversion from string to arbitrary precision](#)
11. Fast Computation of Math Constants in Arbitrary Precision. [HVE Fast Computation of Math Constants in arbitrary precision.docx](#)
12. Fast arbitrary precision integer multiplication. [HVE Fast arbitrary precision integer multiplication](#)

The Math behind arbitrary precision

The Math behind arbitrary precision for integer and floating point arithmetic.....	1
Abstract:.....	1
Introduction:.....	1
Change log	2
Introduction to Arbitrary Integer Arithmetic	8
Arbitrary Integer Arithmetic	8
Addition:	8
Subtraction:.....	9
Multiplication:.....	9
Elementary school multiplication	10
Linear convolution multiplication.....	11
No Splitting, No FFT – Just Multiply and Add	11
The Algorithm.....	11
Benefits of Linear Convolution.....	12
Karatsuba multiplication.....	13
Toom-Cook multiplication.....	14
Fast Fourier Transformation (FFT).....	15
Limitations of the FFT	16
NTT method.....	17
NTT Limitations with Primes less than 2^{32}	18
Why do we need multiple primes to make NTT useful?	18
NTT Limitations with Primes less than 2^{64}	18
Schönhage-Strassen Theory vs Practice	19
Fürer's Method	19
Multiplication Performance	19
Division & Remainder:.....	20
Knuth's D Algorithm for division.	22
How it works.....	22
Why is it correct?	22
Efficiency	22
Division Algorithms Beyond Knuth D	23
Burnikel–Ziegler Division	23
Practical recommendation.....	25
Barrett Division.....	25
Montgomery Division.....	25
Hybrid division method using floating point.....	26
Integer Division Performance	26
Recommendation	27
Integer Remainder.....	27
Integer Remainder Performance	27
Proper detection of a carry in arithmetic operations.....	28
Useful functions for integer arithmetic	30
Integer power x^y :.....	30
Integer power x^y modulus z	30
Greatest Common Divisor (GCD)	31
Least Common Multiple (LCM).....	31
Performance of Arbitrary Integer precision:.....	32
Introduction to Arbitrary Floating-Point Arithmetic	34
Arbitrary Floating-point arithmetic:	34
The normalized number for float_precision	35

The Math behind arbitrary precision

Mixed Precision for float_precision.....	35
Rounding control for float_precision.....	36
Addition:	36
Subtraction:	37
Multiplication:.....	37
Division:.....	37
Newton's method for inverse	37
Cubic convergence method for inverse.....	39
Which method for the inverse?	40
Goldschmidt Division method	40
Performance of Arbitrary Floating-point precision:	42
Needed extra functionality	43
Square root:	45
Newton's Method for the square root.....	45
The Initial guess.....	46
Brent's improvement	47
Halley's method for the square root.....	47
Which method for the square root?.....	48
N'th root.....	49
Elementary functions:	51
Exponential functions	52
e^x using the Taylor series	52
Argument Reduction for e^x	53
The issue with arbitrary precision for e^x	54
Finding a reasonable reduction factor for e^x	55
Brent enhancement	56
Guard Digits for e^x	56
Further Improvement of the Taylor series for e^x ?.....	57
Full coefficient scaling.....	58
e^x using Sine Hyperbolic function	59
Argument Reduction in e^x using Sine Hyperbolic	60
Further Improvement of e^x using Sine Hyperbolic?	61
e^x using the binary splitting method.....	61
Argument reduction for e^x , for the binary splitting method.....	62
Finding a reasonable reduction factor for e^x	63
What precision is needed to avoid loss of accuracy?.....	63
Which method to use for e^x ?.....	64
Logarithmic functions:.....	66
Log(x) using the Taylor series.	66
Argument Reduction	67
The issue with arbitrary precision for ln(x)	68
Finding a reasonable reduction factor for ln(x).	69
Guard Digits for ln(x) calculation.....	70
Further Improvement of the methods for ln(x)?	70
Log(x) using the Newton method.	71
Log(x) using the Halley method.	71
Log(x) using the AGM method.....	71
AGM Algorithm.....	72
Log(x) performance	73
Log(x) using the AGM method and multiple threads.	73

The Math behind arbitrary precision

Recommendation for calculating $\log(x)$	73
$\text{Log}_{10}(x)$:.....	74
x to the power of y	74
Constants: e , $\text{Log}_e(2)$, $\text{Log}_e(10)$ & π	75
The constant e	75
AHJ Sale algorithm for e	75
Binary splitting method for e	75
Recommendation for calculating e	76
The Constant $\text{Log}(2)$	77
The Constant $\text{Log}(10)$	80
The constant π	82
Borwein π	82
Brent-Salamin π	82
Binary splitting of the Chudnovsky infinite series.....	83
Recommendation for the Infinite series for π	84
Trigonometric functions:	85
$\text{Sin}(x)$ using Taylor Series	85
The issue with arbitrary precision for $\text{sin}(x)$	87
Finding a reasonable reduction factor for $\text{sin}(x)$	88
Guard Digits for $\text{sin}(x)$	88
Further Improvement of the Taylor series methods?	88
No coefficient scaling	89
Partial coefficient scaling.....	89
Full coefficient scaling.....	89
Partial coefficient scaling.....	89
Full coefficient scaling.....	90
Performance for $\text{sin}(x)$	92
Recommendation for calculating $\text{sin}(x)$	92
$\text{Cos}(x)$ using Taylor Series.....	92
$\text{Cos}(x)$ using double-angle reduction	94
$\text{Cos}(x)$ using full coefficient scaling.....	95
$\text{Cos}(x)$ using $\text{sin}(x)$	95
Performance for $\text{Cos}(x)$	95
Recommendation for calculating $\text{cos}(x)$	96
$\text{Tan}(x)$	96
$\text{Arcsin}(x)$	97
Recommendation for calculating $\text{Arcsin}(x)$	101
$\text{Arccos}(x)$:	101
$\text{Arctan}(x)$ using the Taylor series.....	102
$\text{Arctan}(x)$ using coefficient scaling.....	104
$\text{Arctan}(x)$ using the Euler method.....	105
$\text{Arctan}(x)$ using $\text{Arcsin}()$	106
Recommendation for calculating $\text{Arctan}(x)$	106
Hyperbolic functions:.....	108
$\text{Sinh}(x)$ using $\text{Exp}(x)$	108
$\text{Sinh}(x)$ using the Taylor series	108
The issue with arbitrary precision.....	109
Argument Reduction	110
Finding a reasonable argument reduction factor.....	110
Guard Digits	111

The Math behind arbitrary precision

Further improvements of the method?	111
sinh(x) Performance	114
Recommendation for calculating sinh (x)	116
Cosh(x) using Exp(x)	116
Cosh(x) using Taylor series:	116
Argument Reduction	117
Cosh(x) using double-angle reduction	118
Coshx(x) with full coefficients scaling	118
Cosh(x) Performance	120
Recommendation for calculating cosh(x)	122
Tanh(x).....	122
Recommendation for calculating tanh(x).....	123
Arcsinh(x)	123
Arcsinh(x) direct method	123
Arcsinh(x) using the Taylor series	123
Recommendation for calculating Arcsinh(x).....	124
Arccosh(x).....	125
Arccosh(x) direct method.....	125
Arccosh(x) using the Taylor series	125
Recommendation for calculating Arccosh(x)	126
Arctanh(x)	126
Arctanh(x) direct method.....	126
Arctanh(x) using the Taylor series.....	126
Example Arctanh(0.5)	127
Recommendation for calculating Arctanh(x).....	128
Overall Recommendation for calculating Hyperbolic functions	128
Gamma function.....	129
Algorithm for Gamma computation	130
Lanczos-Spouge method	130
Stirling asymptotic series method.....	131
Integration by parts method	132
Gamma Performance.....	133
Recommendation for the Gamma function	134
The Beta function.....	134
The Error function.....	135
Performance of the error function.....	137
Recommendation for the Error function	138
Lambert W function	138
A Suitable starting point for Lambert W Iteration.....	139
Newton's quadratic method	140
Halley's cubic method	140
Boyd's quadratic method.....	140
Initial performance of the Lambert W function	141
Performance of Lambert W function	141
Recommendation for Lambert W function	142
Riemann Zeta function.....	142
Euler-Mascheroni constant γ	145
Brent-McMillan method	146
Brent enhancement.....	146
The binary splitting method for γ	147

The Math behind arbitrary precision

Performance of the Euler-Mascheroni constant.....	149
Recommendation for the Euler-Mascheroni constant.....	150
Catalan's constant G	150
Ramanujan's method I.....	150
Ramanujan's method II.....	150
Broadhurst series.....	151
The Binary Splitting method for the Catalan constant.....	151
Lupas' Binary Splitting method	152
Guillera's Binary Splitting method.....	153
Pilehood binary splitting method.....	154
Zuniga's binary splitting method.....	156
Comparison of the Catalan Methods.....	156
Catalan Constant Performance.....	157
Recommendation for the Catalan constant	158
Apéry's constant $\zeta(3)$	159
Amdeberhan-Zeilberger series.....	159
Wedeniwski series	160
Zuniga series (v)	160
Zuniga series (vi)	161
Comparison of Apéry's Methods.....	162
Apéry Constant $\zeta(3)$ performance.....	162
Recommendation for the constant $\zeta(3)$	163
The Lemniscate Constant ϖ	164
Guillera's Binary Splitting method.....	166
Comparison of the Lemniscate methods.....	167
Recommendation for the Lemniscate constant ϖ	168
Appendix.....	169
Reference	171

Introduction to Arbitrary Integer Arithmetic

Arbitrary precision integer arithmetic is the foundation of all higher-level computations. Unlike fixed-size machine integers, arbitrary precision integers can grow to millions or even billions of digits, with their size limited only by memory. All subsequent algorithms in floating-point arithmetic and transcendental functions ultimately rely on the ability to add, subtract, multiply, and divide very large integers efficiently.

This chapter presents the mathematics behind integer operations, beginning with the elementary schoolbook methods for +, -, ·, and /, and extending them to faster multiplication algorithms such as Karatsuba, Toom–Cook, FFT, and the Number Theoretic Transform (NTT). These methods trade simplicity for speed, each with its own optimal range of performance. Since multiplication is the backbone of many advanced functions, considerable attention is devoted to analyzing multiplication strategies, carry handling, and performance scaling. Division, remainder, and utility functions such as GCD and LCM are also included, as they are essential in modular arithmetic, cryptography, and the implementation of rational and floating-point arithmetic.

By mastering integer arithmetic at arbitrary size, we establish the building blocks that make the rest of arbitrary precision mathematics possible.

Arbitrary Integer Arithmetic

In integer arithmetic, we use the notation i_n for an n -digit integer number i where n is greater than or equal to zero. In our description, we assume the integer is in base 10 to simplify the description of the underlying math. 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:

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

For Base $\beta=10$, you get:

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

For Base $\beta=2^{64}$, you get:

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

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 from the least significant to the highest.

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

The Math behind arbitrary precision

Algorithm: addition.

```
BASE=264
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 table below

+	$b_m \geq 0$	$b_m < 0$
$a_n \geq 0$	$C = a_n + b_m$	$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 is:

Algorithm: Subtraction

```
BASE=264
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 table below

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

Multiplication:

Multiplication is also trivial

$$a_n \cdot b_m = c_{n+m}$$

Elementary school multiplication

For the multiplication, we divide the case into two scenarios: one where the operand b contains a single limb, $m=1$ (64-bit integer), and one where $m>1$. For $m=1$, we use the forward loop:

Algorithm: Unsigned multiplication with a single limb digit

```
Multiply (vector a, limb b)
BASE=264 // Our Base number is a single limb
carry=0 // Zero the carry
for(i=0..n) // n is the number of limbs in a vector, c is the result vector
    c[i]=(a[i]·b+carry)%BASE
    carry=(a[i]·b+carry)/BASE
if(carry!=0)
    c[n+1]=carry
```

For $m>1$, we repeatedly use the above formula for multiplying a single digit and adding the intermediate results.

Algorithm: unsigned Multiplication of two vectors a and b.

```
Multiply (vector a, vector b)
BASE=264
ck=an·b[0] // Single digit multiplication
for(i=1..m)
    tmp=an·b[i] // Single digit multiplication
    ck=ck+tmp·BASEi // BASEi is the offset
```

Note. The carry handling is omitted in the above algorithm.

Multiplying the intermediate result by $BASE^i$ is easy, as you postfix the temporary result with i number of zeros. The above algorithm has a complexity of $O(n^2)$ and is typically referred to as elementary school multiplication.

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

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

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

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

Linear convolution multiplication

Linear convolution is one of the most straightforward and exact ways to multiply two arbitrary-precision integers. The idea is simple: treat the two input vectors as polynomial coefficients and compute the coefficient-wise product directly, accounting for carry propagation.

Each input vector stores the number in base 2^{64} , using a standard format where `std::vector<uintmax_t> v[0]` is the least significant 64-bit word (limb), and the value of the full number is:

$$A = \sum_{j=0}^{n-1} a_j \cdot 2^{64j}$$

Multiplying two such vectors corresponds to computing the full polynomial product:

$$C_k = \sum_{i+j=k} a_i \cdot b_j$$

This is precisely what linear convolution computes, and the result has a maximum length of $n + m$ limbs, assuming inputs of size n and m digits.

No Splitting, No FFT – Just Multiply and Add

Unlike divide-and-conquer methods (Karatsuba, Toom-Cook) or fast transforms (FFT/NTT), this approach doesn't require splitting the input vectors or transforming them into another domain. It processes the multiplication directly, using nested loops and explicit carry tracking.

This has two advantages:

- Simplicity. It's easy to reason about, debug, and get it correct.
- Accuracy. There's no rounding error, no need for fixed-point tuning, and no modular arithmetic.

The tradeoff is performance: the algorithm runs in $O(n \cdot m)$ time, which becomes expensive as the inputs grow. However, it's still a good choice for small to medium-sized multiplications or as the base case of recursive methods.

The Algorithm

Below is a high-level description of the algorithm in pseudocode. For each combination of $a[j] \cdot b[i]$, we compute a 128-bit product and carefully add it to the result vector using portable 64-bit additions with explicit carry handling.

Algorithm: Linear convolution

```
function linear_convolution(vector vec_a, vector vec_b)
    size_a=size(vec_a)
    size_b=size(vec_b)
    for(i=0..size_b)
        for(j=0..size_a)
            tmp = umul64(vec_a[j], vec_b[i]); // Return 128bit result
            // portable 64-bit add with explicit carries
            s1 = tmp.lo + res[i + j];
            c1 = (s1 < tmp.lo); // carry from first add
```

The Math behind arbitrary precision

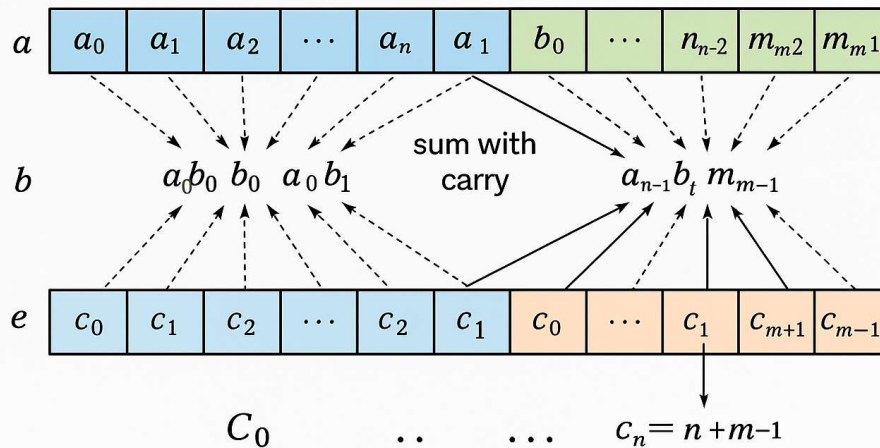
```

s2 = s1 + carry;
c2 = (s2 < carry);    // carry from second add
res[i + j] = s2;
carry = tmp.hi + c1 + c2;
res[i + n] += carry;
// propagate carry to res[i + size_a] and beyond if needed
return res

```

Linear Convolution

$$A = \sum_{j=0}^{n-1} a_j 2^{64j}, \quad B = \sum_{i=0}^{m-1} b_i 2^{64i}$$



Above is a graphic depicting what is going on.

Benefits of Linear Convolution

Linear convolution is the foundation of arbitrary-precision multiplication. Even though it has quadratic time complexity, it is simple to implement, exact, and easy to optimize for small input sizes.

Key points:

- 64-bit limbs are optimal on 64-bit CPUs.
- Explicit carry handling avoids undefined behavior and ensures correctness.
- Carry must be fully propagated after each row of multiplication.
- Pre-sizing the result vector avoids reallocation.
- Avoid 32-bit decompositions unless necessary for special hardware or compatibility.

This method may not win speed races for large inputs, but it remains a reliable workhorse and the ideal base case for recursive multiplication strategies.

Karatsuba multiplication

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 multiplications and three additions/subtractions instead of the usual four multiplications.

Karatsuba's result was the first clear sign that long multiplication is not a fundamental limit. He began by writing each number in two roughly equal parts, say $x=x_1B^m+x_0$ and $y=y_1B^m+y_0$, where B is the limb base and is about half the digit-count. The standard expansion has four half-size products. x_1y_1 , x_1y_0 , x_0y_1 , and x_0y_0 , which together form xy .

Karatsuba noticed that only one extra product can recover the two cross terms. Compute

$$z_0=x_0y_0, z_2=x_1y_1, z_1=(x_1+x_0)(y_1+y_0).$$

The sum z_1 contains every cross contribution once, so

$$(x_1y_0+x_0y_1)=z_1-z_2-z_0.$$

With these three products and two linear-time add-subtract steps, the whole product is reconstructed as

$$xy=z_2B^{2m}+(z_1-z_2-z_0)B^m+z_0.$$

Replacing four half-size multiplies by three shifts the cost recurrence from $T(n)=4T(n/2)+O(n)$ to $T(n)=3T(n/2)+O(n)$. The solution drops from quadratic growth to $n^{\log_2 3} \sim n^{1.585}$. The savings at a single split are modest. Yet, when the process recurs, the benefit compounds, and once each half spans a few dozen machine limbs, Karatsuba outperforms schoolbook multiplication for multiplication with a higher number of decimals. This divide-and-conquer idea set the stage for every faster method that followed, from Toom-Cook to the FFT-based algorithms used in today's very large integer libraries.

Algorithm: Karatsuba multiplication

```
function karatsuba(a,b)
  if(a<=KARATSUBA_CUTOFF&& b<=KARATSUBA_CUTOFF)
    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*102*m)+(z1-z2-z0)*10m+z0
```

Karatsuba algorithm reduces the complexity of multiplication from $O(n^2)$ to $O(n^{1.58})$

Toom-Cook multiplication

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 three parts (Toom-Cook-3); however, GNU arbitrary precision also offers four-part 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<=TC3_CUTOFF|&&b<= TC3_CUTOFF)
        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*10m
```

```
I2=i2*102m
I3=i3*103m
I4=i4*104m
result=i0+i1+i2+i3+i4
if(sign(a)*sign(b)<0)
    result=-result
return result
```

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, the method consists of converting the two numbers a_n and b_m via a series of Fourier transformations, multiplying them together, and then doing the inverse Fourier transformation of the result back to the digital domain.

FFT complexity is $O(n \cdot \log(n))$, making it preferable over the other multiplication methods.

Algorithm: FFT multiplication

```
function multiplyFFTRadix2(A, B, chunkBits):
```

```
    // 1. Validate chunk size
```

```
    if chunkBits  $\notin$  {4, 8, 16}:
```

```
        error "chunkBits must be 4, 8, or 16"
```

```
    // 2. Unpack limbs into "digits" (base = 2chunkBits)
```

```
    fa  $\leftarrow$  unpack(A, chunkBits) // returns complex vector of length lenA
```

```
    fb  $\leftarrow$  unpack(B, chunkBits) // returns complex vector of length lenB
```

```
    // 3. Choose FFT length N = next power of two  $\geq$  (lenA + lenB - 1), then double for safety
```

```
    need  $\leftarrow$  len(fa) + len(fb) - 1
```

```
    N  $\leftarrow$  1
```

```
    while N < need:
```

```
        N  $\leftarrow$  N << 1
```

```
    N  $\leftarrow$  N << 1
```

```
    resize fa and fb to N (zero-pad)
```

```
    // 4. Forward FFT on both
```

```
    fft2(fa, invert = false)
```

```
    fft2(fb, invert = false)
```

```
    // 5. Pointwise multiply
```

```
    for i in 0..N-1:
```

```
        fa[i]  $\leftarrow$  fa[i] * fb[i]
```

```
    // 6. Inverse FFT and scale
```

```
    fft2(fa, invert = true) // divides by N
```

```
    // 7. Repack digits and propagate carries
```

```
    result  $\leftarrow$  packAndCarry(fa, chunkBits)
```

```
    // 8. Trim leading zero-limbs
```

The Math behind arbitrary precision

```
while size(result) > 1 and last(result) == 0:  
    remove last(result)  
  
return result
```

Note: the FFT2 function is the Fast Fourier Transform with Radix 2.

Limitations of the FFT

FFT uses floating-point arithmetic using the double type in C++. In the initial step, you need to map the vector of 64-bit binary digits into a vector of complex< doubles>. You do that by splitting each 64-bit binary digit into eight bytes and then converting each byte into a complex<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 [4], they give a formula for the number of decimal digits n that 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 bits in a C++ double)

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

For example, a byte (8-bit) has a sample size of $2^8 = 256$. Where k is “a few”. Let’s choose $k = 2$. We get that for $n = 100,000,000$ (100 million), $52 < 53$, which is true.

If we solve the above equation for n , we get an n of approximately 175M digits.

Unfortunately, k as two is insufficient, as we encounter random errors in the multiplication result when multiplying by 125M+ digits. Instead, I recommend using k as a factor of 2.15, which yields a maximum multiplication size limit of approximately 116 million digits. That leads to the question of what to do if you need more digits in multiplication. A simple solution is to reduce the sample size to 4 bits, rather than 8 bits. Using the above formula again, you can multiply a maximum of 17.8 billion digits. If that is still insufficient, you can do a 2-bit sample size and get a limitation of 229 billion digits. Every time you halve the sample size, the memory requirements increase by a factor of four, thereby reducing the overall performance of FFT multiplication. If you go the other way and use a sample size of 16 bits, you get a limitation of approximately 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 it supports floating-point arithmetic above 64 bits. IEEE 754 specifies an extended 80-bit version, sometimes referred to as long double. (Not all compiler supports it, so the author's arbitrary precision packages only support the standard 64-bit double)

Maximum operand size for multiplication as a function of sample size in the 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

The Math behind arbitrary precision

4-bit	17.8B	20T
2-bit	229B	280T

*) Not supported on all compilers and systems

Notice that a 32-bit sample will overflow the double 53-bit mantissa

NTT method.

The Number Theoretic Transform (NTT) is the modular arithmetic counterpart of the Fast Fourier Transform (FFT). While the FFT works over the complex numbers, the NTT performs similar convolution operations using only integers modulo a prime. The technique builds on concepts from number theory, such as primitive roots of unity modulo a prime. It is ideal for applications that require exact arithmetic, particularly in cryptography and arbitrary-precision multiplication.

The first notable uses of NTT-like techniques in convolution go back to the 1960s and 1970s, when the Cooley–Tukey FFT gained prominence. Over time, NTT gained popularity in contexts where floating-point roundoff errors could not be tolerated, such as in Cryptographic schemes (RSA, lattice cryptography), Modular polynomial multiplication, and large integer multiplication in libraries like FLINT or implementations of Schönhage–Strassen variants.

Modern usage often involves combining multiple NTTs over different primes and reconstructing the final result via the Chinese Remainder Theorem (CRT).

Algorithm for the NTT 32-bit version using either 3 or 6 primes.

```
function multiplyNTT32(A, B, useSixPrimes = false):
  // 1. Split 64-bit limbs into 16-bit chunks (little-endian)
  aC ← unpackToChunks(A, 16) // returns vector<uint32_t>
  bC ← unpackToChunks(B, 16)

  // 2. Choose transform length N = next power-of-2 ≥ (len(aC) + len(bC))
  need ← len(aC) + len(bC)
  N ← 1
  while N < need:
    N ← N << 1

  // 3. Perform NTT convolution under each prime
  // – the first three primes are always used
  r0 ← nttConvolution(aC, bC, primeIndex = 0, N)
  r1 ← nttConvolution(aC, bC, primeIndex = 1, N)
  r2 ← nttConvolution(aC, bC, primeIndex = 2, N)

  if useSixPrimes:
    r3 ← nttConvolution(aC, bC, primeIndex = 3, N)
    r4 ← nttConvolution(aC, bC, primeIndex = 4, N)
    r5 ← nttConvolution(aC, bC, primeIndex = 5, N)

  // 4. Combine residues via CRT
  if useSixPrimes:
```

The Math behind arbitrary precision

```
// pairwise CRT to form three 62-bit residues
c0 ← crtCombine2(r0, r1, primes 0&1)
c1 ← crtCombine2(r2, r3, primes 2&3)
c2 ← crtCombine2(r4, r5, primes 4&5)
// three-way CRT merge into 64-bit coefficients
coeff ← crtCombine64(c0, c1, c2)
else:
  // in-place three-prime CRT merge
  coeff ← crtCombine3(r0, r1, r2)

// 5. Repack coefficients into 64-bit limbs with carry
result ← packAndCarry(coeff, chunkBits = 16)

// 6. Trim any leading zero limbs
while size(result) > 1 and last(result) == 0:
  pop_back(result)

return result
```

NTT Limitations with Primes less than 2^{32}

When using primes under 2^{32} , the modulus fits in a `uint32_t` (32-bit), which means everything, including multiplication, reduction, and root-of-unity operations, can use native 64-bit arithmetic for intermediate values without overflow.

Why do we need multiple primes to make NTT useful?

One 32-bit prime can only represent products up to $\sim 2^{32}$, which is insufficient for serious multiplication. A product of two large integers can be thousands or millions of bits. However, when performing a convolution modulo multiple 32-bit primes, we need to reconstruct the result with the CRT.

Practical choices:

1 prime → Good for modular convolution (e.g., inside NTT-based polynomial multiplication) but not useful in practice.

3 primes → Covers 96-bit dynamic range: good enough for operands up to ~ 100 M decimal digits. However, three selected primes represent a slightly smaller number of bits, making the maximum operand around 40M digits.

From a practical point of view, six primes can increase the operand size to approximately 160M decimal digits, but at the expense of more overhead and with a larger range.

More than six primes can extend the limitation beyond the 160M threshold. However, it will become increasingly more challenging to find useful primes and require more processing time. The downside of adding more primes is that each additional prime adds a complete transform pass + CRT recombination stage.

NTT Limitations with Primes less than 2^{64}

Using primes slightly less than 2^{64} is appealing because you can represent a broader range in a single modulus. But there are drawbacks:

Pros:

- Only one prime → No CRT step needed.

The Math behind arbitrary precision

- Each limb can be larger (up to 64 bits).
- Better for very large operands (100M+ digits).

Cons:

- Arithmetic becomes trickier: you need 128-bit intermediates for multiplication and modulo.
- Only a few suitable primes exist below 2^{64} that support large enough roots of unity.
- Modular reduction (e.g., Barrett or Montgomery method) becomes more complex and heavier.

As a result, three primes under 2^{32} are often preferred for raw speed, even though a single 64-bit prime would be simpler.

Schönhage-Strassen Theory vs Practice

The Schönhage-Strassen algorithm, published in 1971 by Schönhage and Strassen, represents a theoretical step forward in computational number theory. By achieving $O(n \cdot \log(n) \cdot \log(\log n))$ complexity for integer multiplication. It remained the asymptotically fastest known method for over three decades. However, going from theoretical to practical implementation reveals computational compromises between academic theory and real-world software engineering. An interesting reference is found in [27]

The algorithm's theoretical foundation rests on the insight that integer multiplication can be transformed into polynomial multiplication through careful encoding, and polynomial multiplication can be computed efficiently. However, the issue lies in the details of implementation, where the mathematical framework collides with the realities of finite precision arithmetic, memory hierarchies, and the fundamental chicken-and-egg problems that arise when trying to bootstrap efficient arithmetic operations.

Fürer's Method

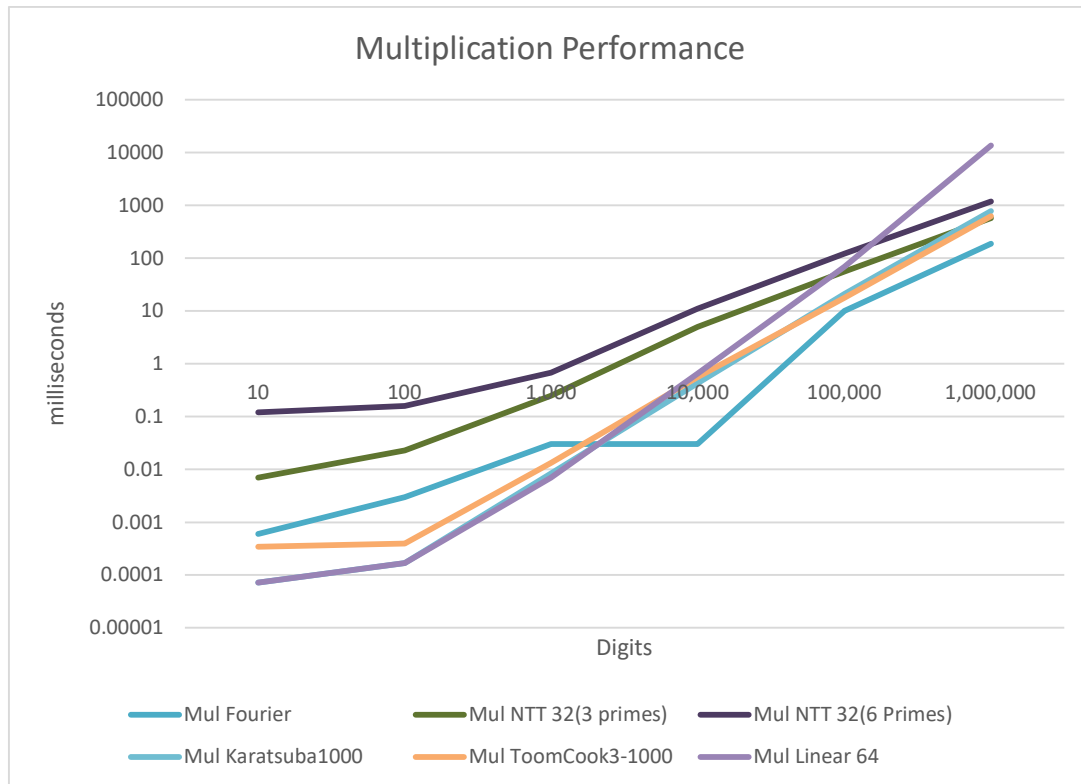
In 2007, Martin Fürer published an algorithm achieving $O(n \log n 2^{O(\log n)})$ complexity [29], technically surpassing the Schönhage-Strassen asymptotic bound. However, the improvement only becomes apparent for numbers with more bits than there are atoms in the observable universe, highlighting the disconnect between theoretical advances and practical relevance.

More recently, Harvey and van der Hoeven achieved $O(n \log n)$ complexity using a refined analysis of the FFT over complex numbers; however, the crossover point still lies far beyond practical computation ranges.

Multiplication Performance

Below is the result of measuring performance for multiplying numbers with different operand sizes from 10 decimal digits to 1 million decimal digits.

The Math behind arbitrary precision



As we can see from the performance chart, multiplying using a 64-bit linear convolution is by far the fastest method below approximately 4,000 decimal digits. (Karatsuba has similar performance as linear convolution from 1000 to 5,000) After that, the optimized Fast Fourier Transformation (special FFT radix 2) takes over and maintains its best performance, extending to over 1 million digits in this performance chart. The FFT radix 2 is an optimized version from [4] that avoids the complex arithmetic library and bases the transformation solely on regular 64-bit floating-point arithmetic. After the optimized FFT Fourier, the ToomCook3 becomes slightly faster than Karatsuba above the 10,000-digit mark.

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 let's call it d_j . For shorthand, this is sometimes written as:

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

If b_m is a single digit, we do it in schoolbook manner by dividing the single digit b_0

The Math behind arbitrary precision

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=264
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 eight digits or a_8 magnitude is in the range of 10^8 and b_m is a two-digit number of magnitude 10^2 then you will have to loop through the subtraction approximately 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 $b_m * \text{BASE}^{n-m}$. If $b_m * \text{BASE}^{n-m}$ 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 $\text{BASE} * \text{BASE}^{n-1-m}$. Repeating this process, you get an approximation. The number of loops and operations of the multiplication and subtraction is $\sim 2(n-m)$. We cannot always assume that $b_m * \text{BASE}^{n-m}$ is less than a_n in which case we subtract $b_m * \text{BASE}^{n-m-1}$ instead. This led to a worst-case scenario that is 10 times higher than the approximation we found before, or $\sim 20(n-m)$. 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, which is to employ the iterative method for division, similar to the method 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 (described 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 logarithmic scaling of the number of operations, which is, of course, much faster.

The Math behind arbitrary precision

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

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

This align–subtract–shift process is essentially the same principle as long division, or more formally, Knuth’s Algorithm D. In Knuth’s version, each step estimates a quotient digit by looking at the top one or two limbs of the dividend and divisor, then corrects if the estimate is too large. My description here differs slightly: instead of explicitly *estimating quotient digits*, the method is phrased as repeated subtraction of scaled divisor blocks. Both approaches yield the same quadratic complexity, $O(nm)$, and are equivalent in spirit. Knuth’s digit-estimation strategy is more precise and a lot faster, and avoids the occasional overshoot that can occur in the block-subtraction description.

Knuth's D Algorithm for division.

Knuth’s Algorithm D is a classical method for dividing large whole numbers. It is widely used in multiple-precision arithmetic libraries as the standard baseline method before more advanced techniques are applied. The goal is to compute a quotient and a remainder when one large number is divided by another.

How it works

The algorithm works by processing the numbers in blocks called limbs, where each limb is a fixed-size word (for example, 64 bits). The divisor is adjusted, or normalized, so that its most significant limb starts with a one-bit. The dividend is shifted by the same amount so that both numbers remain consistent.

Once normalized, the algorithm proceeds from the most significant end of the dividend toward the least significant end. At each step, it makes an educated guess at one digit of the quotient. This guess is based on just the leading two limbs of the current dividend section and the top limb of the divisor. The guess may be a little too high, but the algorithm quickly corrects it if necessary.

After the trial digit is chosen, the algorithm multiplies the divisor by this digit and subtracts the result from the current section of the dividend. If the subtraction results in a negative value, the digit is reduced by one, and the divisor is added back. This ensures that the quotient digit is always correct.

This process is repeated for each position until the entire quotient has been built. In the end, the remaining part of the dividend is adjusted back by the original normalization shift, giving the true remainder.

Why is it correct?

The key reason the method works is that the initial guess for each quotient digit is never more than one off. That means a single correction step is always enough to find the correct value. The repeated loop guarantees that after each step, the section of the dividend under consideration is strictly smaller than the divisor, so the algorithm makes steady progress.

Efficiency

The efficiency of Algorithm D depends on the size of the numbers. Each step requires multiplying the divisor by a single-limb digit and subtracting it from part of the dividend. If

The Math behind arbitrary precision

the dividend has n limbs and the divisor has m limbs, the total cost is roughly proportional to n times m . For small to large numbers, this is perfectly adequate, but for very large numbers, more advanced methods, such as Burnikel–Ziegler or Newton's reciprocal division, are usually faster.

Even so, Knuth's Algorithm D remains the workhorse for many practical cases. It is simple, reliable, and provides the foundation upon which faster algorithms are built.

Pseudo-Algorithm outline (simplified)

```
Algorithm Knuth_Division(U[0..n-1], V[0..m-1]):
  d ← leading_zeros(v[m-1])
  U ← U << d
  V ← V << d
  U[n] ← 0
  for j from n-m downto 0:
    (q̂, r̂) ← div((U[j+m]*B + U[j+m-1]), v[m-1])
    while q̂*v[m-2] > r̂*B + U[j+m-2]:
      q̂ ← q̂ - 1
      r̂ ← r̂ + v[m-1]
      if r̂ ≥ B then break
    subtract q̂*V from U[j..j+m]
    if a borrow occurred:
      q̂ ← q̂ - 1
      add V to U[j..j+m]
    Q[j] ← q̂
  R ← (U[0..m-1] >> d)
  return (Q, R)
```

Division Algorithms Beyond Knuth D

There are several interesting choices to consider. One of them is the Burnikel-Ziegler, another one is the Barrett and Montgomery division.

Burnikel–Ziegler Division

Long division (Knuth's Algorithm D) has a quadratic cost $O(n \cdot m)$, which becomes inefficient when both the dividend and divisor have thousands or millions of limbs. To improve this, Burnikel and Ziegler (1998) proposed a divide-and-conquer approach that reduces a large division into smaller division problems plus multiplications. Its asymptotic cost is $O(M(n))$, where $M(n)$ is the cost of multiplying two n -limb integers. With FFT multiplication, this approaches quasi-linear complexity.

Basic idea

The Burnikel–Ziegler (BZ) method accelerates division by splitting the operands into blocks of size $2k$ limbs (where k is chosen so that the block boundaries align with machine words). Instead of trial subtraction across the whole numbers, it works recursively on the high blocks, correcting with the lower blocks.

The Math behind arbitrary precision

1. Split the dividend and divisor into blocks of size $2k$.
2. Recursively divide the high part of the dividend by the high part of the divisor, producing a partial quotient.
3. Multiply back the partial quotient by the divisor block and subtract from the dividend.
4. Recurse on the reduced remainder with the next block.
5. Continue until all blocks are processed.

Each recursive call handles roughly half-sized problems, giving the recurrence:

$$T(n) = 2T(n/2) + O(M(n)),$$

which solves to $T(n) = O(M(n))$.

Pseudo-algorithm (simplified)

```
function BZ_Divide(A, B):
  # A = dividend with n limbs
  # B = divisor with m limbs

  if n small: return LongDivision(A, B)

  # choose block size k ≈ m/2, rounded to power of 2
  split A into A_high || A_low
  split B into B_high || B_low

  # Step 1: divide high block
  (Q1, R1) = BZ_Divide(A_high, B_high)

  # Step 2: subtract Q1 * B from A_high||A_low
  A' = (A_high||A_low) - Q1 * B

  # Step 3: divide reduced remainder by B_high
  (Q2, R2) = BZ_Divide(A', B_high)

  # Step 4: assemble quotient
  Q = (Q1 shifted by blocksize) + Q2
  R = R2
  return (Q, R)
```

This outline omits essential details, such as normalization, the selection of block sizes, and correction steps for subtraction underflows. In real implementations, recursion bottoms out in a standard long division when the operands are small enough.

Comparison with long division (Knuth D)

- **Complexity**
 - Knuth D: $O(n \cdot m)$. For equal-sized operands, $O(n^2)$.
 - Burnikel–Ziegler: $O(M(n))$. With FFT multiplication, it is quasi-linear in time.
- **Thresholds**
 - For small operands (up to 100 thousand limbs), long division is faster due to lower overhead.

The Math behind arbitrary precision

- For very large operands (millions of limbs), Burnikel–Ziegler dominates.
- **Implementation**
 - Knuth D is conceptually simple and easy to implement.
 - Burnikel–Ziegler requires careful block splitting, normalization, and recursive structure. Libraries such as GMP use it selectively for medium- to large-sized inputs.

Limitations

While attractive in theory, Burnikel–Ziegler is not a self-contained general division method. It only works efficiently when the dividend and divisor meet strict size and alignment conditions:

- The divisor must fit neatly into a block size that is a power of two in limbs.
- The dividend must not be too much larger than the divisor.
- Both must be normalized before splitting.

If these conditions are not satisfied, the algorithm may fail outright or require so many correction steps that it becomes slower than long division. For this reason, any practical implementation must provide a fallback, typically Knuth’s Algorithm D, whenever the preconditions are not met.

Practical recommendation

Burnikel–Ziegler is a specialized optimization. It can be helpful in high-performance libraries as part of a hybrid strategy. Still, it is not suitable as the only division algorithm in a general-purpose arbitrary-precision system.

Barrett Division

Barrett’s method was initially introduced to accelerate modular reduction in cryptography. The idea is to precompute an approximation of the reciprocal of the denominator. Once that reciprocal is available, divisions can be replaced by multiplications and shifts, which are much cheaper at high precision.

The limitation is that the precomputation cost is significant. If you only need to perform a single division, Barrett is not faster. Its strength appears when you are dividing many different numerators by the same fixed denominator. In that case, the one-time precomputation pays off, and every subsequent division is cheap.

Practical takeaway: Barrett is not a general replacement for long division. It is only efficient in workloads that repeatedly use the same denominator.

Montgomery Division

Montgomery’s method is even more specialized. It was designed for modular arithmetic, especially modular multiplication, where the modulus is fixed and performance is critical (cryptographic routines like RSA, ECC, etc.). Montgomery’s trick is to represent numbers in a special “Montgomery form” that makes modular reduction easy after multiplication.

The Math behind arbitrary precision

The catch is that entering and leaving Montgomery form requires additional conversions, and the method only works when the modulus is fixed for many operations. For a single general integer division, Montgomery offers no benefit and often adds overhead.

Practical takeaway: Montgomery reduction is a cornerstone of modern modular arithmetic in cryptography, but it is not helpful for arbitrary one-off divisions. Like Barrett, it only shines when the modulus is fixed and reused.

Hybrid division method using floating point

Although integer division can be performed entirely using the above methods, an interesting alternative is available if arbitrary-precision floating-point arithmetic is available. In this approach, both the dividend and divisor are first converted into floating-point numbers with sufficient precision. The quotient is then computed using Newton iteration (which has quadratic convergence), and the result is converted back into an integer quotient and remainder.

This hybrid method can outperform pure integer long division for very large operands, because the number of operations scales with the cost of multiplication rather than with the number of digits. The drawback is that it introduces a dependency on floating-point arithmetic, which may not be desirable if one wishes to keep integer arithmetic self-contained. For this reason, it is best mentioned as an optional alternative rather than a primary algorithm.

Integer Division Performance.

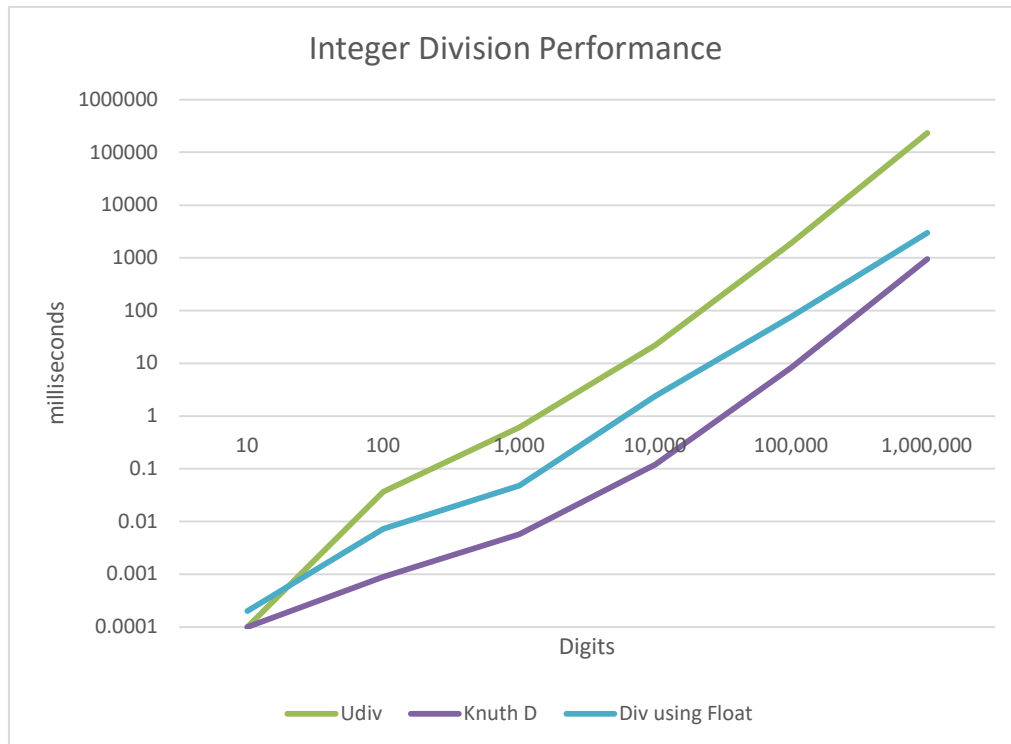
To evaluate different division strategies, three methods were compared across a range of operand sizes:

- Udiv: an early home-made division routine
- Knuth D: the classical long division algorithm described by Knuth
- Division using Newton iteration using floating-point arithmetic

The test measured the time required to divide large integers, with sizes ranging from 10 to one million decimal digits. In each test, the denominator was chosen to be about one-third the length of the numerator. Other ratios between the numerator and denominator were also tried with similar results. The results highlight several points. At small sizes, all three methods are essentially instantaneous. As the digit count grows, the differences become pronounced.

Knuth D is consistently the fastest across the tested range. The float-based Newton method, while asymptotically better for very large balanced operands, is slower here because each iteration requires full-precision multiplications. The older Udiv routine performs poorly in comparison, especially as the size grows.

The Math behind arbitrary precision



Recommendation

- Knuth D offers excellent practical performance for operands up to one million digits, particularly when the denominator is smaller than the numerator.
- The float-based Newton method has heavier overhead and only begins to narrow the gap at the largest tested sizes.
- A simple routine (Udiv, as described at the start of this chapter) is far less efficient and should be avoided for serious use.
- In practice, a hybrid strategy works best: use Knuth D for modest or unbalanced sizes, and reserve Newton or Burnikel–Ziegler for very large and balanced divisions.

Integer Remainder

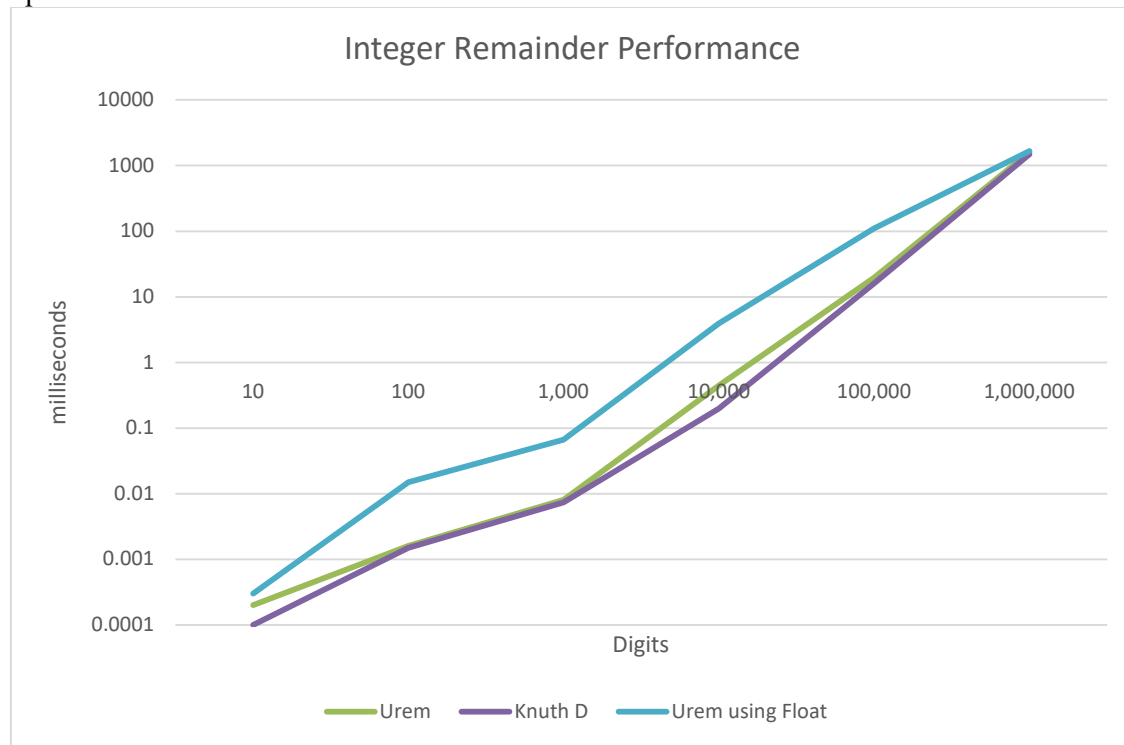
As a byproduct of division, you also get the remainder, which is easy to compute from a division operation. So, we should expect similar performance. Generally speaking, the algorithm mentioned for integer division also exists for the remainder computation. We also have the previous chapter, Knuth's D algorithm, and the Hybrid remainder using float division available.

Integer Remainder Performance

The first green curve is the remainder computed as $\text{rem} = a - (b \cdot (a/b))$ but using Knuth's division function. The Knuth D is the Knuth D algorithm for the remainder, and the last is the remainder computed using floating-point division. We observe the same pattern, where Knuth's algorithm is superior for smaller to larger size remainder operations. However, using

The Math behind arbitrary precision

the floating-point algorithm is approaching the performance of Knuth's for very large operators.



Proper detection of a carry in arithmetic operations

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 simply dividing the operations performed by 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, needs to be less than 2^{64} . Since you can define the arbitrary precision arithmetic with any BASE you can choose a “nice” number less than 64.e.g. $\text{BASE}=2^{60}$. 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 $\text{BASE}=2^{60}$ it will require a vector of 56 64-bit integers to store the number. However, if $\text{BASE}=2^{64}$ it will require only 52 64-bit integers to store the number. This means that a BASE of 2^{60} will require $\sim 7.7\%$ 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 overflows, 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
```

The Math behind arbitrary precision

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

```
a=6, b=4           // Will result in overflow
c=a+b=6+4=10      // The one digit is discarded and stroked out
if(c<a)           // 0<6
    carry=1       // Yes, carry was detected
else
    carry=0       // No, carry 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 additions: 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 (the maximum single digit), the first addition $tmp=a+b=9+9=18$ with a carry detected yields a result of 8. Now, carry can have either a zero or one value, and even with carry = 1, the next addition ($tmp=tmp+carry$) can at most 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 $tmp = 9 + carry = 9 + 1 = 10$. Since the result 0 is less than either 9 or 1, a carry is detected that can be propagated forward.

An algorithm for carry detection in addition

```
BASE=264
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 the bit size environment (32-bit or 64-bit).

Useful functions for integer arithmetic

Several useful functions are typically provided 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 by x , y times. Instead, we use the entity repeatedly when y is an even number:

$$x^y = x^{\frac{y}{2} + \frac{y}{2}} = x^{\frac{y}{2}} x^{\frac{y}{2}} = (x \cdot x)^{\frac{y}{2}} \quad (1)$$

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., 21,000,000 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, which 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
```

The Math behind arbitrary precision

```
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 package.

Algorithm: gcd(a,b)

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

The above version has the drawback of using the % operator, which is notoriously time-consuming in arbitrary precision. Instead, it is better to use the “binary” version, which only requires subtraction and shifting, both of 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.

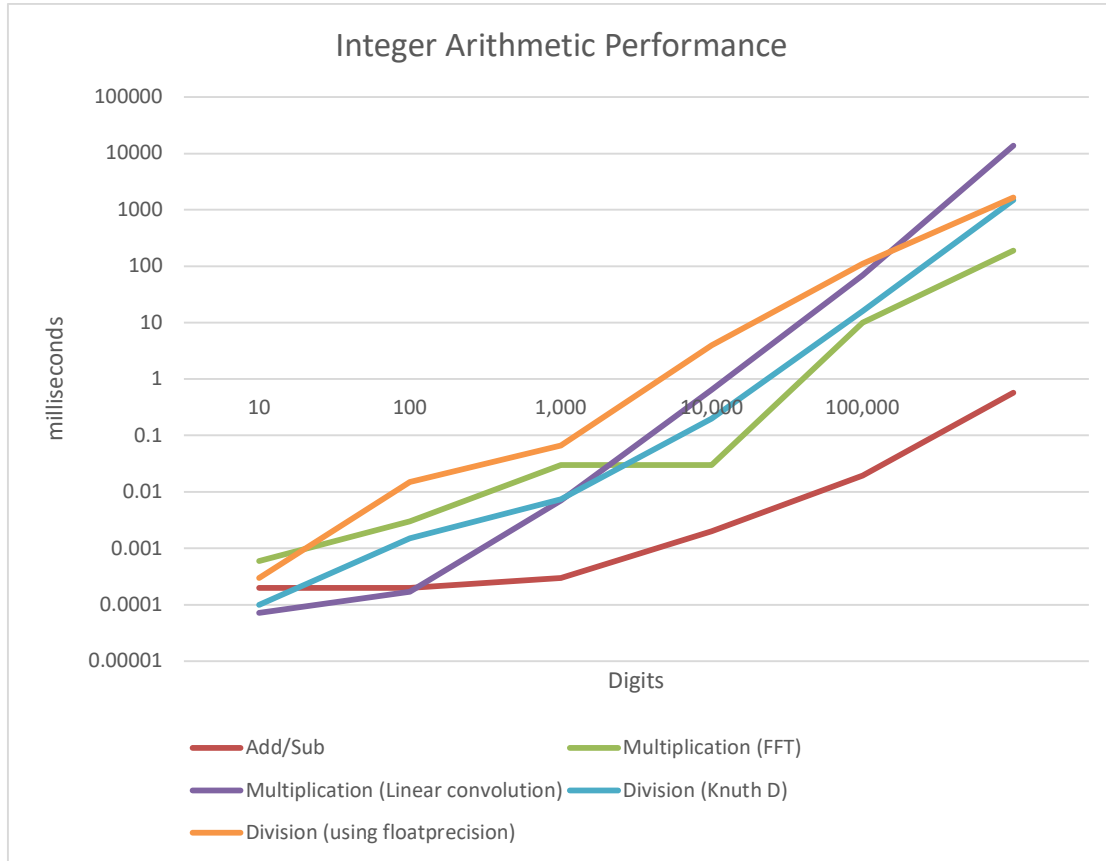
Algorithm lcm(a,b)

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

The Math behind arbitrary precision

Performance of Arbitrary Integer precision:

The following shows the performance of integer precision. The y-axis is a logarithmic scale of milliseconds. The X-axis is the number of digits. For addition/subtraction/multiplication, both operands are of the same number of digits. For the division, the denominator has half as many digits as the dividend.



Integer Operation in milliseconds vs. number of digits

It is not a surprise that addition and subtraction are the fastest operations. For multiplication, the use of multiplication via linear convolution is the fastest multiplication for smaller numbers up to approximately 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 off around the 5,000-6,000 digit mark. Division and remainder, as expected, are the most time-consuming tasks and are often slower than multiplication. The lesson learned is that you should strive to avoid division whenever possible. This knowledge comes into play when using the Taylor series for exponential, logarithm, Trigonometric, and Hyperbolic functions, where we will employ a technique called coefficient scaling to reduce 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
-------------------	--------

The Math behind arbitrary precision

Digits	100	1,000	10,000	100,000	1,000,000
Add/Sub	1	1	1	1	1
Multiplication (FFT)	15	100	15	510	331
Multiplication (Linear convolution)	1	23	325	3,520	23,891
Division (Knuth D)	8	25	99	816	2,604
Division (using floatprecision)	75	221	1,962	5,663	2,898

We observe that FFT multiplication is the preferred method for calculations involving numbers 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 number of digits in the integer.

The surprise is division. We expected it to be slower, but it is many times slower than any of the other operations (+-*).

Introduction to Arbitrary Floating-Point Arithmetic

While integers provide the foundation, many applications require fractional values, exponents, and transcendental functions. Arbitrary precision floating-point arithmetic extends the ideas of IEEE 754 floating-point numbers but removes the limitation of fixed bit-width. Each number is represented by a sign, a normalized mantissa, and an exponent, with user-selectable precision that can reach millions of digits.

Floating-point arithmetic relies on integer operations internally, but introduces additional challenges such as normalization, rounding, precision alignment, and exponent management. This chapter begins with the core operations of addition, subtraction, multiplication, and division. Then it extends them to more advanced routines, including square roots, nth roots, exponential and logarithmic functions, trigonometric and hyperbolic functions, and finally special functions such as the Gamma, Beta, Zeta, and Lambert W functions.

The focus is not only on correctness but also on performance. Techniques such as argument reduction, Newton and Halley iterations, coefficient scaling, and binary splitting are introduced to make calculations feasible at extreme precision levels. By combining these methods, arbitrary-precision floating-point arithmetic becomes a practical tool for research, numerical analysis, and the computation of mathematical constants at unprecedented scales.

Arbitrary 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 parts are stored as decimal strings (usually in base 10) or as binary numbers (base 2, as the native CPU supports). Typically, you can store a *float_precision* number as normalized binary numbers in the format:

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

For the special case where the floating-point number is zero, if the sign is either +1 or -1, 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 indicates that the exponent contains p digits.

In addition, β is the base, typically base 2, 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.

The Math behind arbitrary precision

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 numbers 1-9 in base 10) as its integer part, we can store the entire number in a single 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. It is also a design option to either mark the leading one digit in front of the fraction part (i_1) as explicit or implied, as in the IEEE 754 standard for floating-point variables. If implied, you also have to define how zero is stored to avoid it being mistakenly taken as a one. In the author's arbitrary precision library, I have chosen to explicitly represent the single leading integer part in the first limb of the floating-point number, making normalization and other operations on the number easier.

For base 10, you get:

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

Example: 1.234E2

$I_1=1$

$F_3=234$

$E_p=2$

The exact number in base 64 is:

$I_1=0x1$

$F_n=0xed9999999999a000$

$E_p=0x6$

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 operation would be returned as a normalized number.

Mixed Precision for float_precision

Since our floating-point numbers can have different precisions, and we, of course, allow mixed precision for our arithmetic operators, it is essential to understand how 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. If necessary, the expression on the right-hand side is rounded accordingly. For binary operator like $+, -, *, /, \%$. Mixed precision is handled by always aligning the arguments on both sides of

The Math behind arbitrary precision

the operator to the argument with 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' exponents 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 have different precisions. This is no different than in the standard C programming language, where 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. For example, 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 $9.5E0+2.4E0=11.9E0$. However, 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 table below, 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 $
$a_n < 0$	$C=-(a_n +b_m)$	$C=-(a_n + b_m)$

The Math behind arbitrary precision

Subtraction:

It is done using the addition function since $a-b$ is the same as $a+(-b)$. We change the sign on b and then call the addition function. There is no need to implement the same functionality as for addition. Additionally, you need to handle negative arguments of any of the operands anyway, and there is no need to replicate the same code to handle 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 and [40]. Before we call the FFT function, we strip off the sign and exponent and then use the resulting sign as follows.

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

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

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

Now, since we compute the FFT using IEEE754 arithmetic, and we know that for a 64-bit floating point, we have 53 bits in the floating-point mantissa (including the hidden implicit one bit), we can then derive a bound for how large a number we can multiply using only a 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 calculate the inverse of $b = (1/b)$ quickly. This same issue affects many microprocessors, or early RISC (Reduced Instruction Set Computer) CPUs, which lacked hardware support for the division operator. Instead, they use either a Newton iteration or the Goldschmidt method. The Newtons employed the following algorithm to calculate $1/b$. 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 $1/b$:

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

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

The Math behind arbitrary precision

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

Algorithm 1

Traditionally, this method has been used due to its simplicity.

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

Applying it to the Newton method, you get:

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

$$\begin{aligned}x_{n+1} &= x_n - \frac{y - \frac{1}{x_n}}{\frac{1}{x_n^2}} \Rightarrow \\x_{n+1} &= x_n - x_n^2 \left(y - \frac{1}{x_n} \right) \Rightarrow \\x_{n+1} &= x_n - x_n(x_n y - 1) \Rightarrow\end{aligned}$$

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

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 built-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 examine how our arbitrary precision number is constructed. 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.

The Math behind arbitrary precision

$$\frac{1}{b} = \frac{1}{i_1 \cdot f_n 2^{ep}} = \frac{1}{i_1 \cdot f_n} 2^{-ep} \quad (4)$$

We can extract the exponent portion and find the inverse. $\frac{1}{i_1 \cdot f_n}$ And then multiply the result by 2^{-ep} To find the inverse of $1/b$. Extracting the exponent will leave us with a number in the range of $[1..2]$. Since we have the support of the hardware division in using the IEEE 754 standard (a 64-bit floating-point number), we can obtain our initial start guess with approximately 15-16 digits of accuracy and then begin to iterate towards a higher number of digits of precision. If you do not have access to IEEE 754, you can create a lookup table to find a suitable starting point.

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

$$x_{n+1} = x_n + x_n(1 - yx_n) + x_n(1 - yx_n)^2 \quad (5)$$

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

Alternatively, the iteration can be written as:

$$\begin{aligned} z_n &= 1 - yx_n \\ x_{n+1} &= x_n + x_n(z_n) + x_n(z_n)^2 \end{aligned} \quad (6)$$

Algorithm 2

It can be found using the Householder's 2nd order method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f(x_n)^2 f''(x_n)}{2f'(x_n)^3} \quad (7)$$

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

This yields:

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

The Math behind arbitrary precision

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

This method will require one subtraction, two additions, and four multiplications.

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

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

This will require three addition/subtraction operations and three multiplication operations. However, all the multiplication needs to be carried 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 the 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's (second-order) and the cubic (third-order) methods have advantages and disadvantages. If you begin to measure performance, you will notice that the Newton method is sometimes faster, while the cubic method is sometimes 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 must be an integral number. To achieve 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. 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 than or equal to (≤ 1.58), we choose the Newton method.

Goldschmidt Division method

The Goldschmidt method for fast division was invented in the early 1960s. It was implemented on various CPUs to compute division in a fast manner, similar to the Newton

The Math behind arbitrary precision

method introduced in the previous chapter. Like the Newton method, it has a quadratic convergence rate and only requires a few iterations to find the correct quotient in a division.

The basic idea is simple. You want to find $q=n/d$ and Goldschmidt notices that if you multiply the numerator and denominator by a series of F_n . You have:

$$q = \frac{n}{d} \frac{f_1}{f_1} \frac{f_2}{f_2} \dots \frac{f_n}{f_n} \tag{10}$$

He noticed that if you chose $f_i=2-D_i$ and the original d has been scaled so that $0<d<1$. And each iteration step is carried out as:

$$\frac{n_{i+1}}{d_{i+1}} = \frac{n_i f_{i+1}}{d_i f_{i+1}}$$

Then d_i will approach ~ 1 and the result of the division $q=n_i$

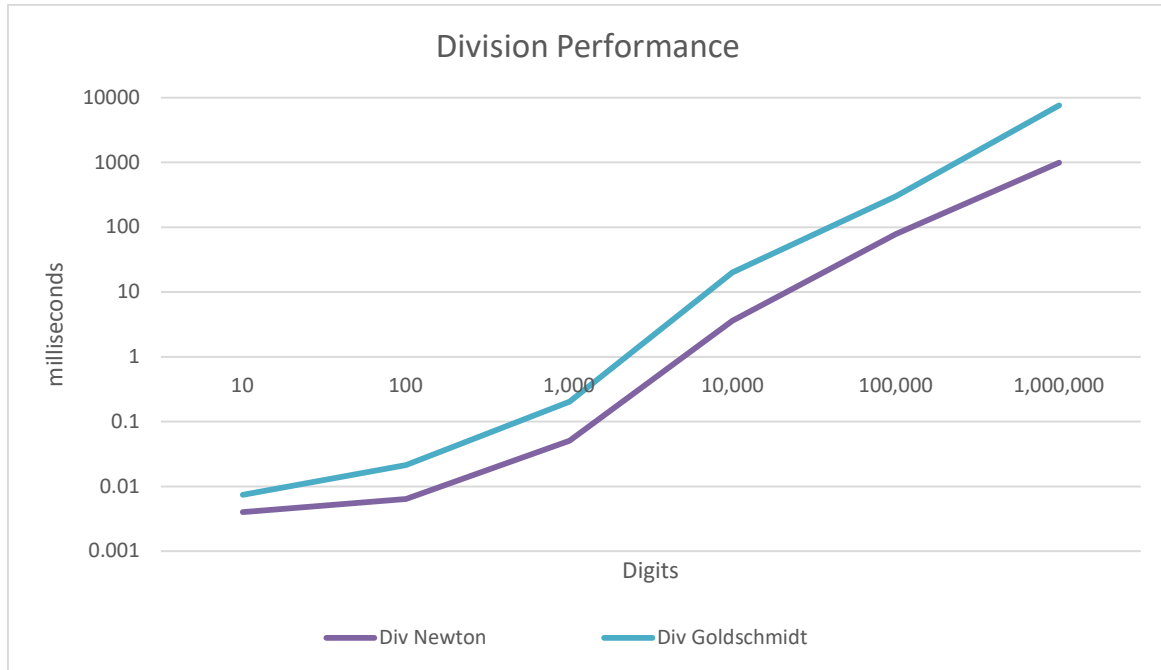
Example of the convergence. $q=1.1/0.25$ is listed for each iteration.

N=		1.1			
D=		0.25		"0<D<=1"	
n	F	Ni	Di	Error	
0		1.1	0.25	7.5E-01	
1	1.75	1.925	0.4375	5.6E-01	
2	1.5625	3.0078125	0.68359375	3.2E-01	
3	1.31640625	3.959503174	0.899887085	1.0E-01	
4	1.100112915	4.355900579	0.989977404	1.0E-02	
5	1.010022596	4.399558009	0.999899548	1.0E-04	
6	1.000100452	4.399999956	0.99999999	1.0E-08	
7	1.00000001	4.4	1	0.0E+00	

We see that after seven iterations, d_i is one and n_i is 4.4, which is the result of the division. We observe a quadratic convergence rate. Technically speaking, the d can be scaled to between 0 and 2. However, scaling can be achieved by just stripping the exponent from d and n . And then reapplying the exponent in the final result as $n_{\text{exponent}-d_{\text{exponent}}}$. When you strip an IEEE754 float from its exponent, you automatically have a magnitude between $0.5 \leq d < 1$, which gives us the starting value for d_0 .

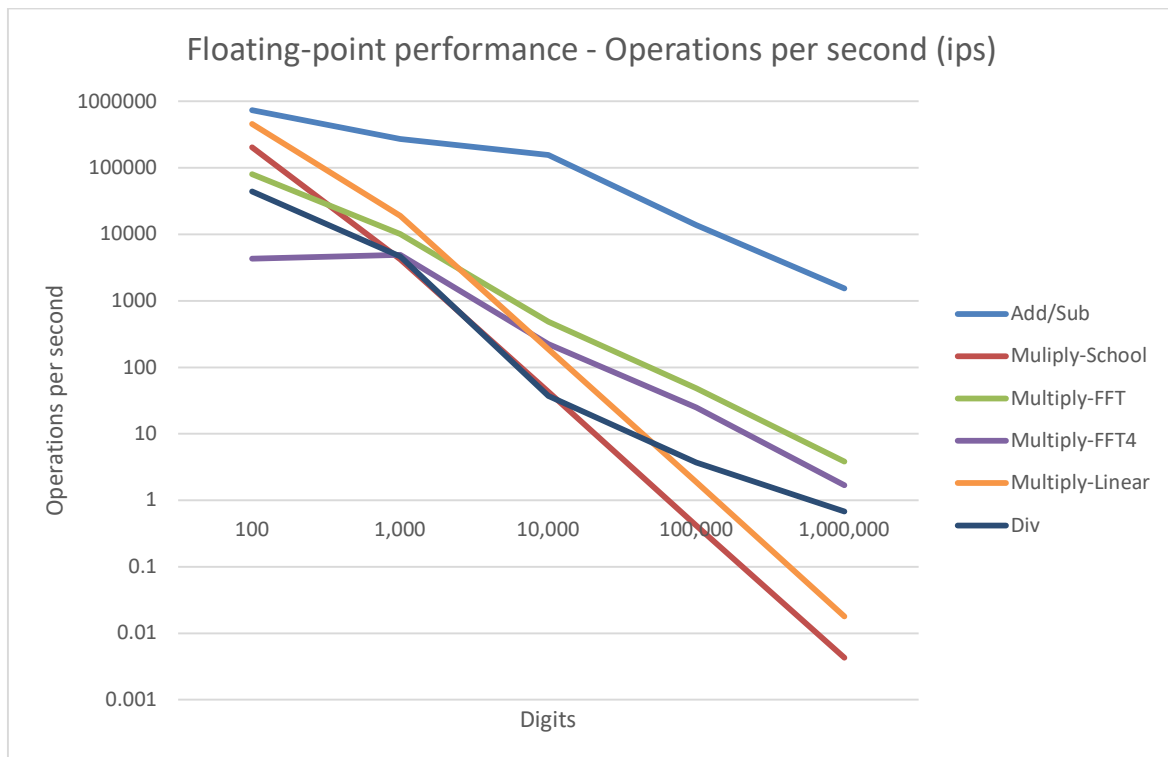
It is, however, slower than the Newton method, as shown in the performance chart below. This is mainly because each iteration of the Newton method requires only one multiplication, whereas the Goldschmidt method requires two. Furthermore, with the self-correcting Newton method, you can start the iteration at a lower precision and then gradually increase the precision to reach the desired target precision, resulting in a roughly fourfold improvement in performance. However, when the Goldschmidt method is implemented in the CPU hardware, the two multiplications can be carried out in parallel, thereby increasing the performance by a factor of two compared to the C++ implementation.

The Math behind arbitrary precision



Performance of Arbitrary Floating-point precision:

The following shows the performance of the floating-point precision. The y-axis is a logarithmic scale of the number of operations per second (Ops). The X-axis is the number of digits. For addition/subtraction/multiplication, both operands are of the same number of digits. For the division, the denominator has half as many digits as the dividend.



The Math behind arbitrary precision

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(8bit sample size)	9	27	324	289	404
Multiplication-FFT(4 bits sample size)	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 traditional schoolbook multiplication doesn't scale well with higher numbers of digits and is not helpful 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 bits) is a performance measure that utilizes a 4-bit sample size instead of the standard 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 half.

The big surprise is division. It is still much slower than multiplication, but faster than the division algorithm for integer arithmetic for very large operands. For low to medium operand size, the integer division using Knuth's D algorithm is faster. Therefore, we recommend that when performing integer division for very large operands, the integer is converted to a float_precision variable, then divided using the float_precision division, and finally converted back to an integer precision variable. (Conversion back and forth between an int_precision variable and a float_precision variable is swift.) In general, the performance graph suggests that it is advisable to avoid division whenever possible.

Needed extra functionality

In the previous section, we established the four basic arithmetic operations: Addition, Subtraction, Multiplication, and Division. These are the fundamental building 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. \sqrt{x}
- Elementary functions:
 - Exponential function.
 - Logarithm functions
 - Power functions x^y
 - Universal constants
 - $e, \pi, \ln 2, \ln 10$, etc.
- Trigonometric functions
 - Sine, Cosine, Tangent, Arcsine, Arccosine, Arctangent
- Hyperbolic functions

The Math behind arbitrary precision

- 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), The Lemniscate constant ϖ

The Math behind arbitrary precision

Square root:

Several methods exist 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], which includes the source code, performance charts, and other relevant information. Also [4] is a good reference for square root calculations.

Newton's Method for the square root

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

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (11)$$

This method can be found in the following way: by restating the problem of finding $\text{Sqrt}(y)$, 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 by 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 $f(x) = \frac{1}{x^2} - y$, $f'(x) = \frac{-2}{x^3}$

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

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

$$x_{n+1} = \frac{1}{2} x_n (3 - y x_n^2) \quad (12)$$

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

$$x_{n+1} = \frac{1}{2} x_n (3 - y x_n^2) \quad (13)$$

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 use the C library $\text{sqrt}(b)$ 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

The Math behind arbitrary precision

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 multiplications per iteration, and then a final multiplication for the calculation of the square root.

Additionally, regarding the Newton method, we will achieve quadratic convergence, meaning that with 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 by $2^{\frac{e_p}{2}}$. After the iteration (assuming e_p is an even integer), 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.

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

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

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

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 exponents) and [1, 4] for odd exponents. With the number well within the range of IEEE754, we can find a good initial guess of $\frac{1}{\text{Sqrt}(y)}$ Using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

Example of Newton's method for the square root

To see how this algorithm works, let us find the square root of 1.6 using an initial starting 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 five iterations, the difference between the iteration and the built-in Sqrt() operator is 0, and the result of Sqrt(1.6) is 1.264911064

The Math behind arbitrary precision

Brent's improvement

Brent [7] points 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 perspective. 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 the 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 compared to the Newton method, which only gives you 2 times as many correct digits. Higher order convergence results in fewer iterations at the expense of a more complex calculation per iteration. Usually, it tends to even out that the time saved in fewer iteration steps is lost due to a more complex iteration.

Halley's square root method uses the following iteration steps 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 Householder's 2nd-order method, also known as 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 yields:

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

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

The Math behind arbitrary precision

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

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

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

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

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

$$x_{n+1} = x_n \frac{1}{8} (15 - z_n(10 - 3z_n)) \tag{20}$$

Per iteration, we have five multiplications and two subtractions. Compared to Newton, we have additional subtraction and two extra multiplications, so that 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 the 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's (second-order) and Halley's (third-order method) have advantages and disadvantages. If you begin to measure performance, you will notice that the Newton method is sometimes faster, while the Halley method is sometimes 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 must 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. 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 than or equal to (≤ 1.58), we choose the Newton method.

The Math behind arbitrary precision

N'th root

Now that we have found a better way of doing the square root, we also need to consider whether we can use a similar technique 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 uses two costly 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 root. It is pretty similar to the way we got the algorithm for the `sqrt()` function. We are trying to find a function for the solution $x = \sqrt[n]{S} \Rightarrow x^n = S \Rightarrow \frac{1}{x^n} = \frac{1}{S}$

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:

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^{-n} - y}{-nx_i^{-n-1}} \Rightarrow \\ x_{i+1} &= x_i + \frac{1}{n}(x_i - x_i^{n+1}y) \Rightarrow \\ x_{i+1} &= x_i + \frac{1}{n}x_i(1 - x_i^n y) \Rightarrow \\ x_{i+1} &= x_i \frac{1}{n}(n + 1 - x_i^n y) \end{aligned} \quad (22)$$

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:

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

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

The Math behind arbitrary precision

Using the Newton method, you get:

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

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

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

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, such as the Halley method, but they will be slower than the Newton version. In the Booth Arbitrary Precision library, they conducted some testing, and the Newton method emerged as the most effective among all other methods. See [20]

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

Elementary functions:

For all elementary functions, regardless of whether they are logarithmic, exponential, trigonometric, or hyperbolic, we use either a Taylor series or an iteration method, such as Newton's, to find our results.

For all these methods, we follow the same approach. Before applying the Taylor series or Newton iterations, we first reduce the argument to improve the efficiency of our methods or simplify the problem to a domain where it is faster to evaluate the function. We reduced the argument x to x_1 and then evaluated the function $f(x_1)$ using x_1 . Finally, we restored 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, which reduces 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 functions with a periodic nature, such as sine and Cosine functions, which have a period of 2π .

Another approach 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 must square the result after our calculation to restore the original value. Needless to say, 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. Traditionally, a Taylor series expansion has been used; however, some have suggested using the $\sinh()$ function to calculate $\exp()$. 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:

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

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 rapidly and will require many Taylor terms to be completed.

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

The Math behind arbitrary precision

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 terms to get the result. If we use $x = 0.1$, then we only need 10 Taylor terms. This led to the observation that the number of Taylor's terms needed depends heavily on the magnitude of 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 this using a technique called *argument reduction*, which allows us to work with a smaller number to achieve faster convergence 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 by 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 by 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 we use the previous example 1 and reduce the argument from one to 0.25, we only need 12 Taylor terms to obtain the same result as before, saving five Taylor terms but gaining two squaring operations at the end. However, overall, huge savings have been achieved since we have avoided five time-consuming divisions in Taylor's terms.

Exp(x) x=			Original 1	X Reduced 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 Taylor's terms.

Exp(x)	Original	X Reduced
--------	----------	-----------

The Math behind arbitrary precision

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 illustrate the effectiveness of argument reduction.

The issue with arbitrary precision for e^x

17 Taylor's terms to reach a result do not seem so bad at 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 $|x| < 1$:

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

For $P = 1,000$ digits, 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 many Taylor terms, it will take a long time to evaluate $\exp(x)$ for high numbers of digits; see the 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 number 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 and 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 and 64 reductions.

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	5	12	75	516	4,393
1 Red.	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 Math behind arbitrary precision

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

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

It is two multiplication and one division. The $n+1$ can be handled using native C++ types and does not contribute to the workload for arbitrary precision.

Now, performing k reduction will require k multiplications before the Taylor iterations start, and k multiplications at the back-end, for a total of $2k$ multiplications. The front operation multiplication for a normalized arbitrary precision number is not performed as a real multiplication (of 0.5) but handled by just subtracting one from the exponent (which is the same as dividing by two or multiplying by 0.5). This does not contribute to the workload and can be disregarded. On the back-end, it will still require k multiplication. As an example, we can calculate the total workload for a 10,000-digit number using one reduction versus two reductions.

1-reduction workload = $3,861 \cdot (2 \cdot \text{multiplication} + 1 \text{ division}) + 1 \cdot \text{multiplication} = 7,723 \cdot \text{multiplication}$ and $3,861 \cdot 1 \text{ division}$.

16-reduction workload: $1,497 \cdot (2 \cdot \text{multiplication} + 1 \cdot \text{division}) + 2 \cdot \text{multiplication} = 2,996 \cdot \text{multiplication}$ and $1,497 \text{ division}$

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

Finding a reasonable reduction factor for e^x

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

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

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 by 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 magnitudes $|x|$ and for high magnitudes $|x|$, by simply 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.

The Math behind arbitrary precision

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	0.11	0.53	17	5,596	291,871
1 Red.	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 repeated squaring at the back end; instead, we just square for every number of reductions performed.

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

We use the identity as suggested by Brent [7]:

$$\begin{aligned}
 e^x - 1 &= (e^{\frac{x}{2}} - 1)(e^{\frac{x}{2}} + 1) \Rightarrow \\
 e^x - 1 &= 2(e^{\frac{x}{2}} - 1) + (e^{\frac{x}{2}} - 1)^2
 \end{aligned}
 \tag{32}$$

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 approximately Per term involving one multiplication, one division, and one addition to be:

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

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 the actual Taylor's terms versus the 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 Red.	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

The Math behind arbitrary precision

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 difference only accounts 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 digit calculation that we add.

$$2 + \text{ceil}(\log_{10}(\text{digits})) \text{ as extra guard digits.}$$

Further Improvement of the Taylor series for e^x ?

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

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

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:

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

Moreover, group them:

$$\begin{aligned} \dots \frac{(n+1)x^n}{(n+1)n!} + \frac{x^{n+1}}{(n+1)!} \dots &=> \\ \dots \frac{(n+1)x^n + x^{n+1}}{(n+1)!} \dots & \end{aligned}$$

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; you can do that for three terms:

$$\begin{aligned} \dots \frac{(n+1)(n+2)x^n + (n+2)x^{n+1} + x^{n+2}}{(n+2)!} \dots &=> \\ \dots \frac{x^n(x^2 + (n+2)x + n^2 + 3n + 2)}{(n+2)!} \dots & \end{aligned}$$

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

In general, you can add g group together:

The Math behind arbitrary precision

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

Because arbitrary precision division is significantly more time-consuming to calculate, implementing this grouping of Taylor terms will be highly advantageous. With four to five terms grouped, you get a speed up of 2-3 times compared to not grouping terms.

Full coefficient scaling

Now, why stop with the grouping of only a few Taylor terms? In [34], they devised a new computation of the Taylor series, postponing the division until all Taylor terms had been calculated.

He noticed that by evaluating the Taylor series backward, you can set this recursion:

$$n!e^x = n! + \dots + ((n(n-1)(n-2) + (n(n-1) + (n+x)x)x)x) \dots \quad (36)$$

Here, you summed up the series and postponed the division to a final division to calculate the e^x . This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Fortunately, determining the required number of Taylor terms is not a difficult task. We are using the Sterling approximation for the factorial. We can write the error terms needed for a given decimal precision P .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (37)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (38)$$

Where $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$

And $f'(y)$:

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (39)$$

Applying Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (40)$$

The Math behind arbitrary precision

$$y_{i+1} = y_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (41)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (42)$$

Since we need an integral number of Taylor terms, we don't need to carry that much precision. As a starting point, [34] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (43)$$

The starting formula works well for the Newton iteration since $0 < x < 1$ due to argument reduction. Since we are only interested in the ceiling n_{i+1} , we will only need a few iterations to find the number of Taylor terms required.

The full coefficient scaling experience approximately matches the same performance as the previous group, scaling by 5 times at a time. However, for precision over 1,000 digits, the group scaling by five shows better performance.

e^x using Sine Hyperbolic function

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

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (44)$$

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

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (45)$$

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

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots \quad (46)$$

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 compared 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.

The Math behind arbitrary precision

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 $|x| < 1$ to ensure the Taylor series converges more quickly. We again use *argument reduction* to work with a smaller number, achieving faster convergence to e^x using fewer *terms* of the Taylor series.

We can use the trisection identity: $\sinh(3x) = \sinh(x)(3 + 4\sinh(x)^2)$ To reduce the argument by 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 compared to 12

Exp(x)		Original	X Reduced	
x=		1	0.1111111111	
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 an 8-fold reduction, you obtain the result after two Taylor terms, compared to 6 using the 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

The Math behind arbitrary precision

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 compared 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 also be applied here. Consider the Taylor series for the sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (47)$$

The issue, again, is clearly 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:

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

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

Then you have replaced one division with two extra multiplications. 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; you can do that for three terms or more terms:

For grouping three Taylor terms, you get:

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

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

e^x using the binary splitting method

This method builds upon the same method for calculating e ; see the section on constants.

It used the Taylor series for $\exp(x)$:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots \quad (48)$$

The Math behind arbitrary precision

However, instead of calculating the series as above, we implement it using the binary splitting method.

The binary splitting methods [12] equate the Taylor series terms with two variables, p and q, and then it is just a matter of dividing p by q to obtain the approximation for e.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots = \frac{P(0,k)}{Q(0,k)} \quad (49)$$

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

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

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

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

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. This can be done easily by simply using argument reduction, as previously described 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 the 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:

$$\frac{x^k}{k!} < 10^{-P} \quad (50)$$

Where we use the Stirling approximation for k!:

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation} \quad (51)$$

This yields:

$$\frac{x^k}{\left(\frac{k}{e}\right)^k \sqrt{2\pi k}} < 10^{-P} \quad (52)$$

Taking log() on both sides, you get:

$$-k \cdot \log(x) + k \cdot (\log(k) - 1) + \frac{1}{2} \log(2\pi k) > P \cdot \log(10) \quad (53)$$

The Math behind arbitrary precision

To solve this for k, we can use Newton's method, which finds 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 table above clearly illustrates the effect of using the argument reduction technique 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 reduction factor for e^x

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

$$|x| < 10^{-M} \text{ and } M > \frac{\ln(P)}{\ln(10)} \tag{54}$$

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 by 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 magnitudes |x| and for high magnitudes |x|, by simply adding the exponent (positive or negative) to the reduction factor.

What precision is needed to avoid loss of accuracy?

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

$$P(a,b) = P(a,m)Q(m,b) + P(m,b) \tag{55}$$

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

The Math behind arbitrary precision

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation} \quad (56)$$

Using $\log_{10}(k!)$ We find the number of decimal digits as the size of $k!$

$$\log_{10}(k!) \approx \log_{10}\left(\left(\frac{k}{e}\right)^k \sqrt{2\pi k}\right) \Rightarrow$$

$$k \cdot \log_{10}(k) - k + \frac{1}{2} \log_{10}(2\pi k) \approx k \cdot \log_{10}(k) - k, \text{ for large } k \quad (57)$$

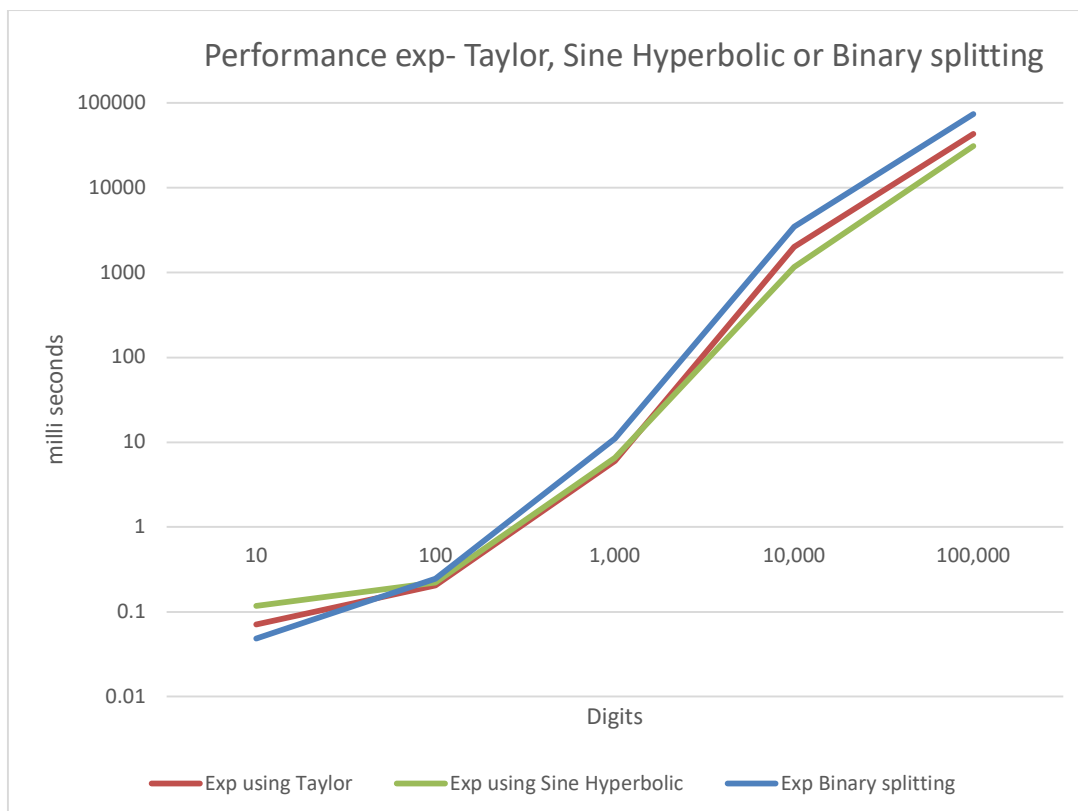
Digits	10	100	1,000	10,000	100,000	1,000,000
Size of $k!$ in decimal digits	9	100	2,000	30,000	400,000	5,000,000

Table of the decimal size of various values for $!$.

As expected, $!$. It is a powerful factor that requires us to adjust the required accuracy or precision when calculating e^x at a certain 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 calculations 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 it lacks the performance of the two other methods.



The Math behind arbitrary precision

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

The Math behind arbitrary precision

Logarithmic functions:

There are quite a few ways you can calculate $\log(x)$ in arbitrary precision. A 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's second-order method to calculate $\log(x)$
3. Using Halley's 3rd order method to calculate $\log(x)$
4. Using AGM algorithm to calculate $\log(x)$

The most common approach for arbitrary-precision libraries is the standard Taylor series expansion method; however, as will be shown, this is not the preferred choice if performance is desired. When we say $\log(x)$, we mean the natural logarithm, which 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:

$$\ln(x) = (x - 1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \dots \quad (58)$$

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, converges slowly to $\ln(x)$ and is therefore not suitable for arbitrary precision calculations. Instead, most implementations use the inverse hyperbolic tangent function:

$$\ln(x) = 2 \cdot \operatorname{artanh}\left(\frac{x-1}{x+1}\right) = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \dots\right) \quad (59)$$

The above series 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

The Math behind arbitrary precision

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 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 led 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 this using a technique called argument reduction, which allows us to work with a smaller number and achieve faster convergence to $\ln(x)$ using fewer *terms* of the Taylor series.

We can use the identity:

$$\ln(x) = \ln\left(\left(\sqrt{x}\right)^2\right) = 2 \cdot \ln\left(\sqrt{x}\right) \quad (60)$$

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 $x \Rightarrow x^{\frac{1}{2^k}}$. As we approach one, we can then, after the Taylor iterations, multiply the result by 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

Suppose we use the previous example 1 and reduce the argument from 2 to 1.1892... In that case, we only need 7 Taylor terms to obtain the same result as before, saving 8 Taylor terms but gaining two squaring operations and multiplication of $2^2 = 4$ at the end. However, overall, huge savings have been achieved 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

The Math behind arbitrary precision

If we use an eight-times reduction we get the same results after just four Taylor's 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)

A reasonable number of Taylor's terms to reach a result do not seem so bad at 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 be convenient if we could estimate the required number of Taylor terms for a given argument so that 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:

$$2 \cdot \frac{z^{2n-1}}{2n-1}, \text{ where } z = \frac{x-1}{x+1} \quad (61)$$

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:

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

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

$$n = \frac{1}{2} \left(\frac{-P \cdot \ln(10)}{\ln(z)} + 1 \right) \quad (62)$$

If we use the example of $x = 2$, we obtain the following estimated Taylor 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 number 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 by more than 8-10 times between a reduction factor of 0 and a reduction factor of 8.

The Math behind arbitrary precision

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 and 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 shown in the table above, a higher reduction factor significantly improves performance. However, how many times of reduction is adequate? 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 $\geq 1 < x < 2$.

$$\begin{aligned} & \text{Solve } x^{\frac{1}{2^k}} < \text{limit} \Rightarrow \\ \ln\left(x^{\frac{1}{2^k}}\right) < \ln(\text{limit}) & \Rightarrow \frac{1}{2^k} \cdot \ln(x) < \ln(\text{limit}) \Rightarrow \\ \frac{\ln(x)}{\ln(\text{limit})} < 2^k & \Rightarrow \ln\left(\frac{\ln(x)}{\ln(\text{limit})}\right) / \ln(2) < k \end{aligned} \quad (63)$$

A reasonable 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

The Math behind arbitrary precision

As you can see, for higher precision, you will benefit even by 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 digit calculation that we add.

$2 + \text{ceil}(\log_{10}(\text{digits}))$ as extra guard digits.
--

Further Improvement of the methods for $\ln(x)$?

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

$$\ln(x) = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \dots\right) \quad (64)$$

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

$$\ln(x) = 2\left(z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \dots\right) \quad (65)$$

As was the case when we discussed this in the exponential function paper [9], 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:

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

Moreover, group them:

$$\begin{aligned} \dots \frac{(n+2)x^n}{(n+2)n} + \frac{n \cdot x^{n+2}}{n(n+2)} \dots &=> \\ \dots \frac{(n+2)x^n + n \cdot x^{n+2}}{n(n+2)} \dots & \end{aligned}$$

Then you have replaced one division with three extra multiplications. The $(n+2)$ can be done using a 32-bit or 64-bit integer, since you rarely need to calculate many Taylor terms in real life. There is no need to stop at just grouping two terms; you can do that for three terms:

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

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

The Math behind arbitrary precision

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

Log(x) using the Newton method.

This method is only relevant if you have a very fast way to compute e^x . This is usually 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:

$$x_{n+1} = x_n - 1 + \frac{y}{e^{x_n}} \quad (66)$$

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, the Taylor series from the previous chapter is faster for fewer than 10,000 digits. However, above 10,000 digits, the Newton method outperforms 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 quicker. The Halley method with cubic convergence is:

$$x_{n+1} = x_n + 2 \frac{y - e^{x_n}}{y + e^{x_n}} \quad (67)$$

Algorithm 8

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

Log(x) using the AGM method.

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

$$\ln(x) = \pi \cdot x \frac{1 + \frac{4}{x^2} \left(1 - \frac{1}{\ln(x)}\right)}{2 \cdot \text{AGM}(x, 4)} \quad (68)$$

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” means 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 by 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 $s \cdot \ln(2)$:

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) \quad (69)$$

The Math behind arbitrary precision

For a given precision, P, s is found:

$$s = P \frac{\ln(10)}{2 \cdot \ln(2)} + 1 - \frac{\ln(x)}{\ln(2)} \quad (70)$$

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

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) = \frac{\pi \cdot x^{s-2}}{2 \cdot \text{AGM}(x^{s-2}, 1)} - s \cdot \ln(2), \text{ for } x > 1 \quad (71)$$

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

AGM Algorithm

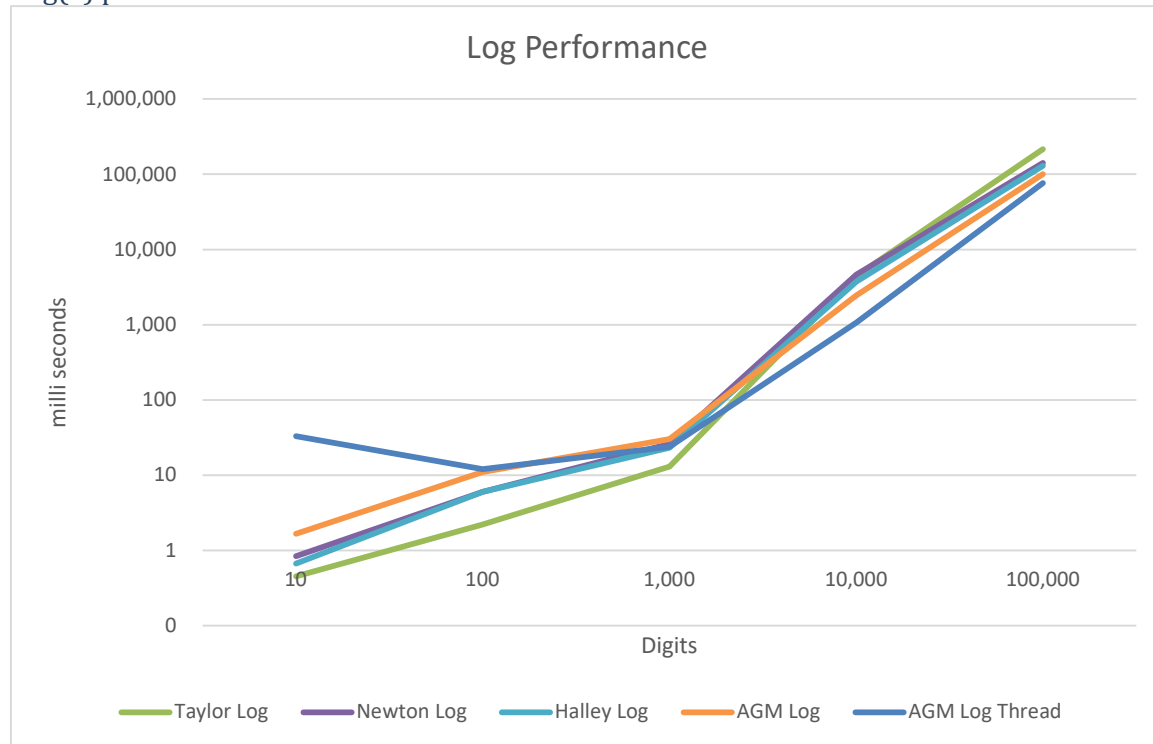
The arithmetic-geometric mean algorithm is defined for two positive numbers x & y by the following algorithm $\text{AGM}(x,y) = \lim_{n \rightarrow \infty} x_n = \lim_{n \rightarrow \infty} y_n$.

```
AGM(x,y)
  a0=x
  g0=y
  iterate:
    an+1 =  $\frac{1}{2}(a_n + g_n)$ 
    gn+1 =  $\sqrt{a_n g_n}$ 
  until an+1=gn+1
  return an+1
```

Algorithm 9

The Math behind arbitrary precision

Log(x) performance



Log Performance.

Based on the performance chart, the Taylor series log is the fastest up to approximately 4,000 digits; however, after this point, the AGM becomes the fastest in both the threaded and non-threaded versions. Above 10,000 digits, the Newton and Halley method also outperforms 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 be run in parallel in separate threads with a few simple changes to the source code, utilizing C++ lambda functions [10].

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 for 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.

The Math behind arbitrary precision

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

$\text{Log}_{10}(x)$:

To calculate $\text{Log}_{10}(x)$, we use the equation: $\text{Log}_{10}(x) = \text{Log}_e(x) / \text{Log}_e(10)$, and $\text{log}_e(x)$ has been handled previously. $\text{Log}_e(10)$ 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 functions using the equation:

$$x^y = e^{y \cdot \ln(x)} \quad (72)$$

$\text{Exp}()$ and $\text{Log}()$ are 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 :

$$x^y = (x)^{\frac{y}{2} + \frac{y}{2}} = (x)^{\frac{y}{2}} (x)^{\frac{y}{2}} = (x \cdot x)^{\frac{y}{2}} \quad (73)$$

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 $y < 0$, 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:

$$x = 2^n \text{ and } n \text{ and } y \text{ is an integer} \Rightarrow (2^n)^y = 2^{n \cdot y} \quad (74)$$

Constants: e , $\text{Log}_e(2)$, $\text{Log}_e(10)$ & π

Constants are not constants since they depend on the actual precision we need. We need the following constants: e , $\text{Log}_e(2)$, $\text{Log}_e(10)$, 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 of precision. We then calculate this constant only once. The next time we need the constant with equal or less precision ($\leq 100,000$ digits), we then use the stored constant and round it to the precision needed $\leq 100,000$ digits). If, on the other hand, we need the constant with higher precision, we then discard the stored constant and recalculate it with higher precision, using that as the new stored constant.

The constant e

The transcendental constant e (same as the function call $\text{exp}(1)$) 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 significantly faster than using the Taylor series for calculating $\text{exp}(1)$, even with the enhancements presented in this paper. Please refer 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. It is not the recommended choice.

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 by q to get the approximation for e .

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \dots = \frac{P(0,k)}{Q(0,k)} \quad (75)$$

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

$$\begin{aligned} m &= \frac{a+b}{2} \text{ integer division} \\ P(a,b) &= P(a,m)Q(m,b) + P(m,b) \\ Q(a,b) &= Q(a,m)Q(m,b) \\ \text{And } P(b-1,b) &= 1; \quad Q(b-1,b) = b; \end{aligned}$$

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 recursion reverse bottom up.

The Math behind arbitrary precision

Notice that 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 a 64-bit environment) from the author's arbitrary-precision packages.

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

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation}$$
$$\left(\frac{k}{e}\right)^k \sqrt{2\pi k} > 10^P$$

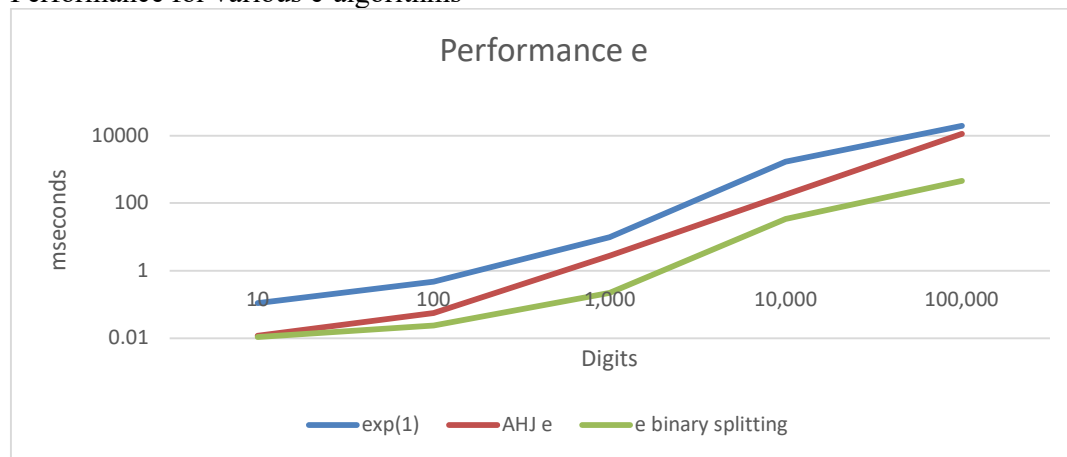
Taking $\log()$ on both sides, you get:

$$k \cdot (\log(k) - 1) + \frac{1}{2} \log(2\pi k) > P \cdot \log(10) \quad (76)$$

To solve this for k , we can use Newton's method, which finds 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 and 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 -algorithms



Recommendation for calculating e

Use the binary splitting method, which is approximately 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 $\exp(1)$ described in the previous chapter.

The Math behind arbitrary precision

The Constant Log(2)

Naturally, you can compute the commonly used constant $\log(2)$ using the general algorithm in the previous section. However, it is, by a long shot, not as fast as the current specialized function for computing $\log(2)$. These specialized methods exploit particular expansions for $\log(2)$ that converge rapidly by design, whereas the general approach must work for all values of x . As a result, the specialized series avoids many of the overheads inherent in the universal algorithm. The paper [13] presents a method utilizing a spigot-like algorithm, which is considerably faster for precision up to 10,000 digits. However, [36] and [37] provide excellent formulas for newer and quicker methods published in 2023-2024.

In this paper, we will outline a method for the rapid computation of the constant $\log(2)$. Because $\log(2)$ is central to everything from information theory to floating-point scaling, it's a common target for high-precision or high-speed computation. Below, we focus on specialized series that exploit exact forms of $\log(2)$.

Log(2) using:

- The general Log(x) algorithm was previously described.
- The spigot-like method `lnxy_64`.
- Xiao's binary splitting method.
- Zuniga-II binary splitting method.

The spigot-like method using built-in integer arithmetic (64-bit)

We now turn to a spigot-like method that relies heavily on integer arithmetic to handle moderate precisions (up to tens of thousands of digits) incredibly efficiently. This approach contrasts with binary splitting by focusing on incremental 'digits at a time' generation, simplifying some arbitrary precision integer overhead.

This method is detailed in [13], with the main benefit being that it utilizes only 64-bit integer arithmetic and is significantly faster than the general algorithm for $\log(x)$ for precision up to 50,000 digits. It is listed below and works for all integers or fractions. E.g. $\log(2)$ with 100 digits precision you call `spigot_lnxy_64(2,1,100,4)`; for $\log(10)$ you call `spigot_lnxy_64(10,1,100,4)`; and finally if you want a fraction e.g. $\log(1.25)$ you called it as `spigot_lnxy_64(10,8,100,4)`;

Xiao binary splitting for log(2)

The Xiao series in the form proper for the Binary splitting method is:

$$\log(2) = \frac{1}{4} \sum_{k=1}^n \frac{3927264b^3 - 4300512b^2 + 1209726b - 8189}{k(2k-1)(4k-1)(4k-3)} \prod_{i=1}^k \frac{i(2i-1)(4i-1)(4i-3)}{1440(10i-)(10i-3)(10i-7)(10i-9)} \quad (77)$$

Algorithm: Xiao's Binary splitting method for $\log(2)$

```
set m = (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:

$$P(b-1,b)=3927264b^3-4300512b^2+1209726b-81891$$

The Math behind arbitrary precision

$$\begin{aligned} Q(b-1,b) &= 1440(10b-1)(10b-3)(10b-7)(10b-9) \\ R(b-1,b) &= b(2b-1)(4b-1)(4b-3) \end{aligned}$$

Algorithm 12. Xiao Log(2) Algorithm.

And then

$$\log(2) = \frac{1}{4} \frac{P(0,n)}{Q(0,n)} + O(450000^{-n}) \quad (78)$$

These methods have a linearly convergent cost of ~ 1.23 .

For n terms, the error is $O(450000^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(450000)} \right\rceil \quad (79)$$

Zuniga's binary splitting method for log(2)

Zuniga series in the form proper for the Binary splitting method is:

$$\log(2) = \frac{1}{2} \sum_{k=1}^n \frac{1794k-297}{k(2k-1)} \prod_{i=1}^k \frac{i(2i-1)}{216(6i-1)(6i-5)} \quad (80)$$

It is less complex than the Xiao method.

Algorithm: Zuniga Binary splitting method for Log(2)

$$\begin{aligned} \text{set } m &= \frac{a+b}{2} \text{ integer division} \\ P(a,b) &= P(a,m)Q(m,b) + P(m,b)R(a,m) \\ Q(a,b) &= Q(a,m)Q(m,b) \\ R(a,b) &= R(a,m)R(m,b) \end{aligned}$$

And:

$$\begin{aligned} P(b-1,b) &= 1794b-297 \\ Q(b-1,b) &= 216(6b-1)(6b-5) \\ R(b-1,b) &= b(2b-1) \end{aligned}$$

Algorithm 13. Zuniga Log(2) method.

And then

$$\log(2) = \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O(3888^{-n}) \quad (81)$$

These methods have a linearly convergent cost of ~ 0.97 , which is lower than the previous Xiao method.

For n terms, the error is $O(3888^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(3888)} \right\rceil \quad (82)$$

The Math behind arbitrary precision

Comparison of the Log(2) methods

Having introduced each specialized approach, Spigot, Xiao, and Zuniga, as well as the more general $\log(x)$ routine, we can now compare them in terms of accuracy, convergence rates, and practical runtime. Each approach has its optimal sweet spot in terms of precision required and implementation cost.

The four Log(2) methods are listed below.

Method	Implementation	Error	N(P), P=precision
Log(x)	A combination of the Taylor series and the AGM		
Spigot Lnxy_64	Spigot		
Xiao	Binary-Splitting	$O(450000^{-n})$	0.18P
Zuniga II	Binary-Splitting	$O(3888^{-n})$	0.27P

Table 1. Comparison of key characteristics of the three methods.

The Xia methods require fewer splits to achieve a given level of precision in the result. However, each split is more time-consuming than the Zuniga version.

Performance for log(2)

The performance measurements below were obtained on a 3.0 GHz desktop with 32 GB of RAM, unless otherwise noted, using a single thread. Each algorithm used the same arbitrary precision integer library to ensure a fair comparison. The data points illustrate the raw speed and how each method scales with the number of digits.

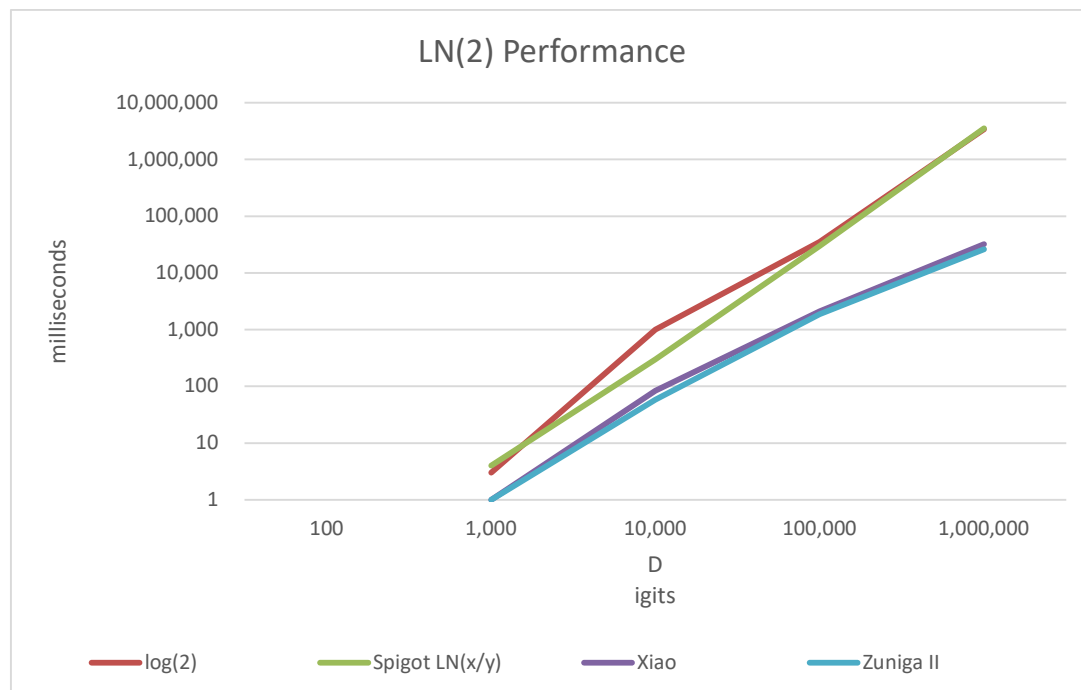


Figure 1. Log(2) performance chart.

The Math behind arbitrary precision

Time to compute the $\text{Log}(2)$ constant up to 1M digits. Notice that the general $\log(x)$ and the Spigot $\text{LN}(x/y)$ are way behind the Xiao and Zuniga binary splitting methods. A threaded version of the Zuniga method is advantageous for precision beyond 10,000 digits.

Log(2) constant. Time in milliseconds.						
Digits	10	100	1,000	10,000	100,000	1,000,000
log(2)	-	-	3	999	35,098	3,404,195
Spigot LN(x/y)	-	-	4	301	29,749	3,547,977
Xiao	-	-	1	83	2,101	32,029
Zuniga II	-	-	1	58	1,847	26,227

Table 2. Performance of the $\text{Log}(2)$ constant.

As you can see, the Xiao and Zuniga methods outperform the others by more than a factor of 100 with 1 million digits. Although the Spigot $\text{LN}(x/y)$ only uses native integer arithmetic, it quickly falls behind.

Recommendation for the $\log(2)$ constant

I recommend the following:

1. Implement one of the binary splitting methods.
2. The Zuniga method is faster than the Xiao; however, both are straightforward to implement compared to the $\log(x)$ and Spigot methods. Therefore, it is recommended.

The Constant $\text{Log}(10)$

There are not that many contenders to the $\log(10)$ constant. A binary splitting version is available, but it is very slow. For each splitting, you only gain around 0.1740 decimal digits. That's why a faster algorithm for $\log(10)$ can be computed using $\log(2)$ and $\log(5)$. Many high-precision applications also deal directly with $\log(2)$ or $\log(3)$. By using existing fast routines for $\log(2)$ and $\log(5)$, we get a de facto 'binary splitting' approach for $\log(10)$ without needing a dedicated series. We can use the identity $(\log(10)=\log(2\cdot 5)=\log(2)+\log(5))$. Which leads us to examine these three versions.

For $\text{Log}(10)$ we will examine:

- The general $\text{Log}(x)$ algorithm is described in the previous section.
- The spigot-like method `lnxy_64`.
- Combined the recommended $\log(2)+\log(5)$ from previous sections and [10].

Each of these three methods has distinct trade-offs. The general $\log(x)$ function is universal but tends to be slower at high precision. The spigot-like method can be quite elegant, but struggles with large arguments. Finally, the combined approach exploits our high efficiency binary splitting solutions for $\log(2)$ and $\log(5)$. Below, we examine how these methods stack up.

Although `lnxy_64` is known to slow down for larger inputs, there is a neat workaround. By decomposing 10 into smaller factors, we can keep arguments within a range where `lnxy64` does better. Specifically we can rewrite $\log(10)=\log(8\cdot 10/8)=\log(8)+\log(10/8)=3\cdot\log(2)+\log(10/8)$. This maps the argument into two ranges where the `lnxy_64` function performs faster. See the graph mark table_LN10 compared to the spigot $\text{LN}(x/y)$ graph.

The Math behind arbitrary precision

Performance of $\log(10)$

To compare these approaches, we measured the time required for each method to achieve various digit precisions. The tests were performed on the same hardware and arbitrary-precision integer library used in the previous sections, ensuring fair comparisons. The tables and figures below reveal how the combined $\log(2)+\log(5)$ approach stands out.

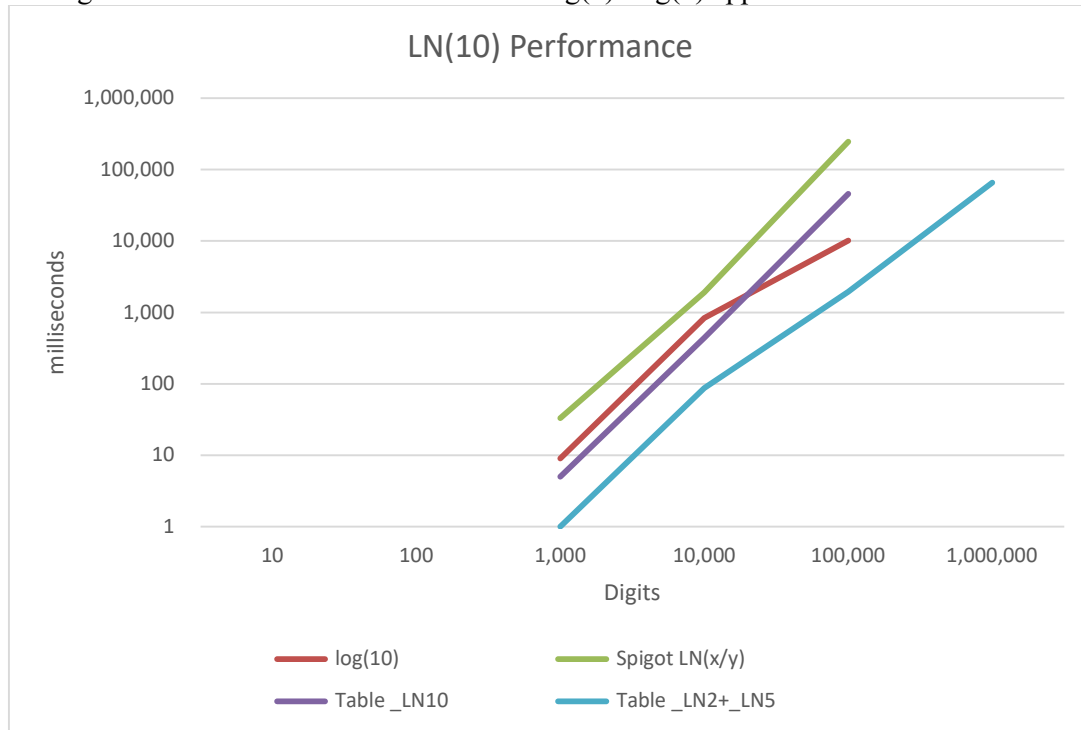


Figure 2. Log(10) performance chart. _Table indicate the call to the build in table for $\log(2)$, $\log(5)$ and $\log(10)$ in the arbitrary precision library.

The faster binary splitting of $\log(2)+\log(5)$ outperforms any other method.

LN(10) Performance Result. all times are in msec						
Digits	10	100	1,000	10,000	100,000	1,000,000
log(10)	-	-	9	836	10,155	
Spigot LN(x/y)	-	-	33	1,914	245,431	
Table_LN10	-	-	5	443	45,770	
Table_LN2+_LN5	-	-	1	87	1,945	65,323
Thread LN2+LN5	7	2	3	92	2,109	49,515

Table 3. Log(10) performance for different methods.

Recommendation for $\log(10)$

I recommend using the faster $\log(2)+\log(5)$ binary splitting method from the previous sections and [10].

If higher performance is required, consider a threaded version where $\log(2)$ and $\log(5)$ are computed in separate threads. In practice, parallelizing the computation is straightforward since $\log(2)$ and $\log(5)$ are independent sums. Each can be spun off into its thread, and once both partial results are ready, you add them together. This parallelism can offer a sizable speed boost on modern multicore processors, especially at very high precision.

The Math behind arbitrary precision

The constant π

This is another interesting constant that has been the subject of considerable attention for the last thousand years. For a thorough walkthrough of various algorithms to calculate π , we recommend reading [5] Borwein, "PI and the AGM". However, [12] is also a good source. For our implementation, we can use one of the many algorithms for calculating π that can be found in [2][5][14][15][16]. In [17] we walk through many of these algorithms for π with practical examples of implementations. Below are a few selected and worth considering.

Borwein π

Algorithm for Borwein π

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

The repeat for $i=1,2,3,\dots$ 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 14

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 three 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, slightly better than the Borwein algorithm, is the Gauss-Legendre algorithm, and the deviation associated with it is also known as the Bent-Salamin deviation.

Algorithm for Brent-Salamin π

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

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

$$a_{n+1} = \frac{1}{2} (a_n + b_n)$$
$$b_{n+1} = \sqrt{a_n b_n}$$

The Math behind arbitrary precision

$$c_{n+1} = c_n - 2^{n+1}(a_{n+1} - b_n)^2$$

$$\pi_{n+1} = 2 \frac{a_{n+1}^2}{c_{n+1}}$$

Algorithm 15

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

Regarding the Borwein algorithm, we achieve quadratic convergence, doubling the number of correct digits with each iteration. After 10 iterations, we have more than 1,000 digits, and after 20 iterations, more than 1 million digits. Borwein also demonstrated several higher-order convergence rate algorithms for finding π , with convergence rates for each iteration that multiply the number of correct digits by factors of 3, 4, 5, and 9. However, these algorithm requires a lot more work to be done per iteration and are usually not worth implementing compared to the above with quadratic convergence.

Binary splitting of the Chudnovsky infinite series

There is one method that surpasses all the classic methods, as outlined in [17], and that is the Chudnovsky method, which utilizes binary splitting.

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

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

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

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

Given the first k terms of the series, you get (see [26]):

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

$$\pi = \frac{4270934400 \cdot Q(0, k)}{P(0, k) + 13591409Q(0, k)} \frac{1}{\sqrt{10005}} + O(151931373056000^{-k}) \quad (83)$$

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

The Math behind arbitrary precision

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

We take k as the ceiling of:

$$k = \left\lceil \frac{P \cdot \log(10)}{\log(151931373056000)} \right\rceil \quad (84)$$

Algorithm for the 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:

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

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

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

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

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 16

Recommendation for the Infinite series for π

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

The Math behind arbitrary precision

Trigonometric functions:

There are quite a few ways you can calculate trigonometric functions with arbitrary precision. The 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:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (85)$$

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

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (86)$$

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 $> 2\pi$ so it falls between zero and 2π by simply taking x modulo 2π .

Step 2: We can further reduce x so it is between $0.. \pi$ using the identity:

$$\sin(x) = -\sin(x-\pi) \text{ for } x \geq \pi.$$

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

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

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.

The Math behind arbitrary precision

This argument reduction is performed to reduce the number of Taylor iterations, minimize round-off errors, and decrease 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	0.70000000000	-5.58E-02	
2	5.72E-02	0.6428333333	0.64283333333	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. For example, with two reductions, you obtain 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	

The Math behind arbitrary precision

Again, we notice that using argument reduction can significantly reduce the number of Taylor terms needed and thereby improve 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 first glance. In the previous examples, we were only using approximately 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 [6], 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 $x=10^M$:

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

$$n \approx \frac{1}{2} \frac{(P-M) \cdot \ln(10)}{\ln(P-M) - 1 - M \cdot \ln(10)} - 1 \quad (87)$$

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 approximately three, and for 100 million digits, it is approximately 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 , rather than just its magnitude. It usually gives a slightly smaller number of needed Taylor terms. This formula can be pretty useful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (88)$$

The Math behind arbitrary precision

Finding a reasonable reduction factor for $\sin(x)$

As shown in the table above, a higher reduction factor significantly improves performance. However, how many times is a reduction 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

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (89)$$

Starting from $x=1$, you obtain, for $P=100$ digits, that the required Taylor term is 24. Performing three reductions, you obtain $x = 1/3^3 = 0.037$. Using the above formula, we expect to need only 14 Taylor terms. Each Taylor term requires one addition/subtraction, one division, and one multiplication, which yields a total saving of 10 additions, 10 subtractions, and 10 multiplications. Compared to three reductions on the front end, there are three divisions and nine multiplications on the back end, resulting in a total saving of seven subtractions/additions, one multiplication, and seven divisions. Since division is a magnitude slower than multiplication and addition/subtraction, we can give a rough saving equivalent of seven divisions. For higher precision, the savings become larger.

We automatically calculate the reduction factor as $k = 8 \left\lceil \frac{2}{3} \ln(2) * \ln(P) \right\rceil$ For higher precision, we adjusted the magnitude of x . After Step 2, we know that x is in the range of $[0.. \pi]$, which is equivalent to 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 becomes smaller as x approaches zero, preventing us from making unnecessary reductions. If x is very small, the reduction factor is negative, and we 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 introduced by combining the front and back end calculations, we increase the precision by a quarter of the k factor. The increased precision generates only a small performance penalty compared to the extra savings 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 digit calculation that we add.

$$\boxed{2 + \text{ceil}(\log_{10}(\text{precision})) \text{ as extra guard digits as the working precision.}}$$

Further Improvement of the Taylor series methods?

Coefficient scaling can be considered an improvement over the standard method.

Generally speaking, you can have three options here.

- No coefficient scaling
- Partial coefficient scaling
- Full coefficient scaling

The Math behind arbitrary precision

No coefficient scaling

No coefficient scaling is just the regular use of the Taylor series, as presented above. Each Taylor term is computed, including the division, which performs poorly compared to the other operators, particularly in arbitrary precision libraries.

Although the method is straightforward, it ensures that each term is as accurate as possible before contributing to the final sum. It can be computationally expensive due to repeated division operations, mainly when used in connection with arbitrary precision libraries.

Partial coefficient scaling

You can group more Taylor terms before performing the division. For example, you reduce the number of division operations by calculating five Taylor terms at a time and then dividing the sum of these terms by the appropriate factorial. Partial coefficient scaling will enhance performance since division is generally more computationally intensive than multiplication or addition. The trade-off here is a potential slight decrease in numerical precision, as errors could accumulate in the grouped terms before the division normalizes them.

Full coefficient scaling

When doing a full coefficient scaling, you postpone the division until all Taylor terms have been computed. This method involves summing all scaled Taylor series terms first and dividing the final sum by the factorial. This method may be the fastest in reducing the number of division operations, but it also carries the risk of the most significant errors. When terms with large magnitudes are summed without immediate normalization, floating-point errors can accumulate, particularly for higher-order terms or larger values of x .

The discussion of partial and full coefficient scaling is as follows.

Partial coefficient scaling

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

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (90)$$

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):

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

Moreover, group them:

$$\begin{aligned} \dots \frac{-(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \dots &=> \\ \dots \frac{-(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \dots & \end{aligned}$$

The Math behind arbitrary precision

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

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

Then you have replaced one division with two multiplications. 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; you can do that for three terms or more:

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

Saving two divisions, however, gaining a few more additions 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 (begin with a minus sign) or an even number starting with a plus sign, and then alternate the sign thereafter.

Full coefficient scaling

Now, why stop with the grouping of only a few Taylor terms? In [34], they devised a new computation of the Taylor series, postponing the division until all Taylor terms had been calculated.

He noticed that by evaluating the Taylor series backward, you can set this recursion:

$$n!e^x = n! - \dots + ((-n(n-1)(n-2) + (-n(n-1) + (n+x)x)x)x) \dots \quad (91)$$

Here, you summed up the series and postponed the division to a final division to calculate the example. This approach is worth considering since division is an expensive operator with arbitrary precision. The \pm sign indicates that you alternate between $+$ and $-$ depending on the power of the Taylor term.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Fortunately, determining the required number of Taylor terms is not a difficult task. We are using the Sterling approximation for the factorial. We can write the error terms needed for a given decimal precision P .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (92)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

The Math behind arbitrary precision

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (93)$$

Where $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$

And $f'(y)$:

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (94)$$

Applying Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (95)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + 1 - (n_i) - \ln(|x|) - 1} \quad (96)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + 1 - (n_i) - \ln(|x|) - 1} \quad (97)$$

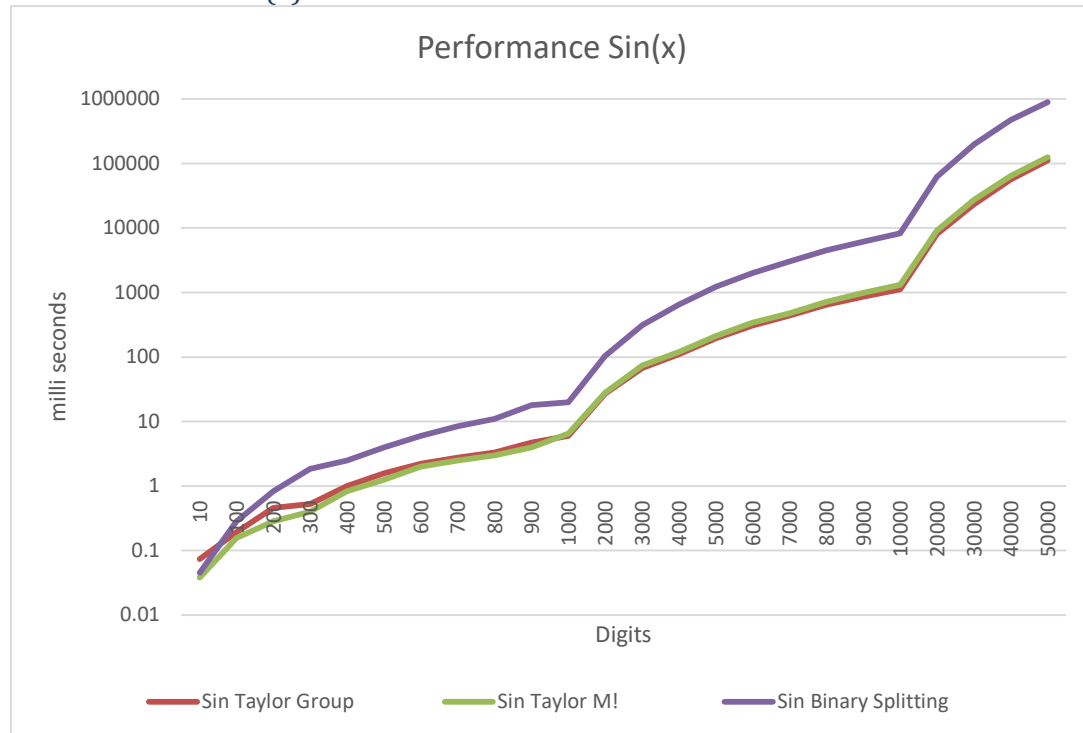
Since we need an integral number of Taylor terms, we don't need to carry that much precision. As a starting point, [34] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - 1) - (-\ln(|x|))} \quad (98)$$

The starting formula works well for the Newton iteration since $0 < x < 1$ due to argument reduction. Since we are only interested in the ceiling n_{i+1} , we will only need a few iterations to find the number of Taylor terms required.

The Math behind arbitrary precision

Performance for $\sin(x)$



Performance chart for $\sin(x)$ calculation.

The above table shows that using binary splitting techniques to calculate $\sin(x)$ is not faster. The two other methods using the Taylor series with partial coefficient scaling are neck and neck with those using the Taylor series with full coefficient scaling. We observe that full coefficient scaling is slightly faster, up to approximately 1,000 digits. In contrast, the partial coefficient scaling takes over and is approximately 10% faster than the full coefficient scaling method. This is a pattern we have seen before for other elementary functions.

Recommendation for calculating $\sin(x)$

Based on the performance measure of the various $\sin(x)$ methods, we recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.. \pi]$
- It is unnecessary to reduce it to the range $[0.. \frac{\pi}{2}]$ Using symmetry, we avoid 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.
 - a. Full coefficient scaling is faster, up to around 1,000 digits of precision.
 - b. Partial coefficient scaling is approx. 10% faster than full coefficient scaling above 1,000 digits of precision.

Cos(x) using Taylor Series.

For $\cos(x)$, we again use a Taylor series until any additional terms do not change the result to the given precision of the number.

The Math behind arbitrary precision

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{for any real value } x \quad (99)$$

We can map the equivalent four-step $\cos(x)$ procedure into the interval. $[0 \dots \frac{\pi}{2}]$.

Step 1: We notice that $\cos(x)$ is cyclic with a period of 2π , so we can quickly reduce any argument greater than 2π to a value between 0 and 2π by simply taking x modulo 2π .

Step 2: We can further reduce x so it is between $0 \dots \pi$ using the identity:

$$\cos(2\pi-x) = \cos(x) \text{ for } x \geq \pi.$$

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

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

Suppose π is ‘expensive’ to calculate (usually the case with arbitrary precision). In that case, we can omit step 3, as we have an alternative method to achieve the same result by 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 the x , the fewer terms we would need. We could also use the double-angle identity as an alternative:

$$\cos(2x) = 2\cos^2(x) - 1$$

Although the trisection identity serves us well for calculating $\sin(x)$, it turns out that there is a much higher loss of precision when using the trisection identity over the double-angle formula. See later.

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

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

Let us find the $\cos(0.7)$ to see how this algorithm works. The error is zero after the 8th Taylor term, 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

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

The Math behind arbitrary precision

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 significantly reduce the number of Taylor terms needed, thereby improving performance in calculating $\cos(x)$.

We have noticed that the error has increased, and we cannot find a more accurate answer than an absolute error of $\sim 1\text{E-}13$. The higher the reduction factor, the worse it gets. It is worth noting that this issue arises solely from the use of a reduction factor, rather than from the Taylor series.

Many of the same arguments used in calculating $\sin(x)$ also apply to $\cos(x)$, including aggressive argument reduction, coefficient scaling, etc. We must be cautious about how aggressively we reduce our argument.

Cos(x) using double-angle reduction

Argument reduction reduces x to a much smaller value that is 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:

$$\cos(2x) = 2\cos^2(x) - 1 \quad (100)$$

Alternatively, it is even better written as:

$$\cos(2x) = 2(1 - \cos(x))^2 - 4(1 - \cos(x)) + 1 \quad (101)$$

Although it does not prevent round-off errors, it is less sensitive than the trisection formula. We calculate the reduction factor for $\cos(x)$ as:

$$k = 2[\ln(2) * \ln(P)] \quad (102)$$

The Math behind arbitrary precision

For higher precision, we made adjustments for the magnitude of x . After Step 2, we know that x is in the range of $[0 \dots \pi]$. This is equivalent to the exponent of our number (in base 2) being in the range $[-\infty \dots 1]$. We add the exponent to the reduction factor. This means that our reduction factor becomes smaller as x approaches zero, preventing us from making unnecessary reductions. If x is very small, the reduction factor is negative, and we do not perform any argument reductions.

Cos(x) using full coefficient scaling.

Similar to $\sin(x)$, we can improve performance by using partial or full coefficient scaling. The version with full coefficient scaling follows the same layout as for $\sin(x)$, but uses the Taylor series for $\cos(x)$.

Cos(x) using sin(x)

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

$$\cos(x) = \sqrt{1 - \sin^2(x)} \quad (103)$$

Using the above formula increases performance by a factor of two compared to the traditional method of calculating $\cos(x)$ directly, and it is recommended.

There is another alternative to using the identity: $\cos(x) = \sin\left(\frac{\pi}{2} - x\right)$. If you have a fast generation of π , you will experience a similar performance to the $\cos(x) = \sqrt{1 - \sin^2(x)}$. But I think relying on the faster $\text{sqrt}(x)$ function will be safer.

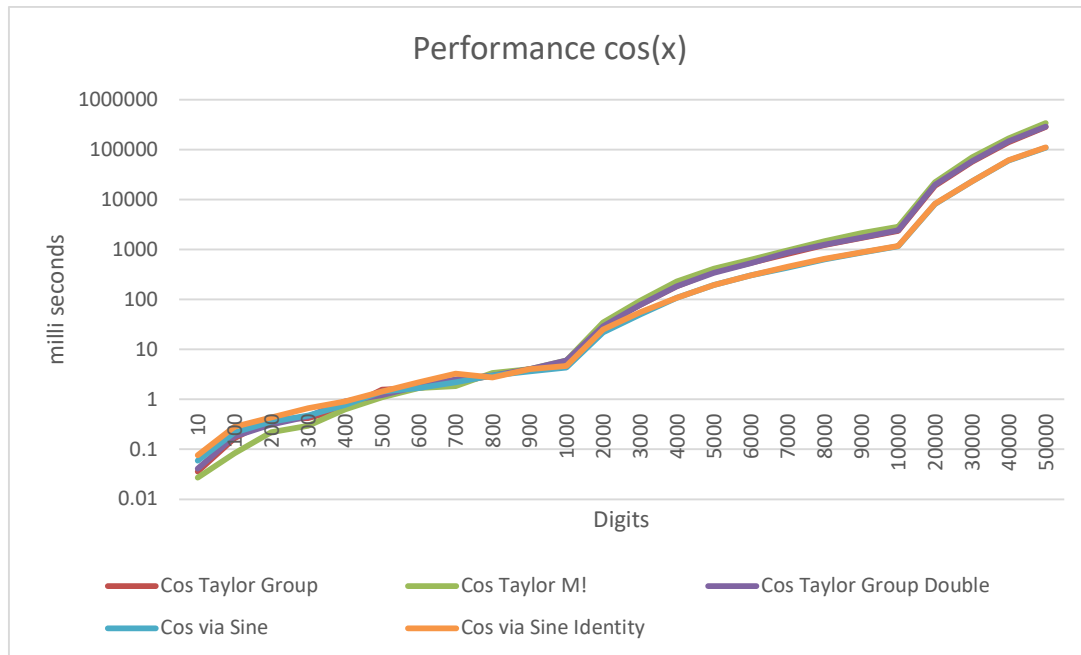
Another alternative to using $\sin(x)$ when calculating $\cos(x)$ is to use the identity.

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

If we have a very fast $\sin(x)$ computation, and we can take advantage of the constant $\frac{\pi}{2}$ It only needs to be calculated once, and then we can use the cache value for all subsequent $\cos(x)$ calculations.

Performance for Cos(x)

The Math behind arbitrary precision



Performance graph for $\cos(x)$ with various methods.

As the graph above shows, computing $\cos(x)$ via the sine identity is significantly faster than the other method, which exhibits similar performance. There is an exception: for a small precision of $\cos(x)$, using full coefficient scaling is slightly faster, below 800-1000 digits.

Recommendation for calculating $\cos(x)$

Based on the performance measure of the various $\cos(x)$ methods, we recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0, \pi]$
- It is unnecessary to reduce it to the range $[0, \dots, \frac{\pi}{2}]$ Using symmetry, we avoid another calculation of π .
- Use the double-angle formula instead of the trisection formula for argument reduction.
- Do not use the Taylor series for $\cos(x)$ with an aggressive reduction factor to speed up the Taylor term calculation. If you must use it anyway, consider using coefficient scaling to increase performance.
- However, use $\cos(x) = \sqrt{1 - \sin^2(x)}$ or $\cos(x) = \sin(\frac{\pi}{2} - x)$. This is the preferred method for calculating $\cos(x)$, which is two times faster than 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:

$$\tan(x) = \frac{\sin(x)}{\sqrt{1 - \sin^2(x)}} \tag{104}$$

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

The Math behind arbitrary precision

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

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^5}{315} + \frac{62x^9}{2835} + \dots + \frac{2^{2n}(2^{2n}-1)B_n x^{2n-1}}{(2n)!} + \dots \quad (105)$$

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, will be 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 use a Taylor series for $\arcsin(x)$.

Arcsin using Newton's method

To find the value of $\arcsin(x)$, it is trendy to resort to a Newton iteration when solving the equation $\arcsin(a)=x \Rightarrow a=\sin(x)$.

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

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\cos(x_n)} \quad (106)$$

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:

$$\cos(x) = \sqrt{1 - \sin^2(x)} \quad (107)$$

Yields:

$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\sqrt{1-\sin^2(x_n)}} \quad (108)$
--

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

$$\text{Arcsin}(x) = 2 \cdot \text{ArcSin}\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (109)$$

The x argument will always be between -1 and 1 per definition, so we only need a maximum of two argument reductions to get below 0.5.

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

$$x_k = \frac{x_{k-1}}{\sqrt{2}\sqrt{1+\sqrt{1-x_{k-1}^2}}} \quad (110)$$

Until x_m is sufficiently low, we can start with an initial guess of $\arcsin(x)$ using standard IEEE754. That provides us with a starting point for the Newton iteration, with at least 15 significant digits. The Newton iteration will converge quickly, with a convergence rate of 2,

The Math behind arbitrary precision

meaning the number of correct digits doubles with each iteration. After we find the new x_n , we will need to multiply the result by $x = x_n \cdot 2^k$, Reverse the argument reduction we performed before the Newton iteration.

Let us find the $\text{arcSin}(0.3)$ to see how this algorithm works. 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 we did not do any argument reduction for a moment, we will see a much slower convergence when x gets 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

The Math behind arbitrary precision

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

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 results in relatively few iterations; however, it is still not as fast as the direct approach using the Taylor series, as seen in the 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:

$$\text{Arcsin}(x) = x + \frac{x^3}{2 \cdot 3} + \frac{3x^5}{2 \cdot 4 \cdot 5} + \frac{3 \cdot 5x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{3 \cdot 5 \cdot 7x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \dots \quad (111)$$

This provides a more direct approach to arcsin(x), and when combined with the reductions, we observe a speed-up in the calculation of two. This method will become increasingly faster as precision rises compared to the Newton version.

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

$$r_1 = x$$

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

We calculate the reducing factor, k, as:

$$2 \cdot \lceil \ln(2) * \ln(\text{precision}) \rceil \quad (112)$$

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 formula:

$$\text{Arcsin}(x) = 2 \cdot \text{ArcSin}\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (113)$$

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

The Math behind arbitrary precision

Below is an example of using the Taylor Series for calculating $\text{arcSin}(x)$ 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	0.3000000000000000	0.3000000000000000	4.69E-03
2	4.50E-03	0.3045000000000000	0.3045000000000000	1.93E-04
3	1.82E-04	0.3046822500000000	0.3046822500000000	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 15-16 correct decimal digits. If we run it with a reduction factor of two, we get:

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. Again, this demonstrates that argument reduction can significantly reduce the workload.

Arcsin coefficient scaling

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

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

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

The Math behind arbitrary precision

$$\begin{aligned} \text{Two Taylor terms} &= r_{n-1} \frac{u_1 \cdot x^2}{l_1} + r_{n-1} \frac{u_1 \cdot x^2}{l_1} \cdot \frac{u_2 \cdot x^2}{l_2} \Rightarrow \\ &r_{n-1} x^2 \left(\frac{u_1}{l_1} + \frac{u_1}{l_1} \cdot \frac{u_2 \cdot x^2}{l_2} \right) \Rightarrow \\ &r_{n-1} x^2 \left(\frac{u_1 l_2}{l_1 l_2} + \frac{u_1 u_2 \cdot x^2}{l_1 l_2} \right) \Rightarrow \\ &r_{n-1} x^2 \left(\frac{u_1 l_2 + u_1 u_2 \cdot x^2}{l_1 l_2} \right) \end{aligned}$$

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

$$r_1 = x, \quad r_{n+1} = r_{n-1} x^4 \frac{u_1 u_2}{l_1 l_2}$$

Continue one by grouping three Taylor terms you get.

$$r_{n-1} x^2 \left(\frac{u_1 l_2 l_3 + u_1 u_2 l_3 \cdot x^2 + u_1 u_2 u_3 \cdot x^4}{l_1 l_2 l_3} \right)$$

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

$$r_1 = x, \quad r_{n+2} = r_{n-1} x^6 \frac{u_1 u_2 u_3}{l_1 l_2 l_3}$$

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, we recommend:

- The preferred method uses the Taylor series for arcsin(), 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.
- Use a moderate number of argument reductions, as calculating them can be very time-consuming. (This involves a division and calculation of two square roots.)

Arccos(x):

To find Arccos(x) we used the identity:

$$\text{Arccos}(x) = \frac{\pi}{2} - \text{Arcsin}(x) \quad (114)$$

There is not much else you can do.

The Math behind arbitrary precision

Arctan(x) using the Taylor series

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

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:

$$\text{Arctan}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \dots, \text{ where } |x| \leq 1 \quad (115)$$

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

$$\text{Arctan}(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (116)$$

k number of times until x is sufficiently low.

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

We calculate the reducing factor, k, as:

$$2 \cdot \lceil \ln(2) * \ln(\text{precision}) \rceil \quad (117)$$

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 formula:

$$\text{Arctan}(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (118)$$

It requires one division, one square root, and two additions. 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 by 2^k to find our arctan(x) result. 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:

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

or

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

Let us find the arctan(0.3) to see how this algorithm works. After the 13th Taylor term, the errors do not get lower, and the result is ~ 0.291456794477867 .

The Math behind arbitrary precision

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	0.3000000000000000	0.3000000000000000	8.54E-03
2	9.00E-03	0.2910000000000000	0.2910000000000000	4.57E-04
3	4.86E-04	0.2914860000000000	0.2914860000000000	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

If we take a two-argument reduction, we reduce 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 for reducing the number of Taylor terms required when precision is increased.

The issue with arbitrary precision

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

$$\frac{x^{2n-1}}{2n-1} < 10^{-P} \tag{119}$$

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

The Math behind arbitrary precision

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

$$(2n + 1) \ln(x) - \ln(2n + 1) = -P \cdot \ln(10) \quad (120)$$

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

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

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

$$n \approx \frac{-P \cdot \ln(10) - \ln(x)}{2 \cdot \ln(x)} \quad (121)$$

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

$$n \approx \frac{-P-M}{2 \cdot M} \quad (122)$$

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. For example, 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, which 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 observed that implementing coefficient scaling can typically yield two to three times better performance. If we try to group two Taylor terms to avoid a division, we get from the Taylor series for \arctan , where n denotes 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 as we advance:

The Math behind arbitrary precision

$$\begin{aligned}
 \text{Two Taylor terms: } & -\frac{x^{n-1}}{2n-1} + \frac{x^{n+1}}{2n+1} \Rightarrow \\
 & -\frac{(2n+1)x^{n-1} + (2n-1)x^{n+1}}{(2n-1)(2n+1)} \Rightarrow \\
 & x^{n-1} \cdot \frac{-(2n+1) + (2n-1)x^2}{(2n-1)(2n+1)}
 \end{aligned}$$

If we group three Taylor terms, we get the following:

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

We can continue grouping Taylor terms. From a practical point of view, grouping five Taylor terms is reasonable 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:

$$\text{Arctan}(x) = \sum_{n=0}^{\infty} \frac{2^{2n}(n!)^2}{(2n+1)!} \frac{z^{2n+1}}{(1+x^2)^{n+1}} \quad (123)$$

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

ArcTan(x) x=	Euler	Original 0.6	X Reduced 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

The Math behind arbitrary precision

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

As with the Taylor series, argument reduction significantly reduced the number of terms needed. For example, $\arctan(0.6)$ uses a reduction factor of four and requires only five terms (same as 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, using the Euler version of $\arctan(x)$ over the Taylor series version is not worth it.

Arctan(x) using Arcsin()

It could be interesting to use the identity:

$$\text{Arctan}(x) = \text{Arcsin}\left(\frac{x}{\sqrt{1+x^2}}\right) \quad (124)$$

Particularly if you want to reduce the size of your code and reuse existing code for the $\arcsin()$ function, 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, we recommend:

- The preferred method uses the Taylor series for $\arctan(x)$, argument reduction, and coefficient scaling.
- $\text{Arctan}(x)$ using the Euler series has no advantages over the Taylor series for argument $x < 0.4$. For argument $x > 0.4$, sticking with the Taylor series and using the recommended argument reduction and coefficient scaling for increased performance is more beneficial.

The Math behind arbitrary precision

- $\text{Arctan}(x)$ using $\text{arcsin}(x)$ is a slower alternative that can be used to simplify and reduce code size.
- Use a moderate number of argument reductions, as calculating them can be very time-consuming. (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 identities to calculate $\tanh(x)$, $\operatorname{arcsinh}(x)$, $\operatorname{arccosh}(x)$, and $\operatorname{arctanh}(x)$. This chapter will examine:

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

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

$\operatorname{Sinh}(x)$ using $\operatorname{Exp}(x)$

It is tempting to use the definition of $\operatorname{Sinh}(x)$:

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right) \quad (125)$$

We only need to calculate e^x once. In particular, 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:

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (126)$$

If you have implemented the above method for $\exp(x)$, you will experience slightly slower performance when using $\exp(x)$ to calculate $\sinh(x)$. Usually, $\sinh(x)$ is faster to compute than $\exp(x)$.

$\operatorname{Sinh}(x)$ using the Taylor series

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

$\operatorname{Sinh}(x)$ is found with the Taylor series:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (127)$$

Example $\sinh(1)$

We are calculating $\sinh(1)$ using no argument reduction. We need seven Taylor terms to get the result using the Taylor series.

$\operatorname{Sinh}(x)$	Original	X Reduced
x=	1	1
Taylor reductions=	0	

The Math behind arbitrary precision

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

The number of Taylor terms to reach a result does not seem too bad at first glance. In the previous examples, we only used approx. Seven Taylor terms. However, when dealing with higher precisions, e.g., 1,000, 10,000, or even 100,000 digits, we suddenly have to perform many 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 for the $\sin(x)$ as a function of the number of precision 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 $\sinh(x)$:

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

$$n \approx \frac{1}{2} \frac{(P-M) \cdot \ln(10)}{\ln(P-M) - 1 - M \cdot \ln(10)} - 1 \quad (128)$$

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								
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. 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 arguments of 10^{-9} . The factor is only around three for a precision of 100,000 digits; for 100M digits, it is around 2.2. The lesson is that argument reduction is more efficient for smaller than higher precision. However, overall argument reduction is beneficial at any precision. There is another approximation for n based on the actual value of

The Math behind arbitrary precision

x, rather than just its magnitude. It usually provides a little less of the required Taylor terms. This formula can be pretty helpful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (129)$$

Argument Reduction

Examining the Taylor series for $\sinh(x)$, it is evident that we prefer $|x| < 1$ to ensure the Taylor series converges more quickly. As we have seen before, we can use *argument reduction* to work with a smaller number to converge $\sinh(x)$ faster 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 by 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 compared 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	0.111111111111	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 eight times argument reduction, you get the result after two Taylor terms compared 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	

Unsurprisingly, using argument reduction significantly reduces the number of Taylor terms needed and results in faster performance.

Finding a reasonable argument reduction factor.

As the table above shows, a higher reduction factor significantly improves performance. However, how many times of argument 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

The Math behind arbitrary precision

did the reductions on the front end. $\sinh(3x)=3\sinh(x)+4(\sinh^3(x))$ taking $\sinh(x)$ out as a factor you get this: $\sinh(3x)=\sinh(x)(3+4(\sinh^2(x)))$ or one subtraction and three multiplication. Using

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (130)$$

Starting from $x=1$, you obtain, for $P=100$ digits, that the required Taylor term is 24. Performing three reductions, you obtain $x = 1/3^3 = 0.037$. We expect to use the above formula, requiring only 14 Taylor terms. Each Taylor term requires one addition/subtraction, 1 division, and one multiplication, which yields a total saving of 10 subtractions, 10 divisions, and 10 multiplications. Compared to three reductions on the front end, there are three divisions, and on the back end, three subtractions and nine multiplications, a total saving of seven subtractions/additions, one multiplication, and seven divisions. Since division is slower than multiplication and addition/subtraction, we can give a rough saving equivalent of seven divisions. For higher precision, the savings become larger.

We automatically calculate the reduction factor as follows:

$$k = 8 \left\lceil \frac{2}{3} \ln(2) * \ln(P) \right\rceil \quad (131)$$

For higher precision, we made adjustments for the magnitude of x . We add the exponent to the reduction factor. If x is large, we do more argument reductions; if x is small, we reduce the number of reductions. This means that our reduction factor becomes smaller as x approaches zero, preventing us from making unnecessary reductions. If x is very small, the reduction factor is negative, and we do not perform any argument reductions. For example, for $P=100$, you get 24, and for $P=10,000$, you get 40. To compensate for inaccuracies when adding the front and back end calculations, we increase the precision by the reduction factor, $k/4$. The increased precision generates only a minor performance penalty compared to the extra savings in Taylor's terms of the overall calculation.

To calculate a reasonable reduction factor, we make it a function of the desired precision and the magnitude of the argument x . For example, argument reduction increased as a logarithmic function of the wanted precision, and argument reduction increased with a large magnitude of the number and decreased for a smaller magnitude of argument x .

Guard Digits

When summarizing a Taylor series as $\sinh(x)$, you need quite a lot of summarizing, which will produce round-off errors.

For our $\sinh(x)$ function, we use a simple guard digit calculation that we add.

$2 + \lceil \log_{10}(\text{precision}) \rceil$ As extra guard digits for working precision.
--

Further improvements of the method?

Coefficient scaling can be considered an improvement over the standard method.

Generally speaking, you can have three options here.

- No coefficient scaling

The Math behind arbitrary precision

- Partial coefficient scaling
- Full coefficient scaling

No coefficient scaling

No coefficient scaling is just the regular use of the Taylor series, as presented above. Each Taylor term is computed, including the division, which performs poorly compared to the other operators, particularly in arbitrary precision libraries.

Although the method is straightforward, it ensures that each term is as accurate as possible before contributing to the final sum. It can be computationally expensive due to repeated division operations, mainly when used in connection with arbitrary precision libraries.

Partial coefficient scaling

You can group more Taylor terms before performing the division. For example, you reduce the number of division operations by calculating five Taylor terms at a time and then dividing the sum of these terms by the appropriate factorial. Partial coefficient scaling will enhance performance since division is generally more computationally intensive than multiplication or addition. The trade-off here is a potential slight decrease in numerical precision, as errors could accumulate in the grouped terms before the division normalizes them.

Full coefficient scaling

When doing a full coefficient scaling, you postpone the division until all Taylor terms have been computed. This method involves summing all scaled Taylor series terms first and dividing the final sum by the factorial. This method may be the fastest in reducing the number of division operations, but it also carries the risk of the most significant errors. When terms with large magnitudes are summed without immediate normalization, floating-point errors can accumulate, particularly for higher-order terms or larger values of x .

The discussion of partial and full coefficient scaling is as follows.

Partial coefficient scaling

The same technique for coefficient scaling (grouping of Taylor terms) can also be applied to the $\sinh(x)$ here. Consider the Taylor series for the sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (132)$$

The issue, again, is clearly the division for each term. Since division is many times slower than calculation and addition, you could group two or more Taylor terms (sometimes called coefficient scaling) and reduce the number of divisions. Consider the n^{th} and the $n+1$ terms:

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

Moreover, group them:

$$\begin{aligned} \dots \frac{(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \dots & \Rightarrow \\ \dots \frac{(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \dots & \end{aligned}$$

The Math behind arbitrary precision

Then, you have replaced one division with two extra multiplications. 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. You can do that for three terms:

For grouping three Taylor terms, you get:

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

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

In the source code [19], we utilize five Taylor terms simultaneously.

Full coefficient scaling

Now, why stop with the grouping of only a few Taylor terms? In [34], they devised a new computation of the Taylor series (albeit for the e^x Taylor series), postponing the division until all Taylor terms had been calculated.

We can use the same rationale in [34] for the $\sinh(x)$. By noticing that by evaluating the Taylor series backward, you can set this recursion:

$$n! \sinh(x) = x(n! + \dots + ((n(n-1)(n-2) + (n(n-1) + x^2)x^2)x^2 \dots)) \quad (133)$$

Here, you summed up the series and postponed the division to one final division to calculate $\sinh(x)$. This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Fortunately, determining the required number of Taylor terms is not a difficult task. We are using the Sterling approximation for the factorial. We can write the error terms needed for a given decimal precision P .

$$(n+0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (134)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

$$f(y) = (n+0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (135)$$

The Math behind arbitrary precision

Where $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$. Accuracy is not essential here for high accuracy.

And $f'(y)$:

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (136)$$

Applying Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (137)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + 1 - (n_i-1) - \ln(|x|) - 1} \quad (138)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (139)$$

Since we require an integral number of Taylor terms that must be odd, we don't need to carry that much precision. As a starting point, [34] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (140)$$

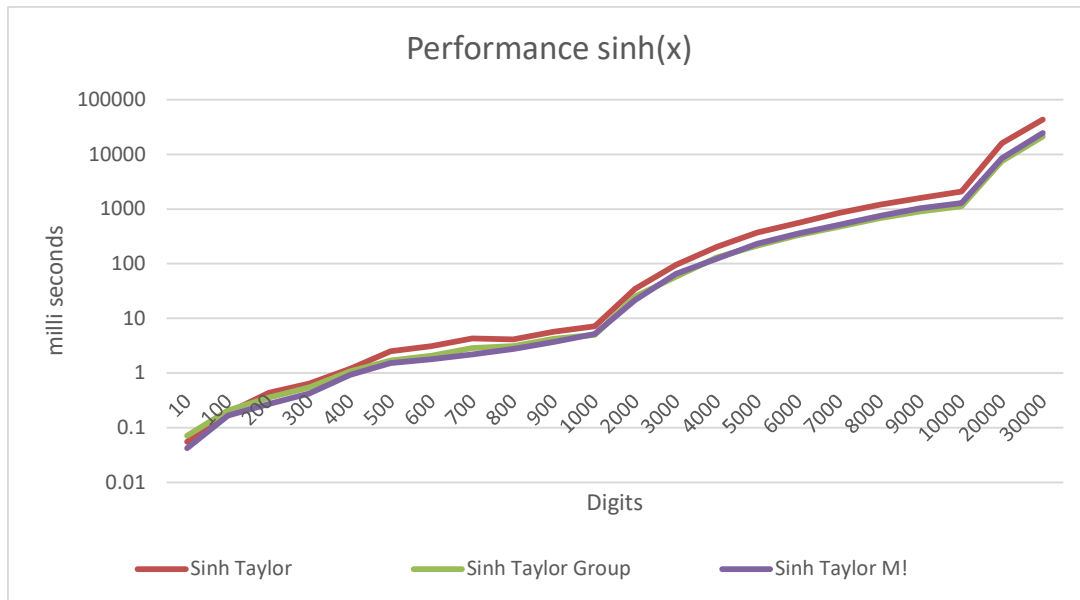
The starting formula works well for the Newton iteration since $0 < x < 1$ due to argument reduction. Since we are only interested in the ceiling n_{i+1} , we will only need a few iterations to find the number of Taylor terms required.

We can, therefore, replace the previous code for coefficient scaling for five Taylor terms with this one.

sinh(x) Performance

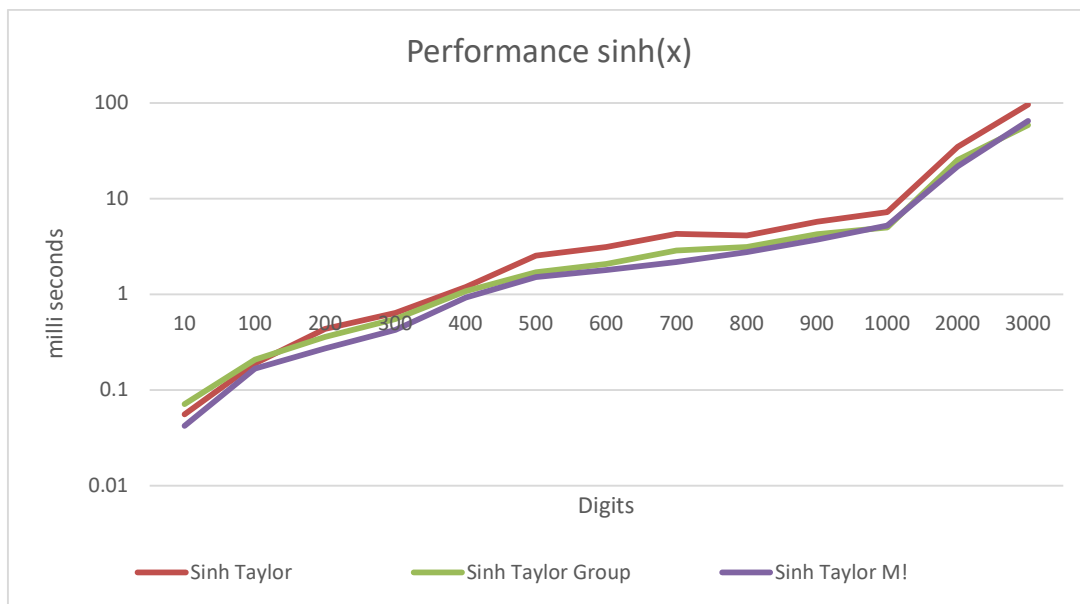
Below is the performance of the three mentioned sinh(x) computation methods. The performance chart shows the performance in milliseconds for a precision ranging from 10 decimals to 30,00 digits. The y-axis is on a logarithmic scale, and computing one Taylor term at a time is considerably slower than the other methods (such as the sinh Taylor method). Roughly half the speed of the Taylor series, grouping five terms at a time, and the Taylor series, where we postponed the division until after the computation of the Taylor terms. (sinh Taylor M!)

The Math behind arbitrary precision



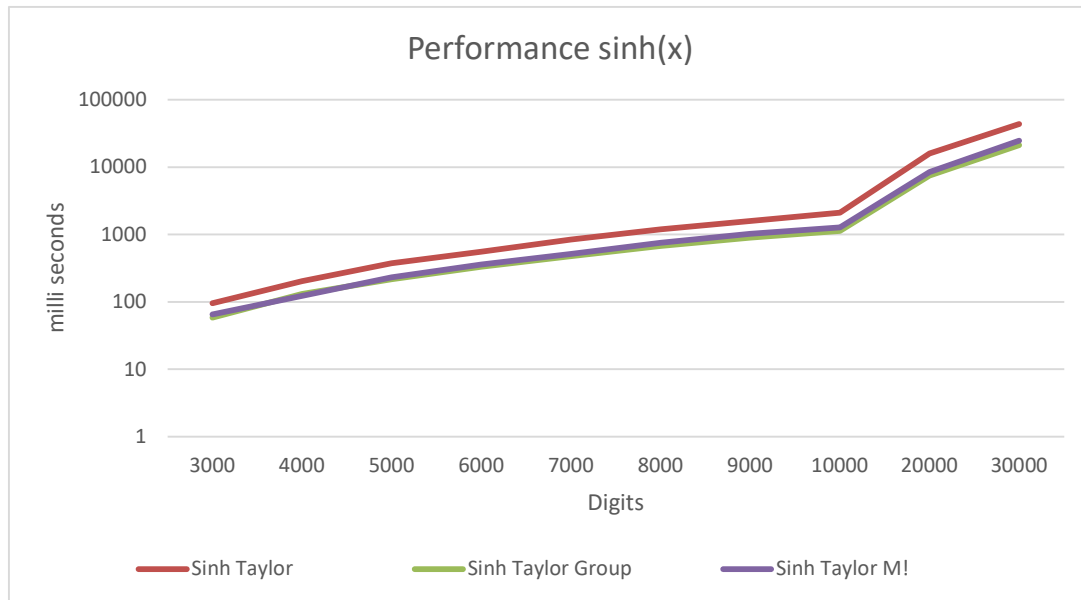
Sinh (x) performance between 10 and 30,000 decimal digits of computation.

The performance between partial coefficient scaling (Taylor Group) and full coefficient scaling (Taylor M!) is much closer. Taylor M! is fastest up to approximately 1,000-2,000 decimal digits, whereas Taylor Group takes over. However, only around 10% faster.



Sinh (x) performance between 10 and 3,000 decimal digits of computation.

The Math behind arbitrary precision



Sinh(x) performance between 3,000 and 30,000 decimal digits of computation.

Recommendation for calculating sinh (x)

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

- Use a standard Taylor series for sinh(x) with an aggressive reduction factor to speed up the calculation of Taylor terms.
- Use coefficient scaling to increase performance.
 - a. Either use the full coefficient scaling or
 - b. The partial coefficient scaling.
- The sinh(x) full coefficient scaling is the fastest to around 1,000-2,000 decimal digits.
- The sinh(x) partial coefficient scaling is fastest above 1,000-2,000 decimal digits.

Cosh(x) using Exp(x)

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

$$\cosh(x) = \frac{1}{2}(e^x + e^{-x}) = \frac{1}{2}\left(e^x + \frac{1}{e^x}\right) \quad (141)$$

We only need to calculate e^x once. 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 compute than using the Taylor series for $\exp(x)$.

Cosh(x) using Taylor series:

For $\cosh(x)$, we again use a Taylor series until any additional terms do not change the result to the given precision of the number.

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{ for any real value } x \quad (142)$$

Example $\cosh(1)$:

We are calculating $\cosh(1)$ using no argument reduction. We need nine Taylor terms to get the result using the Taylor series.

The Math behind arbitrary precision

Cosh(x)		Original	X Reduced		
x=		1	1		
Taylor reductions=		0			
Terms	Term value	Taylor sum	Cosh(x)	Error	
1	1.00E+00	1	1.000000000000000	5.43E-01	
2	5.00E-01	1.5	1.500000000000000	4.31E-02	
3	4.17E-02	1.54166667	1.541666666666667	1.41E-03	
4	1.39E-03	1.54305556	1.543055555555556	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	

Most issues arising in arbitrary precision for sinh (x) also apply to cosh(x).

Argument Reduction

Examining the Taylor series for cosh(x), it is evident that we prefer $|x| < 1$ to ensure the Taylor series converges more quickly. As we have seen before, we can use *argument reduction* to work with a smaller number, resulting in faster convergence of cosh(x) using fewer *Taylor terms* of the Taylor series.

We can use the trisection identity:

$$\cosh(3x) = \cosh(x)(4\cosh^2(x) - 3) \tag{143}$$

To reduce the argument by a factor of three, 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 compared 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	1.000000000000000	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 eight times argument reduction, you get the result after two Taylor terms compared to five using two argument reductions. However, the error is considerably higher (less accurate) than the equivalent calculation for sinh (1).

The Math behind arbitrary precision

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	1.000000000000000	5.43E-01	
2	1.16E-08	1.00000001	1.54308063305537	1.76E-09	

Unsurprisingly, using argument reduction significantly reduces the number of Taylor terms needed and results in faster performance. However, aggressive reductions in argument result in a significant decrease in accuracy. This is due to the trisection identity, and the Taylor series is approaching one for a small argument, resulting in a higher loss of accuracy unless precautions are taken. To avoid inaccuracy in the result, we increase the precision by 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 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:

$$\cosh(2x) = \cosh^2(x) - 1 \quad (144)$$

Alternatively, it is even better written as:

$$\cosh(2x) = 2(1 - \cosh(x))^2 - 4(1 - \cosh(x)) + 1 \quad (145)$$

Although it does not prevent round-off errors, it is less sensitive than the trisection formula. We calculate the reduction factor for $\cosh(x)$ as $k = 8 \lceil \ln(2) * \ln(P) \rceil$. For higher precision, we adjust for the magnitude of x . We add the exponent to the reduction factor. This means that our reduction factor becomes smaller as x approaches zero, preventing us from making unnecessary reductions. If x is very small, the reduction factor is negative, and we do not perform any argument reductions.

Since we are slightly less sensitive using the double angle formula compared to the trisection formula, we only increase the precision by $0.75 \times k$.

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 $\cosh(x)$.

Cosh(x) with full coefficients scaling

As we saw with $\sinh(x)$, we can also apply the full coefficient scaling to $\cosh(x)$. $\cosh(x)$ is very similar to the $\sinh(x)$.

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{ for any real value } x \quad (146)$$

By evaluating the Taylor series backward, you can set this recursion:

$$n! \cosh(x) = n! + \dots + ((n(n-1)(n-2)(n-2) + (n(n-1) + x^2)x^2)x^2) \dots \quad (147)$$

The Math behind arbitrary precision

Here, you summed up the series and postponed the division to one final division to calculate $\cosh(x)$. This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Fortunately, determining the required number of Taylor terms is not a difficult task. We are using the Sterling approximation for the factorial. We can write the error terms needed for a given decimal precision P .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (148)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (149)$$

Where $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$. We don't need to carry this with high precision.

And $f'(y)$:

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (150)$$

Applying Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (151)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (152)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (153)$$

Since we require an integral number of Taylor terms that must be even, we don't need to carry that much precision. As a starting point, [34] suggested.

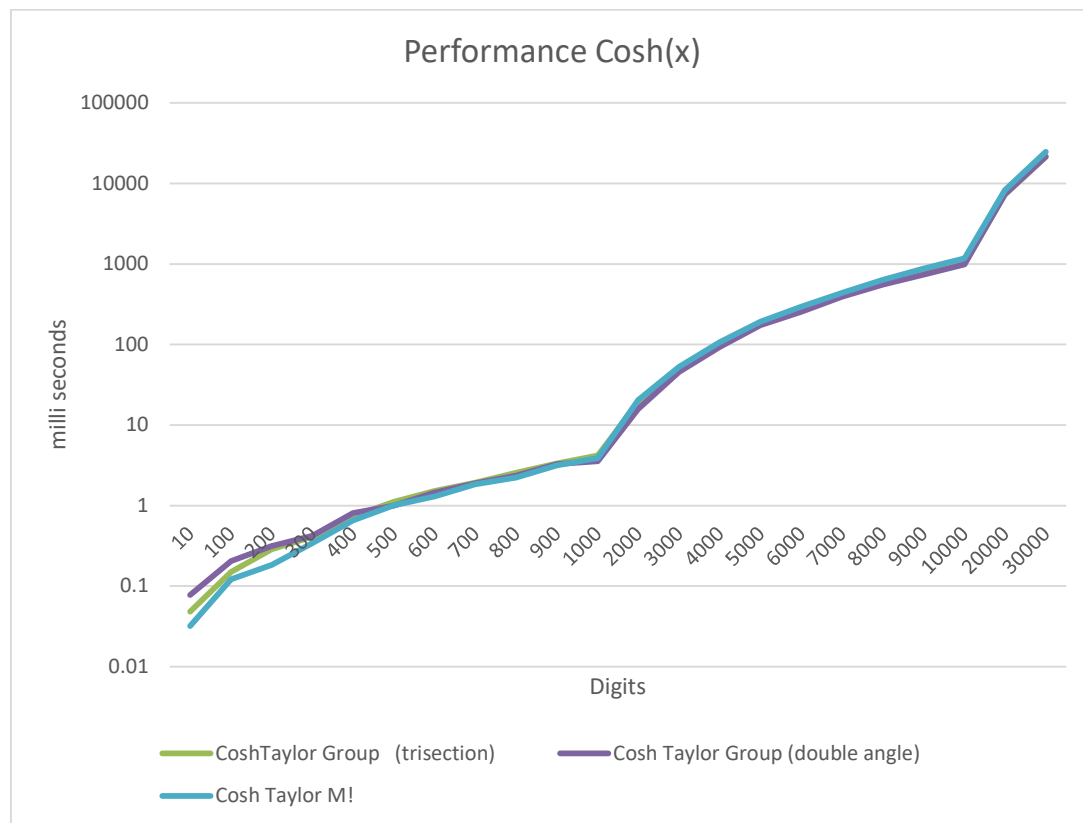
$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - 1) \cdot (|x|) - 1 \cdot (|-\ln(|x|)|)} \quad (154)$$

The Math behind arbitrary precision

The starting formula works well for the Newton iteration since $0 < x < 1$ due to argument reduction. Since we are only interested in the ceiling n_{i+1} , we will only need a few iterations to find the number of Taylor terms required.

We can, therefore, replace the previous code for coefficient scaling for five Taylor terms with this one for $\cosh(x)$ using full coefficient scaling.

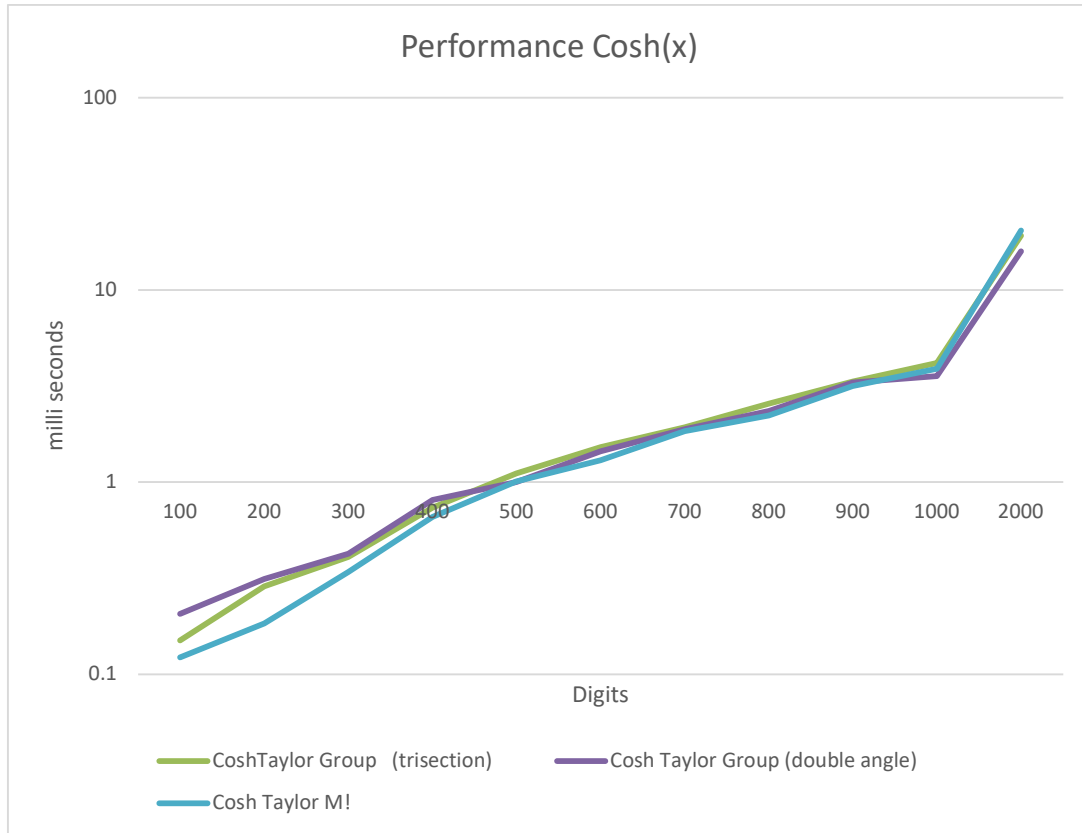
Cosh(x) Performance



Cosh(x) performance for digits in the range of 10 to 30,000.

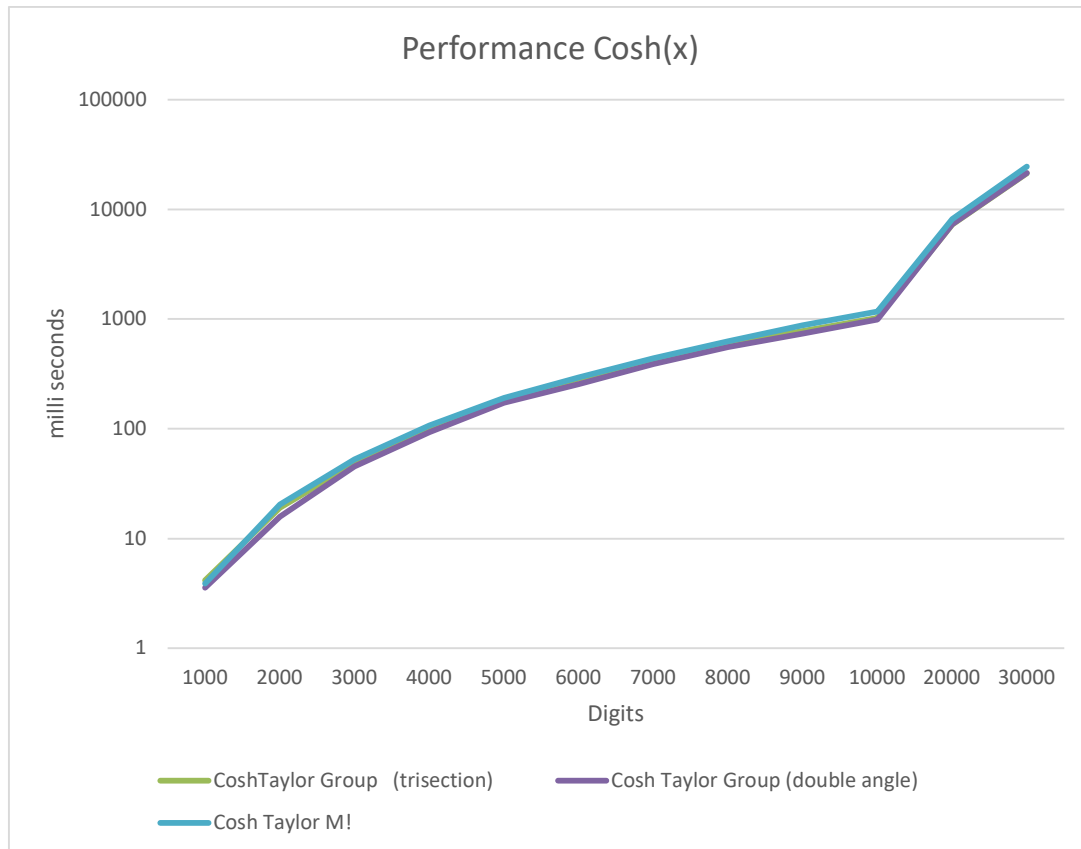
It is hard to see any difference between Cosh Taylor Group and $\cosh(x)$ Taylor M!

The Math behind arbitrary precision



Cosh(x) in the range of 100 to 2,000 digits.

The Math behind arbitrary precision



Cosh(x) in the range from 1,000 to 30,000 digits.

Recommendation for calculating cosh(x)

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

- It is a matter of taste whether to use the cosh(x) using the double angle formula or the trisection formula, since the performance is equivalent.
- Do not use the Taylor series for cosh(x) with an aggressive reduction factor to accelerate the calculation of Taylor terms.
- Use coefficient scaling to enhance performance—particularly full coefficient scaling for values below 1,000 digits and partial coefficient scaling for values above 1,000 digits.

Tanh(x)

Tanh(x) is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (155)$$

Which seems to be the most effective way of calculating tanh(x) using a single call for exp(x)?

We could also use the Taylor series for tanh(x):

The Math behind arbitrary precision

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \dots - \frac{(-1)^{n-1} \cdot 2^{2n} (2^{2n}-1) \beta_n x^{2n-1}}{(2n)!} + \dots \quad (156)$$

Where B_n is the Bernoulli number, however, since we do not know how many Bernoulli numbers we need, this requires us to calculate Bernoulli numbers on the fly; therefore, it is significantly more complicated to implement than simply calling 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 and the method using the Taylor series.

Arcsinh(x) direct method

Arcsinh(x) is equal to:

$$\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1}) \quad (157)$$

$\ln(x)$ is relatively fast to calculate, and the same goes for the square root. This direct method is the preferred approach for calculating $\text{arsinh}(x)$.

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 $|x| < 1$:

$$\text{Arcsinh}(x) = x - \frac{1}{2} \frac{x^3}{3} + \frac{1 \cdot 3}{2 \cdot 4} \frac{x^5}{5} - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{x^7}{7} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{x^9}{9} - \dots \quad (158)$$

The second Taylor series is for $|x| \geq 1$:

$$\text{Arcsinh}(x) = \pm \ln(2x) + \frac{1}{2} \frac{1}{2x^2} - \frac{1 \cdot 3}{2 \cdot 4} \frac{1}{4x^4} + \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{1}{6x^6} - \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{1}{8x^8} - \dots \quad (159)$$

Where the + applies for $x \geq 1$ and - for $x \leq -1$.

Example: $\text{Arcsinh}(0.1)$

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	0.10000000000000	0.1000000000000000	-1.66E-04
2	1.67E-04	0.09983333333333	0.0998333333333333	7.46E-07
3	7.50E-07	0.09983408333333	0.0998340833333333	-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

The Math behind arbitrary precision

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 $\sim 10^{-12}$)

ArcSinh(x)		Original		$ x < 1$
	x=	0.7		
Terms	Term value	Term Sum	ArcSinh(x)	Error
1	7.00E-01	0.700000000000000	0.700000000000000	-4.73E-02
2	5.72E-02	0.642833333333333	0.642833333333333	9.83E-03
3	1.26E-02	0.655438583333333	0.655438583333333	-2.77E-03
4	3.68E-03	0.651762052083333	0.651762052083333	9.05E-04
5	1.23E-03	0.652988073129340	0.652988073129340	-3.22E-04
6	4.42E-04	0.652545702444649	0.652545702444649	1.21E-04
7	1.68E-04	0.652713831661927	0.652713831661927	-4.73E-05
8	6.63E-05	0.652647532707247	0.652647532707247	1.90E-05
9	2.69E-05	0.652674405721047	0.652674405721047	-7.84E-06
10	1.11E-05	0.652663278564660	0.652663278564660	3.29E-06
11	4.69E-06	0.652667964952025	0.652667964952025	-1.40E-06
12	2.00E-06	0.652665963605294	0.652665963605294	6.02E-07
13	8.65E-07	0.652666828220438	0.652666828220438	-2.62E-07
14	3.77E-07	0.652666451029002	0.652666451029002	1.15E-07
15	1.66E-07	0.652666616960717	0.652666616960717	-5.09E-08
16	7.35E-08	0.652666543435125	0.652666543435125	2.26E-08
17	3.28E-08	0.652666576221552	0.652666576221552	-1.01E-08
18	1.47E-08	0.652666561519732	0.652666561519732	4.56E-09
19	6.63E-09	0.652666568144933	0.652666568144933	-2.06E-09
20	3.00E-09	0.652666565146113	0.652666565146113	9.36E-10
21	1.36E-09	0.652666566508912	0.652666566508912	-4.27E-10
22	6.22E-10	0.652666565887360	0.652666565887360	1.95E-10
23	2.84E-10	0.652666566171770	0.652666566171770	-8.94E-11
24	1.31E-10	0.652666566041240	0.652666566041240	4.11E-11
25	6.01E-11	0.652666566101311	0.652666566101311	-1.90E-11
26	2.77E-11	0.652666566073596	0.652666566073596	8.76E-12
27	1.28E-11	0.652666566086412	0.652666566086412	-4.06E-12

There are several issues with the Taylor series approach.

- First, the Taylor series converges slowly as x approaches 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:

The Math behind arbitrary precision

- Use the Direct method: $\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$

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:

$$\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1}), \text{ where } x \geq 1 \quad (160)$$

$\ln(x)$ is relatively fast to calculate, and the same goes for the square root. This direct method is the preferred way of calculating $\text{Arccosh}(x)$.

Arccosh(x) using the Taylor series

Arccosh(x) also have a Taylor series equivalence for $|x| \geq 1$:

$$\text{Arccosh}(x) = \ln(2x) - \left(\frac{1}{2} \frac{1}{2x^2} + \frac{1 \cdot 3}{2 \cdot 4} \frac{1}{4x^4} + \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{1}{6x^6} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{1}{8x^8} - \dots \right) \quad (161)$$

However, it converges just as slowly as the $\text{Arcsinh}(x)$.

Example: $\text{Arccosh}(5)$ with argument reduction

A result is found using nine Taylor terms.

ArcCosh(x)		Original	x>=1	
Terms	x=	5	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: $\text{Arccosh}(2)$ with argument reduction

A result is found using twenty-one Taylor terms

ArcCosh(x)		Original	x>=1	
Terms	x=	2	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

The Math behind arbitrary precision

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 converges slowly when x approaches 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, we recommend:

- Use the Direct method: $\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$

Arctanh(x)

There are two interesting methods to use. One is the standard Taylor series, and the other is the direct method.

Arctanh(x) direct method

Arctanh(x) is equal to:

$$\text{Arctanh}(x) = \frac{1}{2} \ln \left(\frac{1+x}{1-x} \right), \text{ where } |x| < 1 \quad (162)$$

$\ln(x)$ 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:

The Math behind arbitrary precision

$$\text{Arctanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} - \dots, \text{ where } |x| < 1 \quad (163)$$

Notice the similarity with the Taylor series of $\arctan(x)$. $\text{Arctanh}(x)$ does not use any alternating signs between Taylor terms as the $\arctan(x)$ Taylor series does.

$\text{Arctanh}(x)$ suffers from the same weakness as the other hyperbolic functions, in that there is no argument reduction formula to lower the argument, x , and increase the performance of the Taylor series.

The $\text{Arctanh}(x)$ Taylor series converges slowly and particularly close to 1 or -1.

Example $\text{Arctanh}(0.1)$

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	0.1000000000000000	0.1000000000000000	3.35E-04	
2	3.33E-04	0.1003333333333333	0.1003333333333333	2.01E-06	
3	2.00E-06	0.1003353333333333	0.1003353333333333	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 $\text{Arctanh}(0.5)$

Now we need 23 Taylor terms to get the result. Which is more than three times as many as for $\text{arctanh}(0.1)$.

ArcTanh(x)		Original		x <1	
	x=	0.5			
Terms	Term value	Term Sum	ArcTanh(x)	Error	
1	5.00E-01	0.5000000000000000	0.5000000000000000	4.93E-02	
2	4.17E-02	0.5416666666666667	0.5416666666666667	7.64E-03	
3	6.25E-03	0.5479166666666667	0.5479166666666667	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	

The Math behind arbitrary precision

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 up the process. 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 $\text{Arcsinh}(x)$ and $\text{Arccosh}(x)$, but not as badly from a performance perspective.

- First, the Taylor series converges slowly when x approaches 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 $\text{Arctanh}(x)$

Based on the performance measure of the various $\text{arctan}(x)$ methods, recommend:

$$\text{Use the direct method. } \text{Arctanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right), \text{ where } |x| < 1 \quad (164)$$

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 $\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$ for calculating $\text{arsinh}(x)$
- Use $\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$ for calculating $\text{arccosh}(x)$
- Use $\text{Arctanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$ for calculating $\text{arctanh}(x)$

Gamma function

The Gamma function, denoted by $\Gamma(z)$, is one of the most important special functions in mathematics. It extends the factorial function to complex and real arguments. While the factorial $n!$ is only defined for non-negative integers, the Gamma function is defined for all complex numbers except the non-positive integers, where it has simple poles. For positive integers n , the Gamma function satisfies $\Gamma(n)=(n-1)!$. So, it can be regarded as a natural generalization of the factorial.

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

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt \quad (165)$$

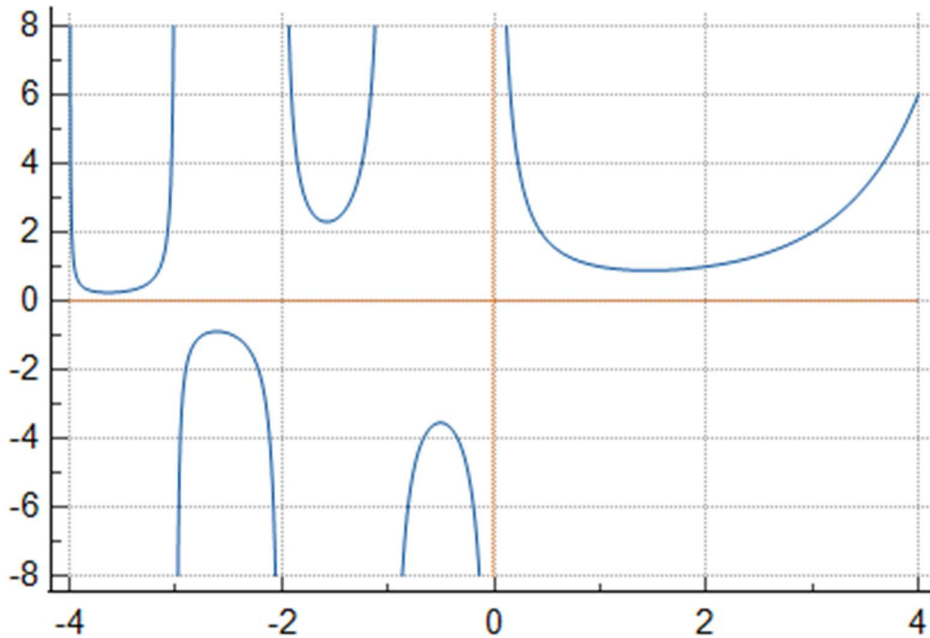


Figure 3. Gamma function in the interval [-4;+4]

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

$$\Gamma(z + 1) = z\Gamma(z) \quad (166)$$

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

$$\Gamma(n) = (n - 1)! \quad (167)$$

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

$$\Gamma\left(\frac{1}{2} + n\right) = \frac{(2n)!}{2^{2n}n!} \sqrt{\pi} \text{ for } n \geq 0 \quad (168)$$

The Math behind arbitrary precision

Or in the negative half of the real axis:

$$\Gamma\left(\frac{1}{2} - n\right) = (-1)^n \frac{2^{2n} n!}{(2n)!} \sqrt{\pi} \text{ for } n > 0 \quad (169)$$

Another important equation is Euler's reflection formula:

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(z\pi)} \Rightarrow \quad (170)$$

$$\Gamma(z) = \frac{\pi}{\Gamma(1-z)\sin(z\pi)} \quad (171)$$

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 (168) and (169).
- 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 17

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 integration by parts is a local method defined on the interval $[1, 2]$. Of course, there are techniques to expand the local method into a global method.

Lanczos-Spouge method

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

$$\Gamma(x) = \frac{(x+a)^{x+\frac{1}{2}}}{e^{x+a}} \left(c_0 + \sum_{k=1}^{a-1} \frac{c_k}{x+k} \right) \quad (172)$$

The Math behind arbitrary precision

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:

$$a = \left[\left(P - \frac{\ln(P)}{\ln(10)} \right) \frac{\ln(10)}{\ln(2\pi)} - \frac{1}{2} \right] \quad (173)$$

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 state that the error is bounded by:

$$e_a(x) = \frac{1}{a^{0.5}(2\pi)^{a+0.5}} \quad (174)$$

For a given precision P , the variable a is:

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

Lanczos-Spouge Approximation	
Pros	Cons
Accuracy is easy to control and maintain	Need to compute π , e^x , and $\sqrt{\quad}$
No need to shift/de-shift the Gamma value	
Fast Method	

Stirling asymptotic series method

The Stirling asymptotic series for the Gamma function is given by:

$$\ln(\Gamma(x)) \sim \left(x - \frac{1}{2}\right) \ln(x) - x + \frac{1}{2} \ln(2\pi) + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n(2n-1)x^{2n-1}} \quad (175)$$

Where B_{2n} is the Bernoulli number. In Yacas [6], they find that the optimal value for n in the summation is given by:

$$n_{\text{optimal}} \sim \pi|x| + 2 \quad (176)$$

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

$$(2\pi - 1)x + (x + 1)\ln(x) + 3.9 > P_{\text{max}} \cdot \ln(10) \Rightarrow \quad (177)$$

The Math behind arbitrary precision

$$P_{max} = \left\lceil \frac{(2\pi-1)x+(x+1)\ln(x)+3.9}{\ln(10)} \right\rceil \quad (178)$$

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 obtain a reasonable precision with this method. To circumvent this deficit, we can use the recurrence. $x\Gamma(x) = \Gamma(x + 1)$, M number of times to increase the magnitude. For example, if $|x|$ is one and we need the result with 35 digits, the magnitude of $|x|$ needs to be more than 10, as indicated in the table above. Instead of calculating $\Gamma(x)$, we calculate $\Gamma(x + 10)$ And then divide $\Gamma(x + 10)$ 10 times as outlined:

$$\Gamma(x) = \frac{\Gamma(x+10)}{x(x+1)(x+2)(x+3)(x+4)(x+5)(x+6)(x+7)(x+8)(x+9)} \quad (179)$$

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

$$\Gamma(x) = \frac{\Gamma(x+M)}{\prod_{m=0}^{M-1}(x+m)} \quad (180)$$

Unfortunately, the above formula for P_{max} is not very accurate in determining the number of shifts needed and, in general, indicates a shifting that is insufficient for the desired accuracy. Instead, we use [22], which states the number of shifts needed as follows.

$$|x + shifts| = P * \frac{\ln(10)}{\ln(2)} * 0.11038 => \quad (181)$$

$$shifts = P * \frac{\ln(10)}{\ln(2)} * 0.11038 - |x| \quad (182)$$

This formula works both ways. If $|x|$ is small, the shift is positive. If $|x|$ is large, the shift is negative. This is a significant benefit, as we can utilize it in both ways to reduce the argument and thereby decrease the number of terms required for the Σ (also reducing the number of Bernoulli numbers that need to be computed).

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

Integration by parts method

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

The Math behind arbitrary precision

$$\Gamma(x) = \int_0^M t^x e^{-t} \frac{dt}{t} + \int_M^\infty t^x e^{-t} \frac{dt}{t} \quad (183)$$

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:

$$\Gamma(x) \approx \int_0^M t^x e^{-t} \frac{dt}{t} = M^x e^{-M} \sum_{n=0}^{\infty} \frac{M^n}{\prod_{k=0}^n (x+k)} \quad (184)$$

The question now is how to choose an appropriate M in Yacas [6]. They find the condition to be that:

$$M > (P + \ln(P)) \ln(10) \quad (185)$$

The only thing missing now is the number of terms (N_{max}) of the series you need to calculate. Again, in [6], they find that to be:

$$N_{max} = P \frac{\ln(10)}{W\left(\frac{1}{e}\right)}, \text{ where } W \text{ is Lambert's function } W\left(\frac{1}{e}\right) \approx 0.2785 \quad (186)$$

With that, we have the final formula:

$$\Gamma(x) \approx M^x e^{-M} \sum_{n=0}^{N_{max}} \frac{M^n}{\prod_{k=0}^n (x+k)} \quad (187)$$

The only issue remaining to be addressed is ensuring that x falls within the specified 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.

$$\Gamma(x) = \prod_{m=0}^{k-1} (x - m) \Gamma(x - k) \quad (188)$$

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 that the method using integration by parts is the most effective one. It consistently outperforms the Stirling and Lanczos-Spouge methods.

The Math behind arbitrary precision

Notice that Stirling is measured with a small, medium, and large argument. The small argument is approximately 20 times slower than for the medium and large arguments. This is deceptive, and the reason is that Stirling-small was first computed, which builds up the Bernoulli numbers that are cached. Therefore, when Stirling-Medium and Stirling-large are called, 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. Even when cached, the Stirling method was significantly slower than the two other methods.

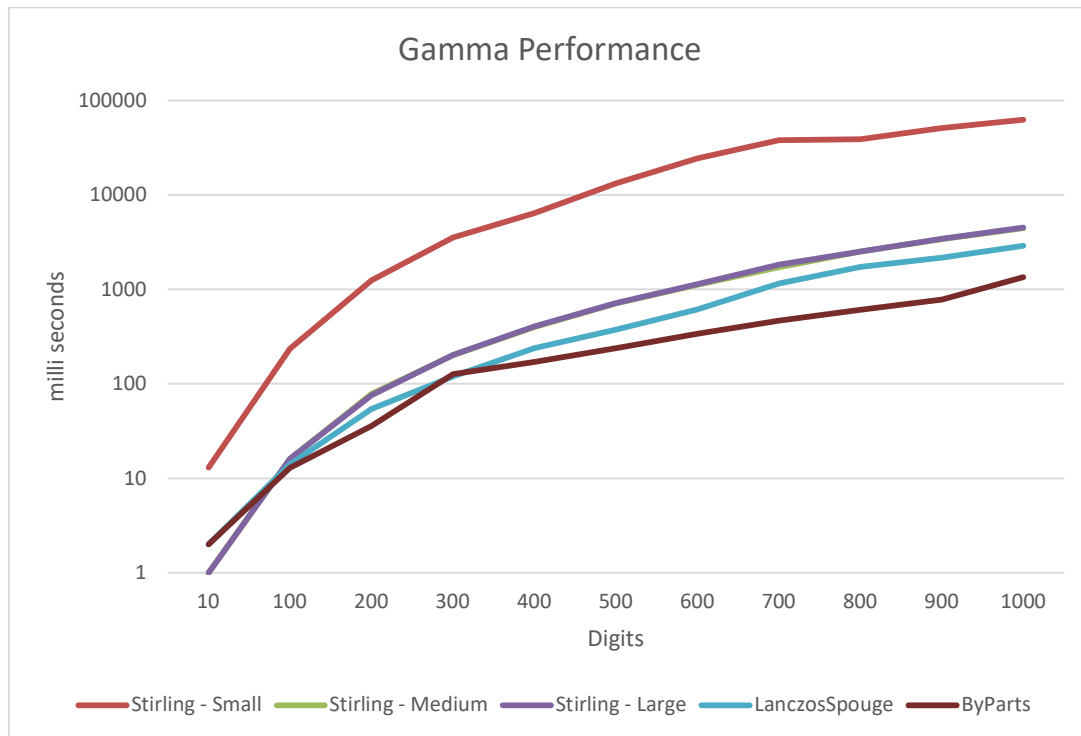


Figure 4. Gamma Performance

Recommendation for the Gamma function

The method of integration by parts is the simplest and fastest method, even when it relies on the shifting technique to accommodate x -values outside the interval $[1, 2]$. In second place is the Lanczos-Spouge method, followed by the Stirling asymptotic method in third place. Because the Stirling Asymptotic method relies on the Bernoulli numbers, it is not recommended, even if the Bernoulli numbers are 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, sometimes called Euler's first integral, is a classical special function that plays a central role in analysis and applications. It is defined for complex numbers z, w with positive real parts and extends naturally through analytic continuation. Historically, the Beta function arose from Euler's study of integrals and later found deep connections to probability theory, combinatorics, and statistics, where it appears as the normalization constant of the Beta distribution.

The Math behind arbitrary precision

Like the Gamma function, the Beta function provides a unifying framework for identities involving factorials and binomial coefficients. In fact, many binomial identities can be rewritten in terms of the Beta function, highlighting its close ties to discrete mathematics. The Beta function is also symmetric in its two arguments, reflecting a balance between the two terms in its integral representation.

The integral defines the beta function:

$$B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1} dt \quad (189)$$

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

A particularly elegant feature is its relationship to the Gamma function. Through the identity

$$B(z, w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \quad (190)$$

The Beta function inherits many of the Gamma function's properties. This formula allows for efficient computation in modern software, since most numerical libraries already provide highly optimized implementations of the Gamma function (e.g., `tgamma` in C and C++). Thus, in practice, the Beta function is often not computed directly from its defining integral, but rather via Gamma evaluations.

The Error function

The error function, commonly denoted as $\text{erf}(x)$, is one of the standard special functions of mathematics. It arises naturally in probability theory, statistics, and partial differential equations. In particular, $\text{erf}(x)$ is closely tied to the normal distribution, since it describes the probability that a normally distributed random variable with mean zero and variance $\frac{1}{2}$ lies within a given interval. The function also appears in solutions of diffusion and heat-conduction problems, where Gaussian integrals are involved.

Formally, the error function is defined by the integral:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (191)$$

This normalization ensures that $\text{erf}(\infty)=1$ and $\text{erf}(-\infty)=-1$. Because it cannot be expressed in terms of elementary functions, $\text{erf}(x)$ is treated as a special function, with many numerical methods developed for its evaluation.

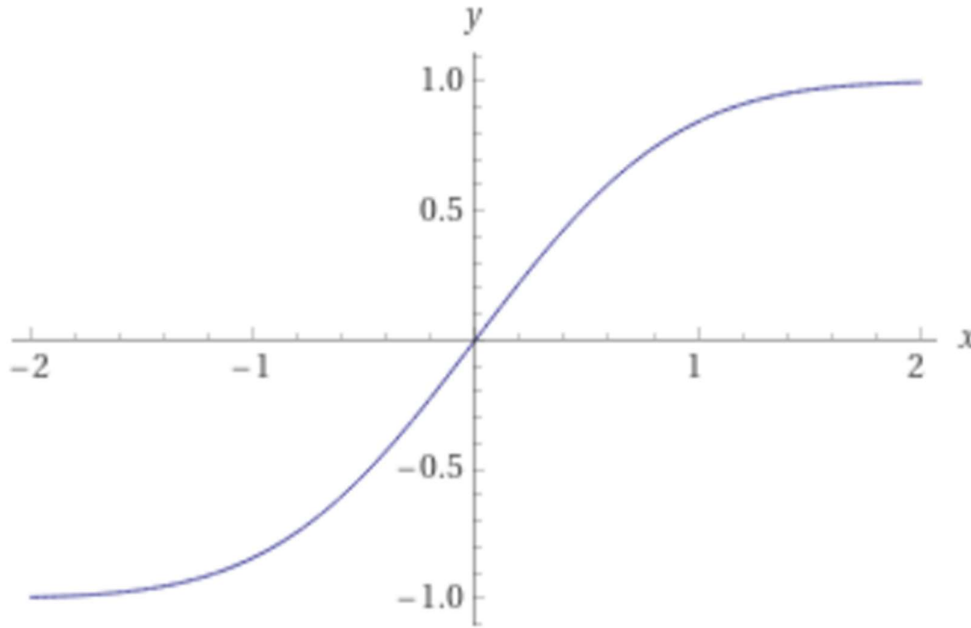


Figure 5. Error function in the interval [-2:+2]

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

The complementary error function is defined as:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 1 - \text{erf}(x) \quad (192)$$

For the numerical computation of the error function with arbitrary precision arithmetic, there are three formulas suited for this job [24]:

$$\text{Formula 1: } \text{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n+1)n!} \quad (193)$$

$$\text{Formula 2: } \text{erf}(x) = \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}, \text{ !! is the double factorial} \quad (194)$$

$$\text{Formula 3: } \text{erfc}(x) = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n-1)!!}{(2x^2)^n}, \text{ !! is the double factorial} \quad (195)$$

Formulas 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 signs in a summation always gives rise to cancellation errors and thereby lacks accuracy if not controlled. In [24], they provide a detailed explanation of each method, along with an error bound for each formula and a practical implementation guide. For all three methods, you don't need to know how many terms you would need in the summation. You can continue until the next term is below the requested precision for x , and then terminate the summation. In [6], they recommend only using formula one for $|x| \leq 1$ and give the following number of terms needed to achieve an accuracy for P decimal digits:

The Math behind arbitrary precision

$$n > 1 + e^{\left(1+W\left(\frac{P \cdot \ln(10)}{e}\right)\right)}, \text{ } W \text{ is the Lambert's } W \text{ function} \quad (196)$$

In [24], they also give an error bound for each formula, and the readers are kindly referred 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.

Regarding formula 3, which also suffers from alternating sign, I found it easier to use the identity: $\text{erfc}(x)=1-\text{erf}(x)$ and rely on just a single solid $\text{erf}()$ implementation. Another deficit of Formula 3 is that the achievable accuracy depends on the magnitude of the number. In [24], they found that the attainable accuracy for P decimal digits was:

$$P \sim e^{21 \cdot (x) \frac{\ln(2)}{\ln(10)}} \quad (197)$$

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. However, in all cases, the method using concurrent series is significantly faster than the straightforward approach.



Figure 6. 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

The Lambert W function, often written as $W(x)$, is defined as the inverse of the function $f(w)=we^w$. In other words, it satisfies.

$$W(x) e^{W(x)}=x$$

This function appears in a wide range of problems where a variable occurs both inside and outside of an exponential, making it indispensable for solving transcendental equations. It is named after Johann Heinrich Lambert, who studied related exponential equations in the 18th century.

The Lambert W function has multiple branches: the **principal branch** $W_0(x)$, which is real for $x \geq -1/e$, and the **lower branch** $W_{-1}(x)$, which takes real values on $[-1/e, 0)$. Because of this branching structure, it is also a useful tool in complex analysis.

Applications of $W(x)$ appear in combinatorics (tree enumeration), physics (quantum statistics, delay differential equations), and engineering (decay processes, population models, circuit

The Math behind arbitrary precision

analysis). Like the error function, it cannot be expressed in terms of elementary functions and is therefore treated as a special function with dedicated numerical methods.

W_0 , as previously mentioned, is called the primary branch, and that is the one we want to compute.

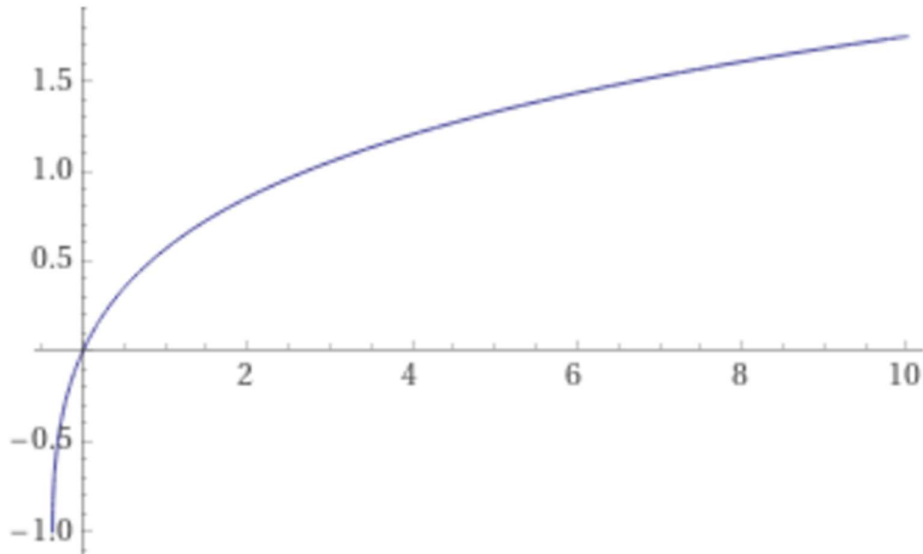


Figure 7 Lambert W function $w_0(x)$ in the interval $]-1/e;10]$

There are three 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 that allows us to easily obtain the first 15-16 digits of accuracy using the built-in double type in C or C++. If you look in the literature, they usually suggest a starting point for the Lambert W function as follows.

The Math behind arbitrary precision

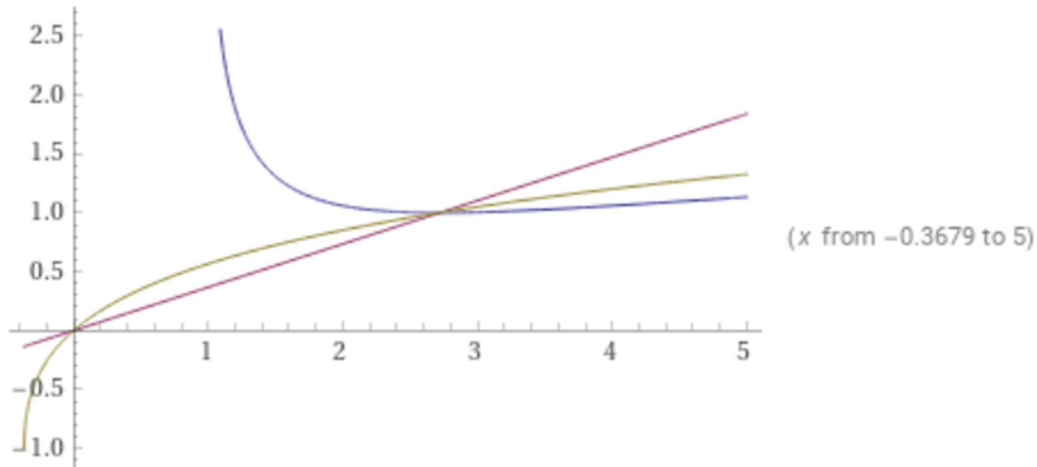


Figure 8: Amber color is $W_0(x)$

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

Although not an impressively precise start point, it usually gives a relative error of less than 10^{-1} as the start point.

Newton's quadratic method

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

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n - x}}{e^{w_n} + w_n e^{w_n}} \quad (198)$$

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, as it will be the most time-consuming part of our computation.

Halley's cubic method

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

$$w_{n+1} = w_n - \frac{w_n \cdot e^{w_n - x}}{e^{w_n(w_{n+1}) + \frac{(w_n+2)(w_n \cdot e^{w_n - x})}{2(w_n+2)}}} \quad (199)$$

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

Boyd's quadratic method

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

$$w_{n+1} = \frac{w_n}{1 + w_n} \left(1 + \ln\left(\frac{x}{w_n}\right) \right) \quad (200)$$

The Math behind arbitrary precision

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 the Lambert W function

Performance-wise, Halley is faster but only up to around 6,000 digits of 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. Therefore, Boyd's iteration will win despite only having a quadratic convergence rate.

However, a technique exists for speeding up this classic iterative method, which has been previously described in this paper, by iterating with full precision for each iterative variable. We instead dynamically adjust them as we iterate to accommodate the target precision for each iteration step. E.g., if we need 1,000 digits of precision of Lambert $W(x)$ we do not need more than 20 digits for the first five iterations (remember that our initial guess was around an accuracy of 10^{-1}). Then we can gradually increase them to our target precision of 1,000 digits over the next six iterations. It would not be very wise to carry these first five and subsequent iterations with a precision of 1,000 digits. Needless to say, this dramatically speeds up the computation.

Performance of Lambert W function

All “dynamic” methods are somewhat similar, but the Boyd methods begin to take off after approximately (faster). With 6,000 digits, it is thereafter the fastest method. See the performance chart below.

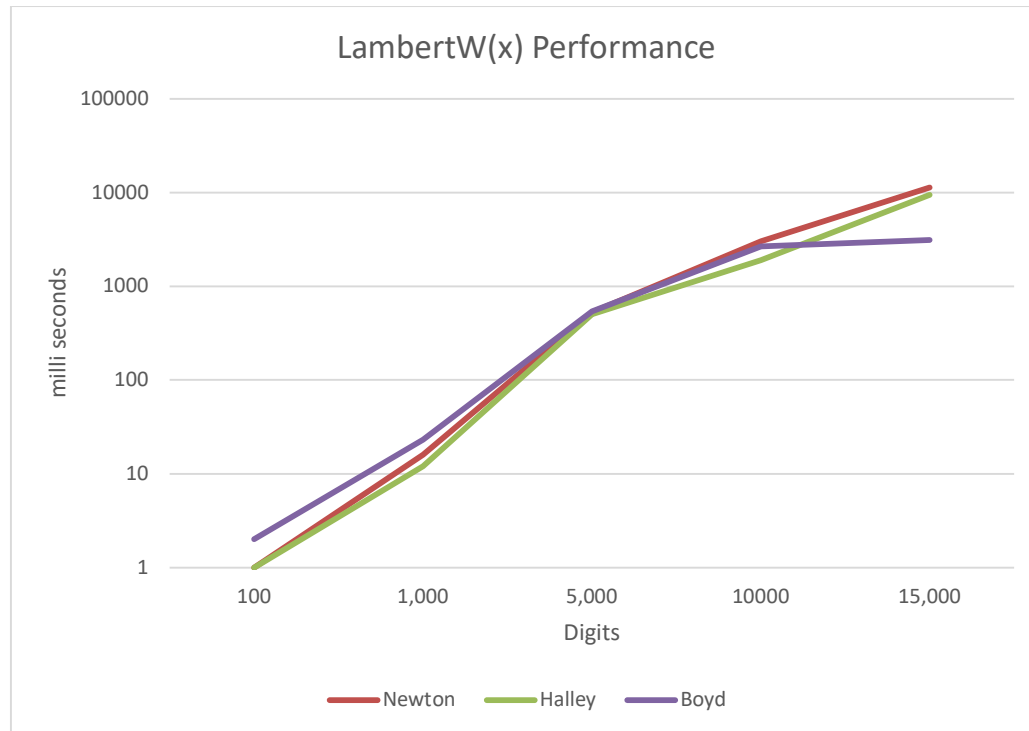


Figure 9. Performance of Lambert $W(x)$ function

The Math behind arbitrary precision

Recommendation for Lambert W function

- The preferred method is Boyd's method.
- Although Halley is close, 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, denoted $\zeta(z)$, is one of the most important functions in analytic number theory and mathematical analysis. It is initially defined for complex arguments, z , with real part greater than one, by the convergent series.

$$\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z} \quad (201)$$

This function can be analytically continued to almost the entire complex plane, except for a simple pole at $z=1$. The zeta function encodes deep information about the distribution of prime numbers through Euler's product formula:

$$\zeta(z) = \prod_{p \text{ prime}} \frac{1}{1-p^{-z}} \quad (202)$$

It also plays a central role in the famous Riemann Hypothesis, which concerns the location of its nontrivial zeros and remains one of the most significant open problems in mathematics.

Beyond number theory, the zeta function appears in physics (statistical mechanics, quantum field theory), probability, and applied mathematics, where zeta-regularization techniques are used to assign values to divergent series.

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

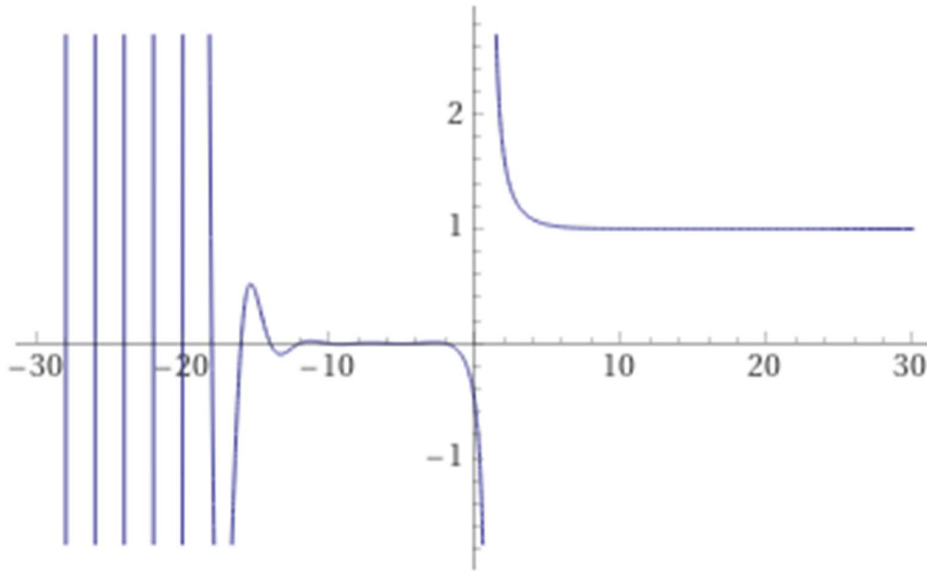


Figure 10 Zeta(x) in the interval [-30:+30]

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

$$\zeta(1 - s) = \frac{2\Gamma(s)}{(2\pi)^s} \cos\left(\frac{\pi \cdot s}{2}\right) \zeta(s) \quad (203)$$

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 due to its simplicity. The formula ([25]) below is valid for any real $s > -(n-1)$.

$$\zeta(s) = \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s} \quad (204)$$

Where e_j is defined as:

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} \right) - 2^n \quad (205)$$

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

$$n = \left\lceil \frac{\ln(10)}{\ln(8)} P \right\rceil \quad (206)$$

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 costly function to call, so we can't expect too high performance when computing the zeta function. However, we get a little bit of a break when s is large, where the use of the actual definition for the zeta function performs faster than the Borwein formula. If the condition

The Math behind arbitrary precision

$$s > 1 + P \frac{\ln(10)}{\ln(P)} \quad (207)$$

Where P is the target precision (in decimal digits), it is met [6]. We can resort to the following series for faster computation.

$$\zeta(s) \approx \sum_{k=0}^N \frac{1}{k^s} \quad (208)$$

And a suitable value for N is:

$$N = \left\lceil 10^{\frac{P}{s-1}} \right\rceil \quad (209)$$

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:

$$\zeta(0) = -\frac{1}{2} \quad (210)$$

$$\zeta(-n) = (-1)^n \frac{B_{n+1}}{n+1} \quad (211)$$

$$\zeta(2n) = (-1)^n \frac{B_{2n}(2\pi)^{2n}}{2(2n)!} \quad (212)$$

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

Optimization of the $\zeta(s)$ series. If we look at the computation for e_j in equation (161).

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} \right) - 2^n \quad (213)$$

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

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \binom{n}{k} \right) - 2^n \quad (214)$$

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

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

And e_j now becomes:

The Math behind arbitrary precision

$$e_j = (-1)^j (sum_j - 2^n) \tag{215}$$

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

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

Algorithm 18 for $\zeta(s)$ where s is any real value not equal to 1

```
if s=0 return -0.5
if s an integer?
  if s negative?
    if s even => return 0
    return compute (70)
  if s even?
    return compute (71)
// s is real or s is an 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)
```

This chapter shows the general formula for the ζ , however, there is a specialized formula and computation for some of the odd values, like the constant $\zeta(3)$. $\zeta(5)$, etc., see later chapters.

Euler-Mascheroni constant γ

The Euler–Mascheroni constant, often denoted by γ , is approximately 0.5772156649. It appears when comparing the harmonic series to the natural logarithm. Specifically, if you take the sum of the harmonic series up to $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$ When subtracting $\ln(n)$, the difference approaches γ as n grows large. This constant also appears in integrals and special functions, making it a recurring element in various branches of analysis.

Leonhard Euler introduced the constant while studying infinite series in the 18th century. Later, the Italian mathematician Lorenzo Mascheroni took an interest in it, leading to deeper investigations and the name we use today. Over time, many mathematicians attempted to understand its algebraic nature. Whether γ is rational or irrational remains unknown, and it continues to spark interest due to its ties to the harmonic series, logarithms, and other core areas of mathematics.

The Euler-Mascheroni constant γ is defined as:

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln(n) \right) \approx 0.577215664 \tag{216}$$

The above equation (1) converges slowly and is useless for arbitrary precision arithmetic. Instead, there are a few other interesting methods to compute this constant.

The Math behind arbitrary precision

- 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 [7].

$$\gamma \approx \frac{S(n)}{V(n)} - \ln(n) \quad (217)$$

$S(n)$ 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:

$$S(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 \cdot H_k \quad (218)$$

$$V(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 \quad (219)$$

The sequence H_n is defined as the partial sum of the Harmonic series:

$$H_n = \sum_{k=1}^n \frac{1}{k} \quad (220)$$

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 of the required precision P to be:

$$n = \left\lceil \frac{P \cdot \ln(10) + \ln(\pi)}{4} \right\rceil \quad (221)$$

The required number of terms k_{max} in the summation as a function of the precision P is:

$$k_{max} \approx 2.07 \cdot P \quad (222)$$

Technically, we don't need the k_{max} a priori since we can terminate the $S(n)$ and $V(n)$ sequence when the individual term value becomes less than the required precision dictated. (Usually, that will require 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:

$$U(n) = S(n) - V(n) \cdot \ln(n) \quad (223)$$

Then

$$\gamma \approx \frac{U(n)}{V(n)} \quad (224)$$

The Math behind arbitrary precision

And

$$U(n) = \sum_{k=0}^{\infty} A_k \tag{225}$$

Where A_k is:

$$A_k = \left(\frac{n^k}{k!}\right)^2 (H_k - \ln(n)) \tag{226}$$

Furthermore, we use B_k as the n^{th} term of the $V(n)$ series.

$$B_k = \left(\frac{n^k}{k!}\right)^2 \tag{227}$$

We can then compute the A_k and B_k simultaneously using the following recurrence.

Algorithm for the 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 19

The binary splitting method for γ

Lastly, you can use the Binary splitting technique as outlined in [26]. The method is derived from the following series (Eq. 217), which leads to the recursive procedure we implement below.

Algorithm: Binary splitting method for γ (7 variables)

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

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

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

$$R(a,b)=R(a,m)S(m,b)+T(a,m)R(m,b)$$

$$S(a,b)=S(a,m)S(m,b)$$

$$T(a,b)=T(a,m)T(m,b)$$

$$U(a,b)=U(a,m)V(m,b)+P(a,m)T(a,m)Q(m,b)R(m,b)+Q(a,m)T(a,m)U(m,b)$$

$$V(a,b)=V(a,m)V(m,b)$$

$$\text{And } P(b-1,b)=1; Q(b-1,b)=b; R(b-1,b)=n^2; S(b-1,b)=b^2; T(b-1,b)=n^2;$$

$$U(b-1,b)=n^2; V(b-1,b)=b^3;$$

Algorithm 20. The 7-variable Binary Splitting for the Euler-Mascheroni constant

You continue this recursive breakdown until $a+1=b$, and you set:

$$P(a,b)=1 \quad Q(a,b)=b \quad R(a,b)=n^2 \quad S(a,b)=b^2 \quad T(a,b)=n^2 \quad U(a,b)=n^2 \quad V(a,b)=b^3$$

And let the formula reverse bottom up.

The Math behind arbitrary precision

In the end, you find γ by:

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+S(0,i))} - \ln(n) \quad (228)$$

In [26], they found that $i=3.5911214766686221366n$ is the number of needed terms as a function of n . And n can be chosen as in [38].

Now, the binary splitting algorithm requires seven variables. You can quickly 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 below. This reduction cuts computational overhead and simplifies the implementation by minimizing data dependencies.

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)=n2; T(b-1,b)=n2; U(b-1,b)=n2;

```

Algorithm 21. The 5-variable Binary Splitting for the Euler-Mascheroni constant

In the end, you find γ by (now that $S(0,i)$ has been replaced by $Q(0,i)^2$):

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+Q(0,i)^2)} - \ln(n) \quad (229)$$

The five-variable version does perform better, but not impressively so. In [26], they have further reduced it to 4 variables.

The reduction to a 4-variable version can be done by eliminating the $T(a,b)$, noting that $T(a,b)=(n^2)^{b-a}$, and substituting it into the four other functions, Q, P, R, U .

Algorithm: Binary splitting method for γ (4 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+(n2)m-a R(m,b)
U(a,b)=U(a,m)Q(m,b)3+P(a,m)(n2)m-a Q(m,b)R(m,b)+Q(a,m)(n2)m-a U(m,b)

And P(b-1,b)=1; Q(b-1,b)=b; R(b-1,b)=n2; U(b-1,b)=n2;

```

Algorithm 22. The 4-variable Binary Splitting for the Euler-Mascheroni constant

The method is faster than the five-variable version, which requires only four variables. We use memorization to calculate each $(n^2)^{m-a}$ once. It is interesting to find out how many different $(n^2)^{m-a}$ you must evaluate between a and b . It grows slowly in proportion to the size of b . E.g., if you plan to perform 9,999 splits, you need to assess only 25 different $(n^2)^{m-a}$. For

The Math behind arbitrary precision

999,999 splits, you need 38 unique values. This makes using memorization highly attractive for implementation.

Performance of the Euler-Mascheroni constant

The performance chart clearly shows that the binary splitting method is superior for computing the Euler-Mascheroni constant. The traditional Brent-McMillan (Euler) and Brent Summation trick (Euler Brent) methods can't be recommended for fast computation. However, the Brent Summarization trick is a significant improvement over the standard Brent-McMillan formula.

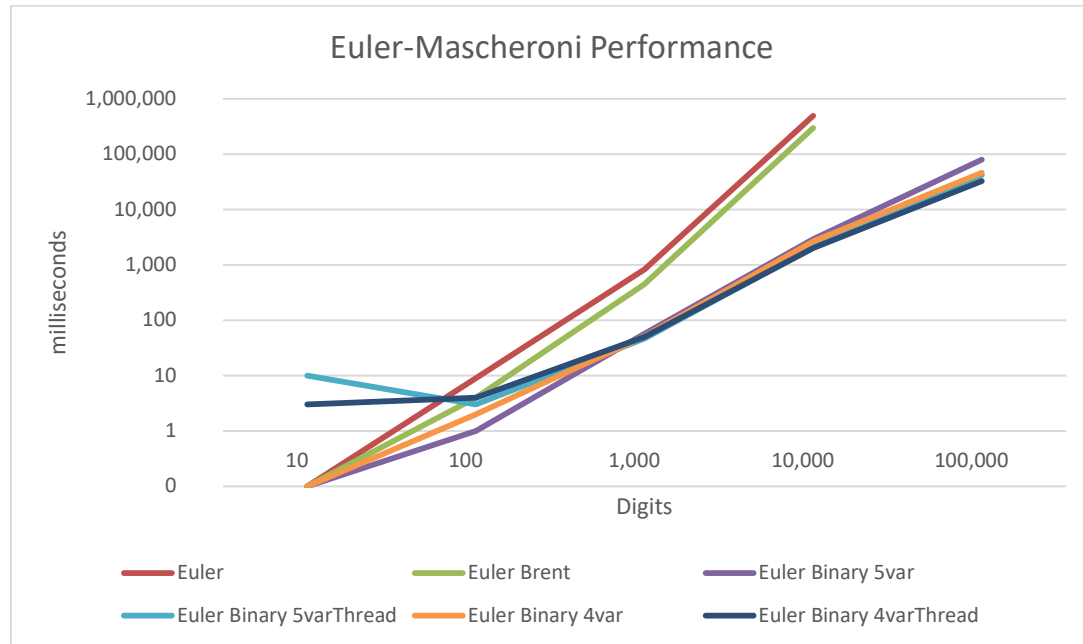


Figure 11. Euler-Mascheroni Performance

The Binary Splitting method, as recommended in [26], is superior to the other methods, and the reduced four variables (Euler Binary 4var) are faster than the five-variable method (Euler Binary 5var). When applying a simple 2-way threading, the performance increases again.

See the table of actual performance, which is 10 to 100,000 digits.

Euler-Mascheroni Performance Result. All times are in milliseconds.					
Digits	10	100	1,000	10,000	100,000
Euler	-	7	830	490,784	
Euler Brent	-	4	451	298,059	
Euler Binary 5var	-	2	57	2,928	79,085
Euler Binary 5varThread	10	4	47	2,091	41,825
Euler Binary 4var	-	2	51	2,648	46,082
Euler Binary 4varThread	3	3	50	2,024	32,775

Table 4. Euler-Mascheroni Performance results.

Table 1 confirms that binary splitting outperforms traditional Brent-McMillan approaches, especially for higher digit counts.

The Math behind arbitrary precision

Recommendation for the Euler-Mascheroni constant

As shown in [26], the four-variable version enhances speed by reducing memory operations. Based on the performance chart above, I recommend the Binary splitting method (with four variables). For digits above 1,000, the threaded binary splitting version outperforms the non-threaded version.

Catalan's constant G

The Catalan constant G is defined as:

$$G = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2} \quad (230)$$

The Catalan constant is $\sim 0.9159655941772\dots$

This series also converges slowly. However, there are several alternative methods to consider.

- Ramanujan method I
- Ramanujan method II
- Broadhurst series
- Binary splitting method (ref. [26])
 - Lucas(2000)
 - Guillera (2008)
 - Guillera (2019)
 - Pilehrood (2010)
 - Zuniga (2023)

Ramanujan's method I

This is one of the many Ramanujan series for fast calculation of the Catalan constant.

$$G = \frac{\pi}{8} \ln(2 + \sqrt{3}) + \frac{3}{8} \sum_{n=0}^{\infty} \frac{(n!)^2}{(2n)!(2n+1)^2} \quad (231)$$

To achieve P , decimal precision, we need to take $P \frac{\ln(10)}{\ln(4)}$. We can use Horner's schema to summarize the terms of the series efficiently. 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:

$$1 + \frac{1}{2 \cdot 3} \left(\frac{1}{3} + \frac{2}{2 \cdot 5} \left(\frac{1}{5} + \frac{3}{2 \cdot 7} \left(\frac{1}{7} + \dots \right) \right) \right) \quad (232)$$

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 23. Horner's evaluation of the fractional sum

Ramanujan's method II

Another of Ramanujan's methods is based on this formula:

The Math behind arbitrary precision

$$G = \sum_{k=0}^{\infty} \frac{(k!)^2}{(2k+1)!} \cdot 2^{k-1} \sum_{j=0}^k \frac{1}{2^{j+1}} \quad (233)$$

In [6], Free uses this formula and the 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($k=1; k \leq n; ++k$)

$tmp=k/(2k+1)$

$B_k=B_{k-1}*tmp$

$C_k=C_{k-1}*tmp+B_k/(2k+1)$

$G_k=G_{k-1}+C_k$

Algorithm 24. Ramanujan II method.

We need a little bit more to achieve P , decimal precision, compared to the first method. In [6], they find that we need to take $P \frac{\ln(10)}{\ln(2)}$ Terms to reach the desired precision.

Broadhurst series

The Broadhurst series has a faster convergence rate than Ramanujan's series at the expense of higher complexity.

$$G = 3 \sum_{k=0}^{\infty} \frac{1}{16^k} \sum_{i=0}^7 \frac{a_i}{(8k+i)^2} - 2 \sum_{k=0}^{\infty} \frac{1}{4096^k} \sum_{i=0}^7 \frac{b_i}{(8k+i)^2} \quad (234)$$

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 [26], 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.

The Binary Splitting method for the Catalan constant.

Several methods are to be considered for the Binary splitting method for the Catalan Constant (ref. [12] & [26]).

- Lucas(2000)
- Guillera (2008)
- Guillera (2019)
- Pilehrood (2010)
- Zuniga (2023)

Lucas (2000) examined how binary splitting could be combined with particular series expansions to handle certain special functions. This work highlighted how breaking large

The Math behind arbitrary precision

sums into smaller intervals and carefully managing partial products can significantly reduce the necessary computational steps. By systematically merging intermediate outcomes, Lucas provided a framework that reduced round-off errors and the time required to attain a specified number of correct digits.

Guillera (2008) built on these ideas by unveiling a new series that benefited from the efficiency of binary splitting. Guillera's approach often produced rapid convergence for constants of interest, and it demonstrated how the method could exploit algebraic factorizations that may not be obvious at first glance. In a later publication, Guillera (2019) refined these techniques by introducing a transformed series that further simplified intermediate products. This made the method more practical for modern high-precision arithmetic libraries and shed light on creative ways to manipulate series for faster convergence.

Pilehrood (2010) studied binary splitting from the perspective of double series and convolution structures. This angle expanded possibilities for summing constant expansions involving multiple summation indices. By focusing on how these numerous indices interact, the Pilehrood work revealed efficient factorization strategies, keeping the core principle of partial-product pairing intact while broadening the range of constants that can be computed. Zuniga (2023) extended the method through alternative representations well-suited to parallel computation, which is valuable in today's multiprocessor environments. The focus here was on creating formulas that minimize dependency between partial sums, allowing computation to be distributed among multiple processing units. This can deliver even faster results for large-scale, high-precision calculations while remaining faithful to the core principles that make binary splitting appealing.

Lupas' Binary Splitting method

Lupas' series for the Catalan constant is:

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{(-1)^{k-1} 2^{8k} (40^2 - 24k + 1) (2k)!^3 k!^2}{k^3 (2k-1) (4k)!^2} \quad (235)$$

As we have seen many times before, we can transform this series into a binary Splitting method using the following 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 25. Lupas' 2000 binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$, and let the formula reverse bottom up.

In the end, you find G by:

The Math behind arbitrary precision

$$G = \frac{P(0,n)+19Q(0,n)}{18 (0,n)} + O(4^{-n}) \quad (236)$$

For n terms, the error is $O(4^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(4)} \right\rceil \quad (237)$$

In [26], they found the linearly convergent cost to be ~ 11.5 , which is not as fast as the Guillera or Pilehrood methods.

Guillera's Binary Splitting method

Guillera published two methods in 2008 and 2019. The first method used:

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k+2)}{(2k+1)^3 \binom{2k}{k}^3} \quad (238)$$

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:

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k+2)k!^6}{(2k+1)!^3} \quad (239)$$

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 26. Guillera 2008 binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$, and let the formula reverse bottom up.

In the end, you find G by:

$$G = 1 + \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O(8^{-n}) \quad (240)$$

For n terms, the error is $O(8^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(8)} \right\rceil \quad (241)$$

The Math behind arbitrary precision

In 2019, Guillera published another formula with a higher convergence rate. The formula looks intimidating at first glance:

$$G = -\frac{1}{1024} \sum_{k=1}^{\infty} \frac{(-4096)^k (45136k^4 - 57183 + 21240k^2 - 3160k + 165)}{k^3 (2k-1)^3} \left(\frac{(2k)!^6 (3k)!^3}{k!^3 (6k)!^3} \right) \quad (242)$$

This method 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)= 45136b4-57184b3+21240b2-3160b+165
Q(b-1,b)=-27(6b-1)3(6b-5)3
R(b-1,b)=512b3(2b-1)3
    
```

Algorithm 27. Guillera 2019 binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$, and let the formula reverse bottom up.

In the end, you find G by:

$$G = -\frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{19683}{64}\right)^{-n}\right) \quad (243)$$

For n terms, the error is $O\left(\left(\frac{19683}{64}\right)^{-n}\right)$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{19683}{64}\right)} \right\rceil \quad (244)$$

Pilehrood binary splitting method

Pilehrood published two formulas in 2010, the short and long formula.

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{256^k (580k^2 - 184k + 15)}{k^3 (2k-1) \binom{6k}{3k} \binom{6k}{4k} \binom{4k}{2k}} \quad (245)$$

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)
    
```

The Math behind arbitrary precision

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 28. Pilehrood 2010 short binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$, and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{729}{4}\right)^{-n}\right) \quad (246)$$

For n terms, the error is $O\left(\left(\frac{729}{4}\right)^{-n}\right)$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{729}{4}\right)} \right\rceil \quad (247)$$

Pilehrood also has a long version formula:

$$G = -\frac{1}{64} \sum_{k=1}^{\infty} \frac{(-256)^k (41984k^6 - 915456k^5 + 782848k^4 - 332800k^3 + 73256k^2 - 7800k + 315)}{k^3(2k-1)(4k-1)^2(4k-3)^2 \binom{8k}{4k}^2 \binom{2k}{k}} \quad (248)$$

This method has a linearly convergent cost of ~ 4.6 , 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 29. Pilehrood 2010 long binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = -\frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O(1024^{-n}) \quad (249)$$

For n terms, the error is $O(1024^{-n})$ and for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil \quad (250)$$

The Math behind arbitrary precision

Zuniga's binary splitting method

A new method was developed in 2023 by Zuniga. [26]

$$G = \frac{1}{768} \sum_{k=1}^{\infty} \frac{(-409)^k (-43203456k^6 + 92809152k^5 - 76613904k^4 + 30494304k^3 - 6004944k^2 + 536620k - 1732)}{k^3(2k-1)(3k-1)(3k-2)(6k-1)(6k-5) \binom{5k}{k} \binom{10k}{5k} \binom{12k}{6k}} \quad (251)$$

Which has a linearly convergent cost of ~3.4, which is slightly higher than the Pilehrood short versions but lower than the Pilehrood long version.

Algorithm: Binary splitting method for Catalan–Zuniga 2023.

```

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)= 43203456b6-92809152b5+76613904b4-30494304b3+6004944b2-
536620b+17325
Q(b-1,b)=5(10b-9)(10b-7)(10b-3)(12b-11)(12b-7)(12b-1)
R(b-1,b)=-128b3(2b-1)(3b-2)(3b-1)(6b-5)(6b-1)
    
```

Algorithm 30. Zuniga's 2023 binary splitting method for the Catalan constant.

You continue this recursive breakdown until $a+1=b$, and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{1}{6} \frac{P(0,n)}{Q(0,n)} + O(12500^{-n}) \quad (252)$$

For n terms, the error is $O(12500^{-n})$ And for a given precision, P, you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(12500)} \right\rceil \quad (253)$$

Comparison of the Catalan Methods

We have outlined several methods for calculating the Catalan constant. To get an overview of the different methods, see the table below, which 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	$O(2^{-4n})$	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	$O\left(\left(\frac{19683}{64}\right)^{-n}\right)$	0.402P

The Math behind arbitrary precision

Pilehrood-short	Binary-Splitting	$O\left(\left(\frac{729}{4}\right)^{-n}\right)$	0.442P
Pilehrood-long	Binary-Splitting	$O(1024^{-n})$	0.332P
Zuniga	Binary-Splitting	$O(12500^{-n})$	0.244P

Table 5. Comparison of the Catalan Methods.

Not surprisingly, performance depends heavily on the convergence speed and implementation type, e.g., a 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 the Guillera 2019 method and 90-100% faster than the Guillera 2008 method. Comparing Pilehrood and Lupas, 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. Among the classical series, the Broadhurst method is by far the fastest.

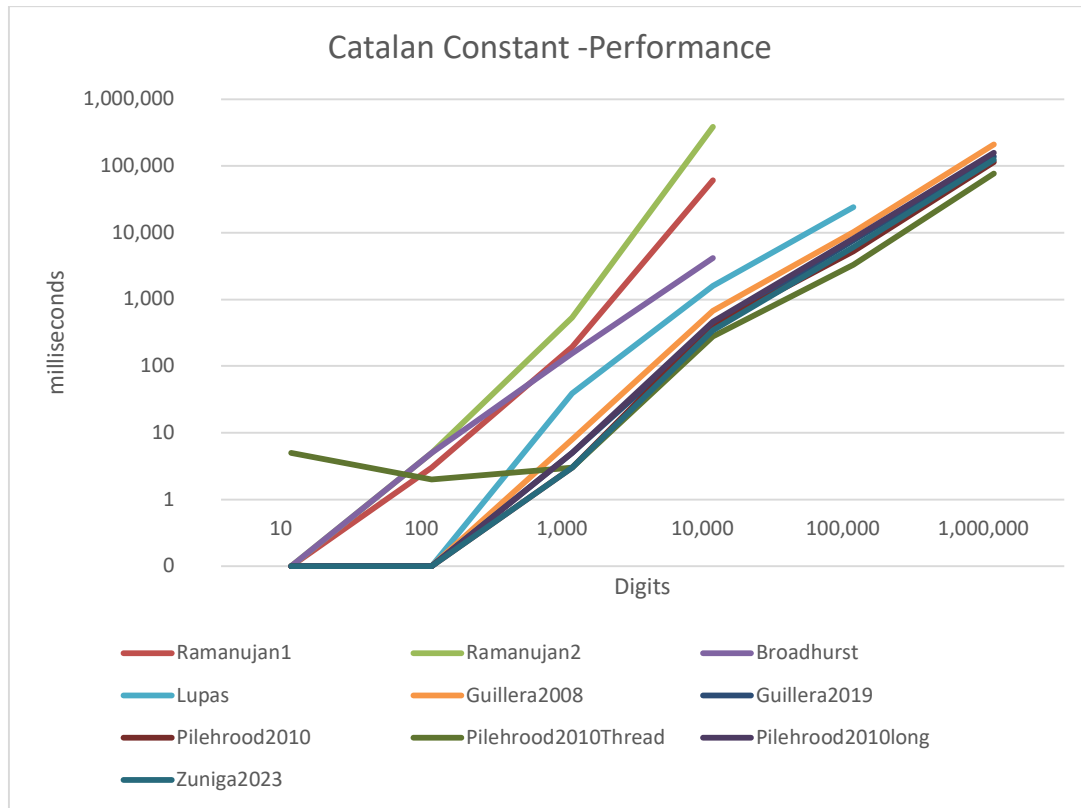


Figure 12. Catalan Constant Performance chart

Digits	10	100	1,000	10,000	100,000	1,000,000
Ramanujan1	0	3	194	61,097		
Ramanujan2	0	5	534	385,528		

The Math behind arbitrary precision

Broadhurst	0	5	155	4,142		
Lupas	0	0	39	1,592	24,183	
Guillera2008	0	0	8	673	10,198	210,692
Guillera2019	0	0	5	388	7,618	138,037
Pilehrood2010	0	0	3	399	5,225	114,603
Pilehrood2010Thread	5	2	3	279	3,314	76,399
Pilehrood2010long	0	0	5	463	8,220	157,580
Zuniga2023	0	0	3	341	6,043	123,388

Table 6. Table of performance of the Catalan Methods.

Notice that even when Zuniga2023 requires fewer splits compared to Pilehrood 2010, it is not faster, primarily due to the more complex variables for P and Q. This results in an increase of approximately double the number of digits compared to the Pilehrood 2010 method's P and Q variables. Notice that the Pilehrood 2010 threaded version is 30-40% faster than the non-threaded version.

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. Use or implement a threaded version of the Pilehrood method if performance is required. Creating a binary splitting method with 2, 3, 4, or more core threads is straightforward. If we only want a classical method, I recommend the Broadhurst method.

Apéry's constant $\zeta(3)$

It is the common short name for the $\zeta(3)$ value. This is a specialized formula for the $\zeta(3)$ instead of using the more general computation of $\zeta(s)$. There has been research into finding a formula, series, etc., for the odd integer values of the zeta function. One of them is the value of $\zeta(3)$. Three methods come to mind, and these are:

- Amdeberhan-Zeilberger series (1997)
- Wedeniwski series (1998)
- And the newer Zuniga (2023)

Amdeberhan and Zeilberger introduced a technique in 1997 that relies on hypergeometric series and symbolic summation. They developed formulas allowing zeta(3) to be expressed in sums that converge at practical rates, using a careful arrangement of terms to reduce numerical error. Their work is notable for combining combinatorial arguments and computer algebra tools, which enable reliable calculations with higher precision.

In 1998, Wedeniwski presented an alternative method that used a series of transformations and computational optimizations. By reorganizing known expansions for zeta(3) and using error bounds to accelerate convergence, Wedeniwski's approach delivered improved efficiency. This was important for applications where high-precision zeta(3) values were needed, such as in certain number-theoretic or physical computations.

Zuniga's contribution in 2023 offered a newer summation framework. The main idea was to derive specialized variants of known series representations for zeta(3) and then apply a combination of convergence accelerators. These refinements further reduced computational complexity and provided another path to reaching reliable decimal approximations with fewer terms, reflecting ongoing progress in the theoretical and practical computing of zeta(3).

Amdeberhan-Zeilberger series

This series was given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} \frac{(-1)^k (205k^2 + 250k + 77)(k!)^{10}}{(2k+1)^5} \quad (254)$$

By now, we should have learned that the most efficient computation is using the binary splitting method. The Amdeberhan-Zeilberger algorithm is laid out below.

Algorithm: Binary splitting method for $\zeta(3)$ (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 31. Amdeberhan-Zeilberger binary splitting algorithm for zeta(3).

The Math behind arbitrary precision

And then

$$\zeta(3) = \frac{P(0,n)+77Q(0,n)}{64Q(0,n)} + O(1024^{-n}) \quad (255)$$

This has a linearly convergent cost of ~ 2.89 , slightly higher than the following Wedeniwski method.

For n terms, the error is $O(1024^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil \quad (256)$$

Wedeniwski series

This series was given by Amdeberhan-Zeilberger in 1997.

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{(-1)^k (126392k^5 + 412708k^4 + 531578k^3 + 336367k^2 + 104000k + 1264)}{(3k+2)!(4k+3)!^3} \quad (257)$$

Algorithm: Wedeniwski Binary splitting method for $\zeta(3)$ (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 32. Wedeniwski's binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{P(0,n)+12463Q(0,n)}{10368 Q(0,n)} + O(110592^{-n}) \quad (258)$$

It has 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:

$$n = \left\lceil P \frac{\ln(10)}{\ln(110592)} \right\rceil \quad (259)$$

Zuniga series (v)

The 2023 Zuniga series is fresh and new; however, it is also a monster series.

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{P(k)}{k^5 (2k-1)(3k-1)(3k-2)(4k-1)(4k-3)(5k-1)(5k-2)(5k-3)(5k-4) \binom{3k}{k} \binom{6k}{3k} \binom{8k}{4k} \binom{9k}{3k} \binom{10}{5k}} \quad (260)$$

The Math behind arbitrary precision

where $P(k)=250765325100000k^{11}-1087318449630000k^{10}+2067749814046250k^9-2269551612681475k^8+1592180015776565k^7-746938801646725k^6+238210943593421k^5-51452348050672k^4+7352050259484k^3-660416507568k^2+33552610560k-731566080$

Algorithm: Zuniga (v) Binary splitting method for $\zeta(3)$ (2023)

$set\ m = \frac{a+b}{2}$ 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:

$P(b-1,b)=250765325100000b^{11}-1087318449630000b^{10}+2067749814046250b^9-2269551612681475b^8+1592180015776565b^7-746938801646725b^6+238210943593421b^5-51452348050672b^4+7352050259484b^3-660416507568b^2+33552610560b-731566080$
 $Q(b-1,b)=288(8b-7)(8b-5)(8b-3)(8b-1)(9b-8)(9b-7)(9b-5)(9b-4)(9b-2)(9b-1)(10b-9)(10b-7)(10b-3)(10b-1)$
 $R(b-1,b)=b^5(2b-1)(3b-2)(3b-1)(4b-3)(4b-1)(5b-4)(5b-3)(5b-2)(5b-1)$

Algorithm 33. Zuniga (v) binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{1}{24} \frac{P(0,n)}{Q(0,n)} + O(34828517376^{-n}) \quad (261)$$

It has a linearly convergent cost of ~ 2.3 , which is lower than the Wedeniwski method.

For n terms, the error is $O(34828517376^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(34828517376)} \right\rceil \quad (262)$$

Zuniga series (vi)

Additionally, in 2023, Zuniga unveiled another method with a computational cost of ~ 2 .

$$\zeta(3) = \frac{1}{48} \sum_{k=0}^{\infty} \frac{-(-1)^k P(k)}{k^5(2k-1)^3(3k-1)(3k-2)(4k-1)(4k-3)(6k-1)(6k-5) \binom{5k}{k} \binom{5k}{2k} \binom{9k}{4k} \binom{10k}{5k} \binom{12k}{6k}} \quad (263)$$

where $P(k)=1565994397644288k^{11}-6719460725627136k^{10}+12632254526031264k^9-13684352515879536k^8+9451223531851808k^7-4348596587040104k^6+1352700034136826k^5-282805786014979k^4+38721705264979k^3-3292502315430k^2+156286859400k-3143448000$

Algorithm: Zuniga (vi) Binary splitting method for $\zeta(3)$ (2023)

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

The Math behind arbitrary precision

And:

$$\begin{aligned}
 P(b-1,b) &= 1565994397644288b^{11} - 6719460725627136b^{10} + 12632254526031264b^9 - \\
 &13684352515879536b^8 + 9451223531851808b^7 - 4348596587040104b^6 \\
 &+ 1352700034136826b^5 - 282805786014979b^4 + 38721705264979b^3 - 3292502315430b^2 \\
 &+ 156286859400b - 3143448000 \\
 Q(b-1,b) &= 270(9b-8)(9b-7)(9b-5)(9b-4)(9b-2)(9b-1)(10b-9)(10b-7)(10b-3)(10b- \\
 &1)(12b-11)(12b-7)(12b-5)(12b-1) \\
 R(b-1,b) &= -b^5(2b-1)^3(3b-2)(3b-1)(4b-3)(4b-1)(6b-5)(6b-1)
 \end{aligned}$$

Algorithm 34. Zuniga (vi) binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{1}{48} \frac{P(0,n)}{Q(0,n)} + O(717445350000^{-n}) \tag{264}$$

It has a linearly convergent cost of ~2.1, which is lower than Zuniga’s version V.

For n terms, the error is $O(717445350000^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(717445350000)} \right\rceil \tag{265}$$

Comparison of Apéry’s Methods

We have outlined quite a few methods for calculating Apéry’s constant. To get an overview of the different methods, see the table below, which outlines the name, implementation type, error, and precision requirement.

Method	Implementation	Error	N(P), <i>P=precision</i>
Amdeberhan-Zeilberger	Binary Splitting	$O(1024^{-n})$	1.661P
Wedeniwski	Binary-Splitting	$O(110592^{-n})$	1.107P
Zuniga (v)	Binary-Splitting	$O(34828517376^{-n})$	0.095P
Zuniga (vi)	Binary-Splitting	$O(717445350000^{-n})$	0.084P

Table 7 Comparison of Apéry’s methods.

Due to its much faster convergence rate, the Zuniga two series requires more than 10 times fewer splits than the Amdeberhan-Zeilberger and Wedeniwski method.

Apéry Constant $\zeta(3)$ performance

Both Binary splitting methods outperform the general zeta function implementation by several magnitudes (not shown in the figure below). It seems that the Wedeniwski method has a slight edge over the Amdeberhan method. 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. Furthermore, none of them can match the performance of the two Zuniga series, even though $P(b-1,b)$, $Q(b-1,b)$, and $R(b-1,b)$ are way more complicated to calculate.

The Math behind arbitrary precision

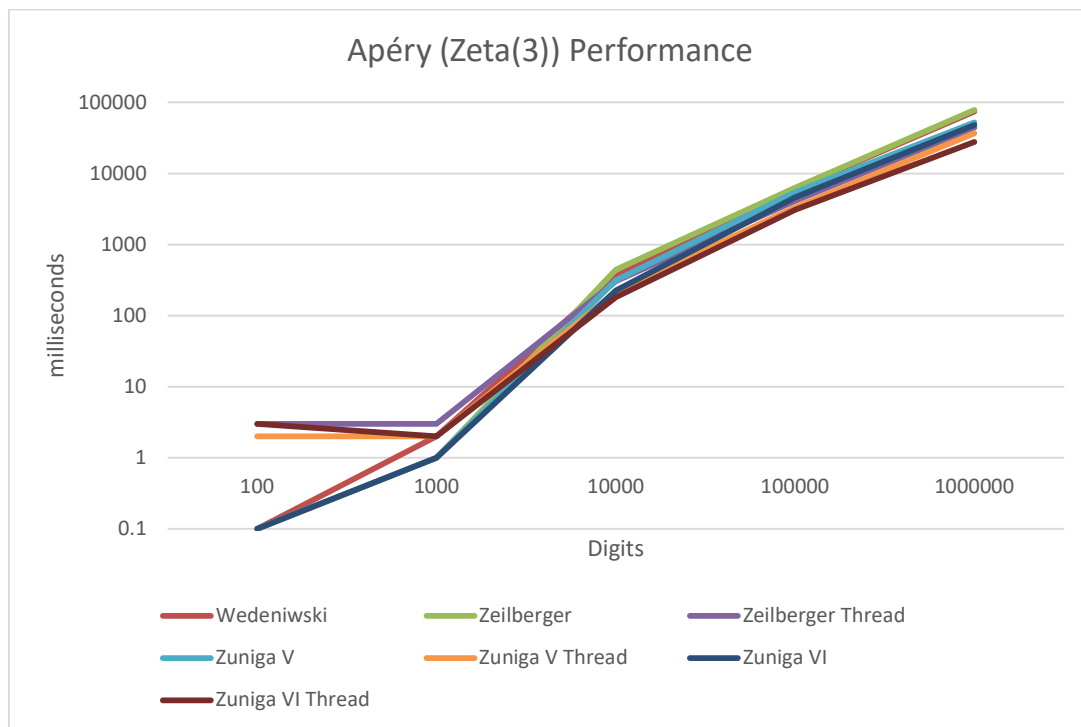


Figure 13 Apéry Constant Performance

It is sometimes clearer to look at the tables below, which show the time required to calculate the Apéry constant from 10 to 1M digits. All times in milliseconds.

Digits	10	100	1,000	10,000	100,000	1,000,000
Wedeniwski	0	0	2	359	6,134	73,926
Zeilberger	0	0	1	447	6,284	78,453
Zeilberger Thread	5	3	3	304	4,098	44,105
Zuniga V	0	0	1	318	5,467	51,975
Zuniga V Thread	0	2	2	217	3,351	36,539
Zuniga VI	0	0	1	228	4,598	47,941
Zuniga VI Thread	0	3	2	181	3,086	27,788

Table 8. Time in milliseconds to compute the Apéry constant from 10 to 1M digits.

It is worth mentioning that simple two-way threading improves performance significantly for all the methods presented.

Recommendation for the constant $\zeta(3)$

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.
- 3) Wedeniwski is slightly faster, but Amdeberhan-Zeilberger is more straightforward to implement.
- 4) However, the fastest method is the Zuniga (vi) version, which performs the fastest in both the non-threaded and threaded versions.

The Math behind arbitrary precision

- 5) Implement the threaded version for the binary splitting method if speed is of the essence.

The Lemniscate Constant ϖ

The lemniscate constant (ϖ) is the ratio of the perimeter of Bernoulli's lemniscate to its diameter. The $\varpi \sim 2.622057554292$. In literature, you sometimes see that 2ϖ or $\varpi/2$ is also referred to as the Lemniscate constant.

There are several definitions of the Lemniscate constant. One of them is:

$$\varpi = 2 \int_0^1 \frac{1}{\sqrt{1-x^4}} dx \quad (266)$$

There are also many available series or continuous fractions to compute the lemniscate constant. As we have seen many times before, the Binary Splitting method can be used to obtain a significantly more efficient computation of the Lemniscate constant. Notably, Zuniga 2023 presents several binary splitting methods that compute 2ϖ as the Lemniscate constant. Other methods have been developed. One of them is the method used by Guillera, which is discussed later.

Zuniga's binary splitting methods

Zuniga published over 10 variations of the formula, two of which are interesting.

Zuniga version vii.

$$\frac{1}{2\varpi} = \frac{214326}{\sqrt[6]{11816941917501}} \sum_{k=1}^{\infty} \left(\frac{-51}{2315685267}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{\left(\frac{1}{36}\right)_k \left(\frac{7}{36}\right)_k \left(\frac{13}{36}\right)_k \left(\frac{19}{36}\right)_k \left(\frac{25}{36}\right)_k \left(\frac{31}{36}\right)_k}{\left(\frac{1}{3}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{2}{3}\right)_k \left(\frac{2}{3}\right)_k k!^2}\right) \quad (267)$$

After some manipulation and simplification, the algorithm below is yielded.

Algorithm: Zuniga (vii) Binary splitting method for ϖ (2023)

```

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)=99446494228488b7-296948949253092b6+339735211540956b5-
185806427026662b4+48479683290426b3-4840729282291b2
Q(b-1,b)=121545688294296b2(3b-1)2(3b-2)2
R(b-1,b)=- (36b-5)(36b-11)(36b-17)(36b-23)(36b-29)(36b-35)
    
```

Algorithm 35. Zuniga 2023 version vii. Notice the complexity in computing P, Q, and R

And then

$$\varpi = \frac{1}{324 \sqrt[6]{453789}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2315685267}\right)^n\right) \quad (268)$$

This method has a linearly convergent cost of ~ 1.566 .

The Math behind arbitrary precision

For n terms, the error is $O\left(\left(\frac{512}{2315685267}\right)^n\right)$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{2315685267}{512}\right)} \right\rceil \quad (269)$$

Zuniga version viii, which is similar.

$$\frac{1}{2\varpi} = \frac{215622}{\sqrt[4]{483153}} \sum_{k=1}^{\infty} \left(\frac{512}{2357947691}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{\left(\frac{1}{36}\right)_k \left(\frac{5}{36}\right)_k \left(\frac{13}{36}\right)_k \left(\frac{17}{36}\right)_k \left(\frac{25}{36}\right)_k \left(\frac{29}{36}\right)_k}{\left(\frac{1}{3}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{2}{3}\right)_k \left(\frac{2}{3}\right)_k k!^2}\right) \quad (270)$$

After some manipulation and simplification, the algorithm below is yielded.

Algorithm: Zuniga (viii) Binary splitting method for ϖ (2023)

```

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)=1768056164733b7-52825631815620b6+60473303319276b5-
33092086224942b4+8638260598818b3-862864755643b2
Q(b-1,b)=123763958405208b2(3b-1)2(3b-2)2
R(b-1,b)=(36b-7)(36b-11)(36b-19)(36b-23)(36b-31)(36b-35)
    
```

Algorithm 36. Zuniga 2023 version viii.

And then

$$\varpi = \frac{1}{1188\sqrt[4]{35937}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2357947691}\right)^n\right) \quad (271)$$

This method has a linearly convergent cost of ~ 1.564 .

For n terms, the error is $O\left(\left(\frac{512}{2357947691}\right)^n\right)$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{2357947691}{512}\right)} \right\rceil \quad (272)$$

Zuniga version x, which is similar.

$$\frac{1}{2\varpi} = 128\sqrt[4]{48315310433320250} \sum_{k=1}^{\infty} \frac{k^2 P(k)}{(16k-3)(16k-7)(16k-11)(16k-15)} \prod_{i=1}^k \frac{(16k-3)(16k-7)(16k-1)(16k-1)}{11008380780544i^2(2i-1)^2} \quad (273)$$

Algorithm: Zuniga (x) Binary splitting method for ϖ (2023)

```

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)
    
```

The Math behind arbitrary precision

And:

$$P(b-1,b)=22934126592b^3-45503382016b^2+28302850848b-5642344589$$

$$Q(b-1,b)=11008380780544b^2(2b-1)^2$$

$$R(b-1,b)=(16b-3)(16b-7)(16b-11)(16b-15)$$

Algorithm 37. Zuniga 2023 version x.

And then

$$\varpi = \frac{1}{128^4 \sqrt[4]{1043320250}} \frac{Q(0,n)}{P(0,n)} + O(671898241^{-n}) \quad (274)$$

This method has a linearly convergent cost of ~ 0.78

For n terms, the error is $O(671898241^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(671898241)} \right\rceil \quad (275)$$

Zuniga version x(2), which is a more straightforward version of it.

$$\frac{1}{2\varpi} = \frac{10304}{\sqrt[4]{6440}} \sum_{k=1}^{\infty} \frac{k^2(8640n-8365)}{(8k-3)(8k-7)} \prod_{i=1}^k \frac{(8k-3)(8k-7)}{1658944i^2} \quad (276)$$

Algorithm: Zuniga (x(2)) Binary splitting method for ϖ (2023)

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:

$$P(b-1,b)=b^2(8640b-8365)$$

$$Q(b-1,b)=1658944b^2$$

$$R(b-1,b)=(16b-3)(16b-7)$$

Algorithm 38. Zuniga version x(2).

And then

$$\varpi = \frac{\sqrt[4]{6440}}{20608} \frac{Q(0,n)}{P(0,n)} + O((25491)^{-n}) \quad (277)$$

These methods have a linearly convergent cost of ~ 0.78

For n terms, the error is $O(25491^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(25491)} \right\rceil \quad (278)$$

Guillera's Binary Splitting method.

Guillera's series in the form proper for the Binary splitting method is:

$$\frac{1}{2\varpi} = \frac{34560}{\sqrt[8]{162000}} \sum_{k=1}^n \frac{k^2(1288k-1247)}{-(8k-5)(8k-7)} \prod_{i=1}^k \frac{-(8i-5)(8i-7)}{1658880i^2} \quad (279)$$

The Math behind arbitrary precision

Not as voluminous as some of the previous Zuniga series.

Algorithm: Guillera Binary splitting method for ϖ (2023)

```

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)=b2(1288b-1247)
Q(b-1,b)=1658880b2
R(b-1,b)=- (8b-5)(8b-7)
    
```

Algorithm 39. Guillera 2023 version.

And then

$$\varpi = \frac{\sqrt[8]{162000} Q(0,n)}{69120 P(0,n)} + O(25920^{-n}) \quad (280)$$

These methods have a linearly convergent cost of ~ 0.78 , which is good and lower than the Zuniga series (except version x and x(2)). However, they require more splits than the Zuniga series. However, a lower linearly convergent cost looks promising.

For n terms, the error is $O(25920^{-n})$ And for a given precision, P , you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(25920)} \right\rceil \quad (281)$$

Comparison of the Lemniscate methods.

The five Lemniscate methods are listed below. The Zuniga versions (vii and viii) have similar characteristics and are more efficient than the Guillera method.

Method	Implementation	Error	N(P), P=precision
Zuniga vii	Binary Splitting	$O\left(\left(\frac{512}{2315685267}\right)^n\right)$	0.15P
Zuniga viii	Binary-Splitting	$O\left(\left(\frac{512}{2357947691}\right)^n\right)$	0.15P
Zuniga x	Binary-Splitting	$O(671898241^{-n})$	0.11P
Zuniga x(2)	Binary-Splitting	$O(25491^{-n})$	0.23P
Guillera	Binary-Splitting	$O(25920^{-n})$	0.23P

Table 9. Comparison of key characteristics of the three methods.

The Zuniga methods (vii, viii, and x) require fewer splits to achieve a given precision of the result. However, each split is more time-consuming than the Guillera version.

The Math behind arbitrary precision

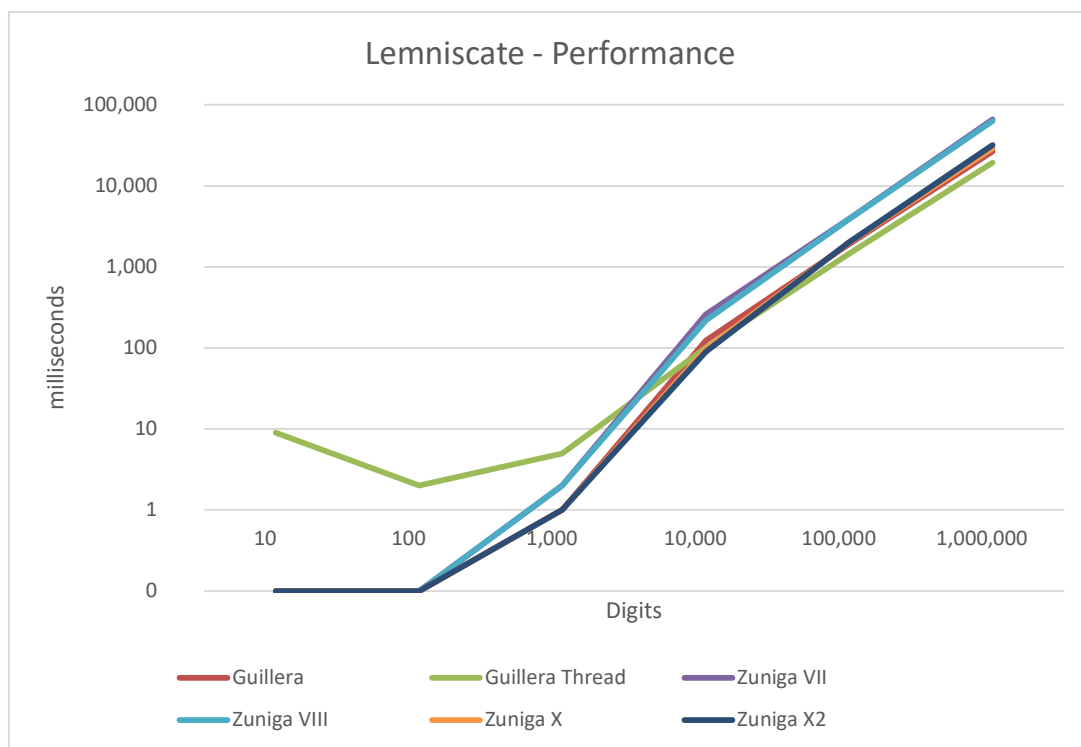


Figure 14. Time to compute the Lemniscate constant up to 1M digits. Notice that Guillermo is considerably faster than the two Zuniga methods. A threaded version of the Guillermo method is advantageous over 10,000-digit precision.

Lemniscate constant. Time in milliseconds							
Digits	10	100	1,000	10,000	100,000	1,000,000	
Guillera	0	0	1	127	2,023	27,658	
Guillera Thread	7	3	3	101	1,671	20,015	
Zuniga VII	0	0	3	266	4,210	64,671	
Zuniga VIII	0	0	3	237	4,127	72,702	
Zuniga X	0	0	1	97	1,975	29,160	
Zuniga X(2)	0	0	1	90	2,013	31,823	

Table 10. Performance of the Lemniscate constant. As you can see, the Guillera method is more than twice as fast as the Zuniga versions, and a threaded version of the Guillera method is approximately 30% faster in line with expectations.

Recommendation for the Lemniscate constant ω

I recommend the following:

1. It is clear that if you are serious, you would implement one of the binary splitting methods.
2. The Guillera method is faster than the Zuniga four versions and more straightforward to implement. Therefore, it is recommended.
3. Implement the threaded version for the binary splitting method if speed is of the essence (a 30% increase in performance above 10,000 digits).

The Math behind arbitrary precision

Appendix

A summary table of the preferred method for arbitrary precision arithmetic.

	Preferred method
Integer arithmetic	
Addition	Schoolbook addition
Subtraction	Schoolbook subtraction
Multiplication	Linear convolution for a smaller number, otherwise FFT
Division	Use Knuth's D algorithm for division
Remainder	Use Knuth's D algorithm for remainder
Floating point arithmetic	
Addition	Schoolbook addition
Subtraction	Schoolbook 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, the Taylor series for log(x) with argument reduction and coefficient scaling is used. 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 $\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$
Arccosh(x)	Use $\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$

The Math behind arbitrary precision

Arctanh(x)	Use $\text{Arctanh}(x) = \frac{1}{2} \ln \left(\frac{1+x}{1-x} \right)$
Special functions	
Gamma function	Integration by parts
Beta function	$B(z, w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$
Error Function	$\text{erf}(x)$ $= \frac{2xe^{-x^2}}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(2x^2)^n}{(2n+1)!!}$, <i>!! is the double factorial</i>
Lamber W function	Boyd's iteration: $w_{n+1} = \frac{w_n}{1+w_n} \left(1 + \ln \left(\frac{x}{w_n} \right) \right)$
Zeta function	An optimized version of P. Borwein's algorithm 3: $\zeta(s) = \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s}$ Where: $e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} \right) - 2^n$
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	The four-variable Binary Splitting Method
Catalan	Pilehrood 2010 short version Binary Splitting Method
Apéry (zeta(3))	Zuniga (vi) Binary Splitting Method
Lemniscate ϖ	Guillera 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 nth root 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

The Math behind arbitrary precision

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
27. J. Schulz, A verified functional implementation of the Schönhage-Strassen-Algorithm, Department of Mathematics, TUM School of Computation, Information Technology, Technical University of Munich.
28. [Fast Fourier transform - Wikipedia](#)
29. M. Fürer. [Fast Integer multiplication - Webarchive](#)
30. A completely arbitrary precision library in C++. Includes arbitrary integer, floating point, fractions, and interval arithmetic. [Arbitrary Precision C++ Packages](#)
31. [Karatsuba algorithm - Wikipedia](#)
32. [Toom–Cook multiplication - Wikipedia](#)
33. [Schönhage–Strassen algorithm – Wikipedia](#)
34. Ronald W. Potter. Arbitrary Precision Calculation of Selected Higher Functions. ISBN #: 978-1-312-59943-7
35. Fast Computation of Math Constants in Arbitrary Precision. [Fast Computation of Math Constants in arbitrary precision.docx](#)
36. Alexander Yee, Binary Splitting Recursion Library. [Binary Splitting Recursion Library](#)
37. Github.com [y-cruncher-Formulas/Official Formulas at master · Mysticial/y-cruncher-Formulas · GitHub](#)
38. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
39. Exploring the Binary Splitting method. [HVE Exploring Binary Splitting Method.](#)
40. Fast Arbitrary integer multiplication. [HVE Fast arbitrary precision integer multiplication](#)