Fast Logarithm function for Arbitrary Precision number.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper is a follow-up to a previous paper that describes the math behind arbitrary precision numbers [7]. First, the original paper was written in 2013, and many things have happened since then. Secondly, I have come across some other interesting methods for calculating the logarithm function. The paper provides a detailed description of how to perform log(x) (logarithm of x) calculations with arbitrary precision, outlining traditional methods while introducing an improved version 10-20 times faster than the original method used in the author's arbitrary-precision math packages. Furthermore, a subchapter on faster methods for log constants log(2), log(3), log(5), and log(10) has been added.

Introduction:

Usually, when implementing arbitrary precision math packages, you would use the standard Taylor series calculation for calculating ln(x) ($\log_e(x)$) for arbitrary precisions. The Taylor series for ln(x) is not particularly fast in its raw form. However, you can apply techniques that significantly improve the method's performance. We will discuss various methods for calculating ln(x) and elaborate on techniques such as clever argument reduction and coefficient scaling to improve the method's performance. Furthermore, we will analyze the Newton and Halley methods for calculating ln(x) and then review the AGM method, which is by far the fastest.

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

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

1. Fast arbitrary precision integer multiplication. HVE Fast arbitrary precision integer multiplication
2. Fast Computation of Math Constants in arbitrary precision. HVE Fast Computation of Math Constants in arbitrary precision
3. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
4. Fast Square Root and inverse calculation in arbitrary precision math. HVE Fast Square Root & Nth-root Computation for arbitrary precision
5. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
6. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision

7. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
8. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
9. Fast trigonometric functions for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
10. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
11. Fast conversion from an arbitrary-precision number to a string. HVE Fast conversion from arbitrary precision to string
12. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision
13. Fast Computation of Stirling's numbers in arbitrary precision. HVE Fast Computation of Stirling numbers in arbitrary precision
14. Fast Prime computation in arbitrary precision. HVE Fast Prime Computation in arbitrary precision
15. Fast PRNG in arbitrary precision. HVE Fast PRNG in arbitrary precision
16. Fast Fibonacci sequence in arbitrary precision. HVE Fast Fibonacci in arbitrary precision
17. Fast Division and Inverse Computation in arbitrary precision math. HVE Fast Division and Inverse Computation in arbitrary precision

Change Log

26 September 2026. Added Chebyshev method and outlined an improved iteration scheme for Newton, Halley, and Chebyshev. We also added a chapter on the Brent division-free method for computing ln(x).
6 April 2025. Added a new section on using the binary splitting method to compute log constants like log(2), log(3), log(5), and log(10).
27 February 2023. Cleaning up the document and correcting minor issues.

Acknowledgment. The author drafted, edited, and improved parts of the text in this revision with the assistance of Claude (Anthropic), an AI language model. The algorithms, the C++ implementation, and all performance measurements are the author's own, and all text, derivations, and results have been reviewed and verified by the author.

Contents

Abstract: 1
Introduction: 1
Change Log 1
The Arbitrary precision library 5
Internal format for float_precision variables 6
Normalized numbers 6
Log(x) 8
Log(x) using the Taylor series. 8
Example 1 9
Argument Reduction 10
Example 2: 10
The issue with arbitrary precision 11
Finding a reasonable reduction factor. 12
Guard Digits 13
Source log(x) using Taylor series 13
Further Improvement of the methods? 14
Log(x) using 2nd- and 3rd-order methods. 16
The precision ladder (iterative deepening) 16
Source of the ladder helpers 18
Source of the small_enough helper 18
Newton Method for log(x) 18
Source ln_newton() 19
Log(x) using the division-free Newton method (Brent). 20
The propagated residual is not self-correcting 20
Source ln_newton_divfree() 21
Implemented correctly, the division-free method is still slower than ordinary Newton, Halley, and Chebyshev at every precision. 24
Log(x) using the Chebyshev method. 25
Source ln_chebyshev() 26
Log(x) using the Halley method. 27
Source ln_halley() 27
Log(x) using the AGM method 28
AGM Algorithm 29
Source AGM() 29
Source ln_AGM() 30
Log(x) performance 32
Log(x) using the AGM method and multiple threads. 33
Threaded lnAGM() source 33
Recommendation for calculating log(x) 34
The Constant Log(2) 36
The spigot-like method using built-in integer arithmetic (64-bit) 36
Source spigot_lnxy_64() 36
Xiao binary splitting for log(2) 39
Source Xiao binary splitting method 40
Zuniga's binary splitting method for log(2) 41

Source Zuniga-II binary splitting method. 41
Comparison of the Log(2) methods 42
Performance for log(2) 43
Recommendation for the log(2) constant 44
The constant Log(3) 44
Zuniga's binary splitting method for log(3) 45
Source Zuniga I 45
Source Zuniga IG4 47
Performance of log(3) 49
Recommendation for the log(3) constant 50
The constant Log(5) 50
Zuniga binary splitting method for log(5) 50
Source Zuniga 51
Performance of log(5) 53
Recommendation for log(5) 53
The constant Log(10) 54
Performance of log(10) 54
Recommendation for log(10) 55
Conclusion for log() constants 56
Reference 57
Appendix 58
Measured performance 59

The Arbitrary precision library

You can skip this section if you are already familiar with the arbitrary precision library.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Internal format for float_precision variables

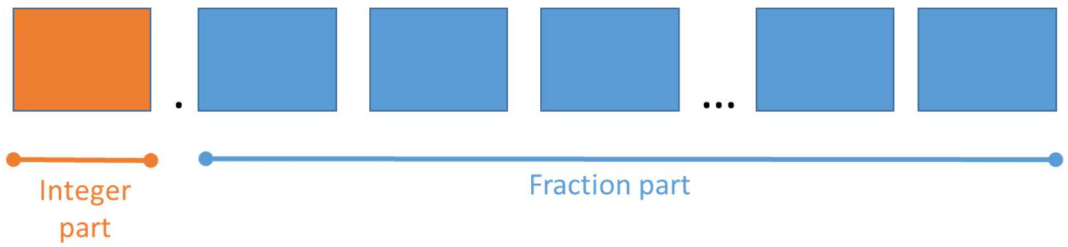
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

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

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

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

The Binary format mBinary



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

Other internal class variables like the sign, exponent, precision, and rounding mode are not important for understanding the code segments.

Normalized numbers

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

For more details, see [1].

Log(x)

You can calculate log(x) in arbitrary precision in many ways. Traditional Taylor series expansion has been used. However, another method involving AGM (Arithmetic-Geometric Mean) has been shown to be efficient for calculating log(x). This chapter will examine the following:

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

We focus on these six methods because they represent a progression from simpler to more sophisticated techniques. The Taylor series approach is conceptually straightforward but can become slow at very high precision. Newton's, Chebyshev and Halley's methods leverage iterative root-finding techniques, offering faster convergence in many scenarios. Finally, the AGM algorithm is known for its exceptional asymptotic performance, making it the method of choice beyond certain precision thresholds.

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

Because most advanced mathematical software treats ln(x) as the base function, we can easily obtain logarithms on other bases using simple scaling factors. For instance, $\log_{10}(x)=\ln(x) / \ln(10)$. This unifies our discussion under a single standard function while allowing conversion to other bases as needed.

Log(x) using the Taylor series.

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

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

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. However, the series converges slowly to ln(x) and is unsuitable for arbitrary precision. Instead, most implementations use the inverse hyperbolic tangent function:

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

Which is valid for any real number x>0.

This series converges with reasonable speed if x is small.

Example 1

Using x=2, we get after 15 Taylor series the result of ln(2)= 0.693147180559945 computing $z = \frac{x-1}{x+1}$.

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**2**	
Taylor reductions=		**0**		
Terms	**z_n**	**Term Sum**	**Ln(x)**	**Error**
1	3.3333E-01	0.333333333333333	0.666666666666667	2.65E-02
2	3.7037E-02	0.345679012345679	0.691358024691358	1.79E-03
3	4.1152E-03	0.346502057613169	0.693004115226337	1.43E-04
4	4.5725E-04	0.346567378666144	0.693134757332288	1.24E-05
5	5.0805E-05	0.346573023695414	0.693146047390827	1.13E-06
6	5.6450E-06	0.346573536879893	0.693147073759785	1.07E-07
7	6.2723E-07	0.346573585128006	0.693147170256012	1.03E-08
8	6.9692E-08	0.346573589774121	0.693147179548241	1.01E-09
9	7.7435E-09	0.346573590229622	0.693147180459244	1.01E-10
10	8.6039E-10	0.346573590274906	0.693147180549812	1.01E-11
11	9.5599E-11	0.346573590279458	0.693147180558916	1.03E-12
12	1.0622E-11	0.346573590279920	0.693147180559840	1.05E-13
13	1.1802E-12	0.346573590279967	0.693147180559934	1.10E-14
14	1.3114E-13	0.346573590279972	0.693147180559944	1.33E-15
15	1.4571E-14	0.346573590279972	0.693147180559945	0.00E+00

After 15 terms, we get the final result. That is not too bad. However, if we change the argument to 10, we need 75 Taylor's terms to get the result, and if we use x=0.1, then we also need 75 Taylor terms. With x=1.1, you only need six Taylor Terms.
This 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.

A direct application of the Taylor series can require tens of thousands (or even millions) of terms for large or very small x. To avoid that explosion in complexity, we rely on argument reduction, a technique that shrinks x near one so that the series converges more rapidly. Below, we demonstrate how repeated extraction of square roots achieves this goal.

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 do this using a technique called argument reduction, which lets us work with a smaller number to converge faster to ln(x) using fewer *terms* of the Taylor series.

We can use the identity:

$$\ln(x) = \ln\left(\left(\sqrt{x}\right)^2\right) = 2 \cdot \ln(\sqrt{x}) \qquad (3)$$

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

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

Example 2:

If we used the previous example 1 and reduced the argument twice from two to 1.1892…, we need only 7 Taylor terms to get the same result as before, saving eight Taylor terms but adding two squarings and a multiplication by 22 = 4 at the end. Overall, we save a lot because we avoided eight time-consuming divisions in Taylor's terms.

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**1.189207115**	
Taylor reductions=		**2**		
Terms	**zn**	**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

We get the same results after four Taylor terms if we use an eight-times reduction.

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**1.002711275**	
Taylor reductions=		**8**		
Terms	**zn**	**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

17 Taylor's terms to reach a result do not seem so bad at first glance. 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.

Now, it would be convenient if we could estimate the needed number of Taylor terms for a given argument to optimize the use of argument reduction. This underscores how drastically the required terms can grow as precision increases. Luckily, we can make the process more systematic by estimating exactly how many terms we need for a given precision, ensuring we don't waste time computing expansions beyond what's necessary. The n^{th} -Taylor term for ln(x) is given by:

$$2 \cdot \frac{z^{2n-1}}{2n-1}, where\ z = \frac{x-1}{x+1} \quad (4)$$

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

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

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

$$n = \frac{1}{2}\left(\frac{-P \cdot \ln(10)}{\ln(z)} + 1\right) \quad (5)$$

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

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

Table 1. Taylor terms needed for log(2) as a function of precisions.

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

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

The number of Taylor Terms is needed as a function of the precision in decimal digits.

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

Table 2. The number of Taylor terms needed as a function of reduction factor applied.

Finding a reasonable reduction factor.

As the above table shows, a higher reduction factor significantly improved the performance. However, how many times of reduction is adequate? At least x should be reduced to some arbitrary number close to one. I use 1.001 as the target in [1]

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

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

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

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

A reasonable number for the limit is 1.001. If x=2, you would need to perform 10 reductions before summing the Taylor terms. After summarizing the Taylor terms, 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 performance 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	5719,74
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	1309,860
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

Table 3. Performance as a function of reduction factors applied.

As you can see, you will benefit even more from large precisions by increasing the reduction factor.

Guard Digits

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

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

$2 + \lceil log10(precision) \rceil$ as extra guard digits for working precision.

Source log(x) using Taylor series

```
float_precision log(const float_precision& x)
	{
	size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
	eptype expo;
	int i, k, no_reduction;
	size_t loopcnt = 1;
	double zd;
	float_precision logx, z(x), zsq, terms;
	const float_precision c1(1);

	if (x <= float_precision(0))
		{
		throw float_precision::domain_error();
		}

	expo = z.exponent(); // Get original exponent
	z.exponent(0);       // Set exponent to zero getting z between [1..2)

	// Check for argument reduction and increase precision if necessary
	zd = (double)z;
	no_reduction = (int)ceil(log(log(zd) / log(1.001)) / log(2));
	no_reduction = std::max(no_reduction, 0);
	precision += no_reduction;
```

```
	// adjust precision to allow correct rounding of result
	z.precision(precision);
	zsq.precision(precision);
	terms.precision(precision);
	logx.precision(precision);

	// The fraction part is [1...1.1) (base 10) at this point
	// Reduce z to less than 0.001so range is now [1..1.001)
	for (k = 0; k < no_reduction; ++k)
		z = sqrt(z);
	// number now in [1...1.001). Setup the iteration
	z = (z - c1) / (z + c1);
	zsq = z.square();
	logx = z;

	// Iterate using Taylor series ln(x) == 2(z + z^3/3 + z^5/5 ... )
	for (i = 3;; i += 2, ++loopcnt)
		{
		z *= zsq;
		terms = z / float_precision(i);
		if (logx + terms == logx)
			break;
		logx += terms;
		}


	// Adjust the result from the reduction by multiplying it with 2^(k+1)
	logx *= float_precision(pow(2.0, (double)(k + 1)));
	if (expo != 0)  // Adjust for original exponent y
		{// Ln(x^y) = Ln(x) + Ln(2^y) = Ln(x) + y * ln(2)
		logx += float_precision(expo) * _float_table(_LN2, precision + 1);
		}

	// Round to same precision as argument and rounding mode
	logx.mode(x.mode());
	logx.precision(x.precision());
	return logx;
	}
```

Further Improvement of the methods?

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) again:

$$\ln(x) = 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right) \qquad (6)$$

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

$$\ln(x) = 2\left(z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \cdots\right) \qquad (7)$$

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

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

Moreover, group them:

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

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

Then, you replace one division with three extra multiplications. The (n+2) can be done using a 32-bit or 64-bit integer since you never get to do many Taylor terms in real life. There is no need to stop at just grouping two terms. 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, gained a few more additions and multiplications.

Because arbitrary precision division is much 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.

Iterations Source for five terms scaling of coefficients replacing:

```
// Iterate using Taylor series ln(x) == 2(z + z^3/3 + z^5/5 ... )
for (i = 3;; i += 2, ++loopcnt)
	{
	z *= zsq;
	terms = z / float_precision(i);
	if (logx + terms == logx)
		break;
	logx += terms;
	}
```

With this:

```
std::vector<float_precision> vn(group);
std::vector<float_precision> cn(group);
int j, l;
const int group=5;

// Calculate the next group z e.g. z^2, Z^4, z^6 etc.
for (i = 0; i < group; ++i)
	{
	vn[i].precision(precision);
```

```
        cn[i].precision(precision);
        if (i == 0) vn[i] = zsq;
        if (i > 0) vn[i] = vn[i - 1] * zsq;
        }

// Now iterate
for (i = 3; ; )
        {
        // Calculate the new constant
        for (j = 0; j < group; ++j)
                {
                cn[j] = c1;
                for (l = 0; l < group; ++l)
                        if (j != l)
                                cn[j] *= i + 2 * l;
                }
        // Add the terms together
        for (j = 0, terms = 0; j < group; ++j)
                terms += cn[j] * vn[j];
        terms *= z / (cn[0] * float_precision(i, precision));

        i += 2 * group;      // Update term count
        loopcnt += group;    // Update loop count
        if (logx + terms == logx)   // Reach precision
                break;       // yes terminate loop
        logx += terms;       // Add Taylor terms to result
        if (group > 1)
                z *= vn[group - 1];  // ajust z to last Taylor term in group
        }
```

While the Taylor series plus argument reduction can be effective at moderately high precisions, it can still become burdensome for extremely large digit requirements. Here is where iterative methods like Newton's algorithm come in, potentially offering faster convergence in high-precision contexts, especially if we can compute exp(x) efficiently.

Log(x) using 2nd- and 3rd-order methods.

Before we go into details, we first need to discuss two essential performance improvements that can be applied to the Newton and higher-order methods.

The precision ladder (iterative deepening)

All the root-finding methods in the following sections, Newton, the division-free Newton, Halley, and Chebyshev, share the same scheduling technique, so we describe it once here and refer back to it afterward.

A Newton iteration doubles the number of correct digits per step; Halley and Chebyshev triple it. There is therefore no reason to perform the early steps at the full working precision. A step that will only deliver 60 correct digits can be carried out with 60 digits of working precision, and only the final step needs the full precision P. Since arbitrary

precision multiplication grows faster than linearly in the number of digits, the lower levels of such a ladder are much cheaper than the top one, and the total cost of the whole iteration approaches the cost of a single step at full precision.

Two details decide how much of that saving you actually get.

First, the ladder should be built top-down rather than bottom-up. The natural way to write the loop is to start at 32 digits and double until the working precision is reached, but this wastes the most expensive level. At P = 1,008 the doubling sequence is 63, 126, 252, 504, 992, 1008: the level at 992 digits costs almost as much as the final one and contributes nothing, because 504 digits would already have been enough to land on 1008. Halving down from P instead, and then running the resulting list in reverse, makes every level land exactly on the next one up. In the author's measurements this alone removed 20–27% of the total time for the division-free method at 1,000 and 10,000 digits.

Second, the loop needs a stopping rule that does not cost an extra iteration. The obvious test, "iterate until x + r == x", spends one full-precision evaluation merely confirming that the correction has become negligible. That confirmation is unnecessary, because the correction just applied already bounds the error that remains. For the Newton residual r, the error after the step is approximately $r^2/2$, and for a cubic method $r^3/12$ or $r^3/3$. Checking $|r| \le |x| \cdot 10^{-need}$ after the step therefore certifies the result without evaluating anything further, and the loop can exit immediately after the full-precision step. Where the test fails — an unusually poor starting guess — the level is simply repeated, which is correct but slower; a small repeat cap keeps the loop finite.

The relative cost of the whole iteration, expressed in units of one full-precision evaluation of exp(), then depends only on the convergence order. With multiplication in the Karatsuba range (cost proportional to p^1.58) the level below the top costs (1/order)^1.58 of the final one:

Convergence order	Next level down	Total cost in top-level exp()
2 (Newton)	(1/2)^1.58 ≈ 0.33	≈ 1.50
3 (Halley, Chebyshev)	(1/3)^1.58 ≈ 0.18	≈ 1.21
4 (Schröder order 4)	(1/4)^1.58 ≈ 0.11	≈ 1.12
∞ **(unattainable)**	0	1.00

Table 4. Cost of a laddered iteration relative to one full-precision exp() evaluation.

This table also sets the limit on what a higher convergence order can buy. The full-precision evaluation at the top level is a fixed cost that no order removes, so moving from order 2 to order 3 saves about 20%, from 3 to 4 about 7%, and everything beyond order 4 is capped at a further 10% in total. Orders above four are not worth the additional code.

The same reasoning applies unchanged to the Newton and Halley iterations used elsewhere in the library for the inverse 1/b and for sqrt(). There the effect is even larger,

because a single step costs only a few multiplications and the precision schedule is essentially the whole cost of the routine.

Source of the ladder helpers

```
#include <vector>

// Top-down precision ladder: precision, precision/order, ... down to <= start,
// so each level lands exactly on the next one up. Run it in reverse.
static std::vector<size_t> build_ladder(size_t precision, size_t start, size_t order)
{
	std::vector<size_t> ladder;
	for (size_t d = precision; ; d = (d + order - 1) / order)
	{
		ladder.push_back(d);
		if (d <= start)
			break;
	}
	return ladder;
}
```

Source of the small_enough helper

```
// true if |t| <= |u| * 10^-need  (sufficient test, binary exponents, mantissa in
[1,2))
static inline  bool small_enough(const float_precision& t, const float_precision&
u, size_t need)
{
	if (t.iszero())
		return true;
	const long needbits = (long)ceil(need * 3.321928094887362);  // log2(10)
	return (long)t.exponent() - (long)u.exponent() <= -needbits - 1;
}
```

A typical loop then has the following shape, where the body computes the residual and applies the correction:

```
const std::vector<size_t> ladder = build_ladder(precision, start, order);

for (size_t i = ladder.size(), repeats = 0; i-- > 0; )
    {
    const size_t digits = ladder[i];
    // ... set the working precision to digits + extra, compute the residual r,
    // ... apply the correction to x
    const size_t need = (digits == precision) ? (a.precision() + order) / order
                                              : digits / order - 1;
    if (!small_enough(r, x, need) && repeats < 2)
        { ++i; ++repeats; }   // the level fell short: repeat it
    else
        repeats = 0;
    }
```

Newton Method for log(x)

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

$$x_{n+1} = x_n - 1 + \frac{y}{e^{x_n}} \quad (8)$$

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. Technically, you could just as easily do this to avoid the division:

$$x_{n+1} = x_n - 1 + y \cdot e^{-x_n} \quad (9)$$

It is faster to move the division into the computation of e^{-x_n}. (Usually more efficient letting exp() handle the division, and we observed a 15% performance gain doing this) For precision with fewer than 1,000 digits, the Taylor series from the previous chapter is faster, but above 1,000 digits, the Newton method exceeds the performance of the Taylor series. At 100,000 digits, Newton's method is four times faster than the Taylor series.

Source ln_newton()

```
static float_precision ln_newton(const float_precision& a)
{
	const size_t extra = 5;
	const size_t precision = a.precision() + (size_t)ceil(log10(a.precision())) +
extra;
	const float_precision c1(1);
	double fx;
	float_precision r, x, y(a);

	if (a <= float_precision(0))
	{
		throw float_precision::domain_error();
	}

	y.precision(precision);
	// Do iteration using guard digits higher precision
	x.precision(precision);

	// Get a initial guess using ordinary floating point
	fx = log((double)y);
	x = float_precision(fx);

	const std::vector<size_t> ladder = build_ladder(precision, 32, 2);

	// Now iterate using Netwon x=x(1+y-ln(x))  x=x-1+y/exp(x)
	for (size_t i = ladder.size(), repeats = 0; i-- > 0; )
	{
		const size_t digits = ladder[i];
		// Increase precision by a factor of two for the working variable r
& x.
		r.precision(digits + extra);
		x.precision(digits + extra);
		r = -c1 + y * exp(-x);		// -c1+y/exp(x)
		x += r;				// x=x-c1+y/exp(x)
```

```
                const size_t need = (digits == precision)
                        ? (a.precision() + 2) / 2
                        : (digits > 4 ? digits / 2 - 2 : 1);

                const bool ok =  small_enough(r, x, need);
                if (!ok && repeats < 2) { ++i; ++repeats; }   // repeat this level
                else repeats = 0;
        }

        // Reapply exponent, mode, and precision
        x.mode(a.mode());
        x.precision(a.precision() + 1);
        return x;
}
```

Log(x) using the division-free Newton method (Brent).

Every Newton step in the previous section evaluates y/exp(x), or equivalently y·exp(−x). Brent observed that the division can be removed entirely. Writing r = y·exp(−x), the Newton correction is simply r − 1, and once x has been advanced by d = r − 1, the new residual follows from the old one by

$$r_new = r \cdot \exp(-d) \qquad (10)$$

because r = y·exp(−x) holds by construction. No division and no fresh evaluation of y·exp(−x) is needed, and since d becomes very small after the first step, the exp() call has a tiny argument and needs no argument reduction at all. On paper this looks strictly better than ordinary Newton. In arbitrary precision it is not, for two separate reasons that are both worth mentioning.

The propagated residual is not self-correcting

The direct formulation recomputes r = y·exp(−x) from y on every iteration. That is what makes Newton self-correcting: whatever error is present in x, including the truncation introduced by lowering the working precision on the early levels of the ladder, is measured afresh against y and squashed quadratically on the next pass. The propagated form measures nothing. It relies on the invariant

$$r = y \cdot \exp(-x) \qquad (11)$$

and that invariant silently assumes the update x += d is exact. With a precision ladder it is not: rounding x to the current level moves it by an extra δ that r never learns about. From that point on, r corresponds to a slightly different x than the one being held.

The failure mode is deceptive. Writing r = exp(e) + ε, where e is the true error of x, the correction d = r − 1 carries ε, the update x += d deposits it into x, and the propagation r·exp(−d) ≈ 1 − e²/2 removes ε from r again. The iteration therefore converges

beautifully, d tends to zero quadratically, and the loop exits normally, but it converges to ln(y) + Σδ, dominated by the coarsest level of the ladder. Raising the requested precision does not help. In the author's first implementation, the result was accurate to roughly the 37 working digits of the first level.

The remedy is to stop propagating and to recompute the residual from a fixed quantity every pass. Reducing the argument first makes this cheap. For any value y_0, the identity

$$\ln(a) = y_0 + \ln(a \cdot \exp(-y_0)) \qquad (12)$$

is exact, so one evaluation of $\exp(-y_0)$ with y_0 taken from the ordinary double-precision log() produces $q = a \cdot \exp(-y_0) - 1$ with $|q|$ of order 10^{-16}. Note that y_0 does not have to be accurate, only used consistently; it is q that must be computed at the full working precision, since an absolute error in q passes straight into the result. For arguments already close to unity the reduction is free: $q = a - 1$ is exact and no exp() is needed at all.

What remains is to solve for $u = -\ln(1+q)$, which is the root of

$$f(u) = (1+q) \cdot e^u - 1 = 0 \qquad (13)$$

The residual $t = (1+q) \cdot e^u - 1$ is recomputed from the fixed q on every pass, so the iteration is self-correcting again and the ladder is safe. Since $f' \approx 1$ near the root, the Newton step is simply $u - t$, with no division: writing $e = u - u^*$, we have $t = e^e - 1 = e + e^2/2 + \ldots$ and therefore an error of $-e^2/2$ after the step. Convergence remains quadratic, and the method remains division-free. Finally, $\ln(a) = y_0 - u$.

Two details of the implementation matter. The product must be grouped as $(q \cdot z + z) + q$ rather than $(1+q) \cdot z$, so that the value 1+q is never formed; forming it would require 16 digits more than the level provides. And $e^u - 1$ loses roughly $-\log_{10}|q|$ digits to cancellation, which must be covered by extra guard digits on that one variable. A dedicated expm1() would avoid the cancellation, but for a library whose exp() is already well optimized it is faster to pay the guard digits than to run a separate, less optimized series.

Source ln_newton_divfree()

```
// ln(a) using Brent's division-free Newton iteration, made self-correcting
// by recomputing the residual from a fixed quantity on every pass.
//
// Stage 1 (reduce to ln(1+q)):
//     0.5 <= a < 1.5 :  y0 = 0,  q = a - 1                        (exact, no exp)
//     otherwise      :  y0 = log((double)a),  q = a*exp(-y0) - 1   (one full-precision
exp)
//     ln(a) = y0 + ln(1+q) is exact for whatever value y0 holds
//
// Stage 2 (top-down precision ladder, no division, no confirmation pass):
//     solve f(u) = (1+q)e^u - 1 = 0, whose root is u = -ln(1+q)
//     t = (1+q)(e^u - 1) + q   recomputed from q every pass -> self-correcting
//     u = u - t                f'(u) ~ 1 near the root: quadratic, error -> -e^2/2
//     A level is accepted when |t/u| <= 10^-need, which guarantees the next
//     level (twice the precision) reaches its target. A level that falls
```

```
//     short is repeated, at most twice.
//
// Result: ln(a) = y0 - u
//
// Limitation: (double)a must be representable (no overflow/underflow),
// as in the other Newton variants.
static float_precision ln_newton_divfree(const float_precision& a)
{
         const size_t extra = 5;
         const size_t precision = a.precision() + (size_t)ceil(log10(a.precision())) +
extra;
         const float_precision c1(1);
         float_precision y(a), y0, q, u, z, t;

         if (a <= float_precision(0))
                  throw float_precision::domain_error();

         y.precision(precision);
         y0.precision(precision);
         q.precision(precision);

         // ---- Stage 1: reduce to ln(1+q) ----
         if (y >= float_precision(0.5) && y < float_precision(1.5))
         {
                  y0 = float_precision(0);
                  q = y - c1;
                  // exact
         }
         else
         {
                  y0 = float_precision(log((double)y));// held exactly; its accuracy
is irrelevant
                  q = y * exp(-y0) - c1;      // must be at full precision
         }

         if (!q.iszero())
         {
                  // ---- Initial guess for u = -ln(1+q) ----
                  const double fq = fabs((double)q);
                  // Relative error of the two-term series -q + q^2/2 is ~q^2/3
                  const double seriesdigits = (fq > 0) ? -2.0 * log10(fq) - 0.5 :
(double)precision;
                  size_t startdigits;

                  u.precision(precision);
                  if (seriesdigits > 16)
                  {
                           u = q * q;
                           u.adjustExponent(-1);       // q^2/2, exact
                           u -= q;
                  // -q + q^2/2
                           startdigits = (size_t)(2.0 * seriesdigits);
                  }
                  else
                  {
                           u = float_precision(-log1p((double)q));// ~16 correct
digits
                           startdigits = 32;
                  }
```

```
                    if (seriesdigits < (double)precision)// otherwise the series is
already exact enough
                    {
                              // e^u - 1 loses about -log10|u| ~ -log10|q| digits to
cancellation
                              const size_t cancel = (fq > 0 ? (size_t)std::max(0.0,
ceil(-log10(fq))) : 0) + 2;

                              // ---- Stage 2: top-down ladder, each level doubles
exactly into the next ----
                              const std::vector<size_t> ladder = build_ladder(precision,
startdigits, 2);

                              for (size_t i = ladder.size(), repeats = 0; i-- > 0; )
                              {
                                        const size_t digits = ladder[i];
                                        u.precision(digits + extra);
                                        t.precision(digits + extra);

                                        z.precision(digits + extra + cancel);// guard
digits for the cancellation
                                        z = u;
                                        z = exp(z) - c1;
                                        // e^u - 1
                                        z.precision(digits + extra);

                                        t = (q * z + z) + q;
                                        // (1+q)e^u - 1, never forms 1+q
                                        u -= t;
                                        // error after step ~ (t/u)^2/2
                                        // Final level: a.precision()+1 correct digits;
the rest are guard digits.
                                        // Intermediate level: ~digits correct, so the
next (doubled) level lands.
                                        const size_t need = (digits == precision)
                                                  ? (a.precision() + 2) / 2
                                                  : (digits > 4 ? digits / 2 - 2 : 1);

                                        if (!small_enough(t, u, need) && repeats < 2)
                                        {
                                                  ++i; ++repeats;
                                        // level fell short: repeat it
                                        }
                                        else
                                                  repeats = 0;
                              }
                    }

                    // ---- ln(a) = y0 + ln(1+q) = y0 - u ----
                    u.precision(precision);
                    y0 -= u;
          }

          // Reapply mode and precision
          y0.mode(a.mode());
          y0.precision(a.precision() + 1);
          return y0;
}
```

Implemented correctly, the division-free method is still slower than ordinary Newton, Halley, and Chebyshev at every precision.

With the method implemented correctly, the comparison against ordinary Newton, Halley, and Chebyshev shows that the Newton division-free method is unfavorable at every precision:

Digits	10	100	1,000	10,000	100,000
Newton	0.09	0.23	2.81	454	100,840
Newton Division free (Brent)	0.07	0.20	3.38	710	175,472
Chebyshev	0.07	0.17	2.19	328	80,832
Halley	0.07	0.18	3.14	440	82,082
AGM	0.07	0.15	2.30	200	4,403

Table 5. Timings in milliseconds, all methods using the top-down precision ladder.

The explanation is a cost structure that has no counterpart in IEEE double arithmetic. Counting full-precision evaluations of exp(), which dominate everything else:

• Ordinary Newton with the ladder performs one full-precision exp() at the top level, with the levels below costing roughly 0.33, 0.11, … of it, for a total near 1.5.

• The division-free method pays the Stage 1 $\exp(-y_0)$ at full precision and then runs the entire ladder on top of it, for a total near 2.5.

The reduction to ln(1+q) buys a tiny argument for the later exp() calls, but a well-implemented arbitrary precision exp() already performs its own argument reduction internally, so a tiny argument saves only the first few dozen halving's out of many. The extra full-precision evaluation is not recovered.

The division that was removed, meanwhile, was never expensive. In IEEE double arithmetic a division costs about as much as an exp(), so eliminating one per iteration is a genuine win, which is why the technique is standard in libm implementations. At 10,000 digits, a division costs a few multiplications while an exp() costs hundreds of them. The quantity to minimize in arbitrary precision is the number of full-precision transcendental evaluations, not the number of divisions, and the division-free formulation increases it by one. The same effect is visible in the comparison between Halley and Chebyshev in a later section: at 10,000 digits Halley's division per level still costs it 25% against the division-free Chebyshev step, but by 100,000 digits the two are within 0.3% of each other, because the division has become negligible next to exp().

One case remains where the method is competitive. For arguments already in [0.5, 1.5) the Stage 1 reduction is free, since $q = a - 1$ is exact and requires no exp() at all. That is also precisely the range where the direct Newton residual $y \cdot \exp(-x) - 1$ is weakest, because $x = \ln(a)$ is small there and the residual has an absolute rounding floor. A

dispatcher could reasonably use the division-free form near unity and Chebyshev elsewhere.

The wider lesson is worth stating plainly, because it applies to any technique imported from fixed-precision floating point: an optimization that trades one operation for another is only valid under the cost model it was derived from. Brent's division-free iteration is correct and elegant; it remains quadratically convergent, removes every division from the loop, and, in arbitrary precision, is slower than the method it replaces.

Log(x) using the Chebyshev method.

Newton and Halley are both members of a larger family, and seeing the family makes the choice between them obvious. Start from the exact correction. If x is any approximation to ln(y) and $d = y \cdot \exp(-x) - 1$ is the Newton residual, then

$$x^* = x + \ln(1+d) \qquad (14)$$

holds exactly, where $x^* = \ln(y)$. Every iterative method for ln(x) built on exp() is simply a truncation of ln(1+d), and the methods differ only in how they truncate it:

Method	Approximation of ln(1+d)	Error after the step	Order	Cost per level
Newton	d	$d^2/2$	2	1 exp
Halley	$2d/(2+d)$ (Padé [1/1])	$d^3/12$	3	1 exp + 1 division
Chebyshev	$d - d^2/2$ (Taylor [2/0])	$d^3/3$	3	1 exp + 1 squaring
Schröder order 4	$d - d^2/2 + d^3/3$	$d^4/4$	4	1 exp + 2 multiplications

Table 6. Iterative methods for ln(x) as truncations of the exact correction ln(1+d).

Truncating the logarithm after two terms therefore gives cubic convergence for the price of one squaring. Each level costs one exp(), exactly as in Newton, while the ladder triples as in Halley, and unlike Halley there is no division. This is Chebyshev's method, also known as Chebyshev–Euler; the family of Taylor truncations to which it belongs is Schröder's method of the first kind, while Halley belongs to the Padé branch known as Householder's methods.

The same result follows from the classical Chebyshev formula. For a function f, the step is

$$x \leftarrow x - f/f' - f'' \cdot f^2/(2f'^3) \qquad (15)$$

and with $f(x) = e^x - y$, where $f' = f'' = e^x$, the two terms collapse to $f/f' = -d$ and $f''f^2/(2f'^3) = d^2/2$, giving the update $x \leftarrow x + d - d^2/2$ directly.

Source ln_chebyshev()

```
// ln(a): Newton residual + 2-term log1p correction -> cubic, division free.
//   d = y*exp(-x) - 1,  x* = x + ln(1+d),  x += d - d^2/2,  error ~ d^3/3
static float_precision ln_Chebyshev(const float_precision& a)
{
          const size_t extra = 5;
          const size_t precision = a.precision() + (size_t)ceil(log10(a.precision())) +
extra;
          const float_precision c1(1), half(0.5);
          float_precision d, x, y(a);

          if (a <= float_precision(0))
                    throw float_precision::domain_error();

          y.precision(precision);
          x.precision(precision);
          x = float_precision(log((double)y));          // ~16 correct digits

          const std::vector<size_t> ladder = build_ladder(precision, 48, 3);

          for (size_t i = ladder.size(), repeats = 0; i-- > 0; )
          {
                    const size_t digits = ladder[i];
                    x.precision(digits + extra);
                    d.precision(digits + extra);

                    d = y * exp(-x) - c1;
                    // error before step ~ d
                    x += d - d * d * half;  // Half optimize to an exponent adjustment
                    // error after step ~ d^3/3

                    const size_t need = (digits == precision)
                              ? (a.precision() + 3) / 3
                              : (digits > 6 ? digits / 3 - 1 : 1);

                    const bool ok = small_enough(d, x, need);
                    if (!ok && repeats < 2) { ++i; ++repeats; }
                    else repeats = 0;
          }

          x.mode(a.mode());
          x.precision(a.precision() + 1);
          return x;
}
```

Chebyshev is the fastest of the exp()-based methods at every precision above 10 digits, by 25–40% over laddered Newton and by up to 25% over Halley:

Digits	Newton y·exp(−x)	Halley	Chebyshev	AGM
100	0.257	0.167	0.162	0.144
1,000	2.82	2.69	2.04	2.16
10,000	461	445	333	194
100,000	101,010	82,553	82,836	4,809

Table 7. Timings in milliseconds for the laddered methods.

Two features of the table are worth commenting on. Chebyshev's advantage over Halley disappears at 100,000 digits, where the two agree to within the noise of a single run: by that size exp() dominates so completely that Halley's division per level no longer registers, and the two methods share the same tripling ladder and the same number of exp() calls. And neither can approach AGM at high precision. All the exp()-based methods cost a fixed multiple of one full-precision exp() and therefore inherit its growth rate, which in this library is a factor of roughly 250 for a tenfold increase in digits, whereas AGM is O(M(P)·log P) and grows by a factor near 23. Chebyshev is the method of choice when AGM is unavailable, and the best fallback across the whole precision range rather than only at high precision.

Extending the truncation to order 4, x += d·(1 − d·(½ − d/3)) with the ladder built on order 4, is a two-line change; from the cost table in the ladder section, it should be worth about 7%. The division by 3 is not a real cost, since 1/3 is computed once at full precision and each step multiplies by it. Orders beyond four are not worthwhile.

Log(x) using the Halley method.

Since the Newton method is faster than the Taylor series for precision above 1,000 digits, it is interesting to check if the cubic-convergence Halley method is even quicker. Halley method is the Padé alternative that gives the same order as Chebyshev at the cost of an extra division. The Halley method with cubic convergence is:

$$x_{n+1} = x_n + 2\frac{y - e^{x_n}}{y + e^{x_n}} \qquad (16)$$

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 around 100,000 digits of precision, and is approximately 20% faster than the Newton Method. Below 10,000, Newton and Halley performed similarly.
Although Halley's iteration can converge faster per step, it also involves an exp(x) evaluation each time, just like Newton's. In practice, the extra overhead is offset by the fact that we gain more correct digits per iteration.

Source ln_halley()

```
static float_precision ln_halley(const float_precision& a)
         {
         const size_t extra = 5;
         const size_t precision = a.precision() + (size_t)ceil(log10(a.precision())) +
extra;
         const float_precision c1(1);
         double fx;
         float_precision r, x, y(a);

         if (a <= float_precision(0))
                  {
                  throw float_precision::domain_error();
                  }
```

```
        // Do iteration using guard digits higher precision
        y.precision(precision);
        x.precision(precision);

        // Get a initial guess using ordinary floating point
        fx = log((double)y);
        x = float_precision(fx);
        const std::vector<size_t> ladder = build_ladder(precision, 48, 3);
        // Now iterate using Halley x=x-2(exp(x)-y)/(exp(x)+y)
        for (size_t i = ladder.size(), repeats = 0; i-- > 0; )
                {
                const size_t digits = ladder[i];
                // Increase precision by a factor of three for the working variable
r & x.
                r.precision(digits+extra);
                x.precision(digits+extra);
                r = exp(x);                 // exp(x)
                r = (y - r) / (r + y);      // r=(y-(exp(x))/(exp(x)+y)
                r.adjustExponent(+1);       // r*=2;
                x += r;                     // x=x+2(y-exp(x))/(exp(x)+y)

                const size_t need = (digits == precision)
                        ? (a.precision() + 3) / 3
                        : (digits > 6 ? digits / 3 - 1 : 1);

                const bool ok = small_enough(r, x, need
                if (!ok && repeats < 2) { ++i; ++repeats; }
                else repeats = 0;
                }

        // Reapply exponent, mode, and precision
        x.mode(a.mode());
        x.precision(a.precision() + 1);
        return x;
        }
```

At very high digit counts, neither Taylor expansions nor root-finding methods can match the asymptotic efficiency of the AGM algorithm. Inspired by the same approach that revolutionized π computations, applying the arithmetic-geometric mean for log(x) uses fewer iterations to achieve each additional digit, ultimately dominating other methods for large precisions.

Log(x) using the AGM method.

The AGM method is the method that has the best asymptotic performance of all the methods. It was found around 1975 and is described in the Yacas [5]:

$$\ln(x) = \pi \cdot x \frac{1+\frac{4}{x^2}\left(1-\frac{1}{\ln(x)}\right)}{2 \cdot AGM(x,4)} \qquad (17)$$

It looks more complex than any of the other methods. However, 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. If x is not "large enough," we must 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 $s \cdot \ln(2)$:

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) \quad (18)$$

For a given precision, P, s is found using the formula below:

$$s = P\frac{\ln(10)}{2\cdot\ln(2)} + 1 - \frac{\ln(x)}{\ln(2)} \quad (19)$$

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

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) = \frac{\pi \cdot x^{s-2}}{2 \cdot AGM(x^{s-2},1)} - s \cdot \ln(2), \text{for } x > 1 \quad (20)$$

If x<1, then we use the identity $\ln(x) = -\ln(\frac{1}{x})$ and use the AGM algorithm with 1/x. Even though we are using two arbitrary precision constants, π and ln(2), that need to be calculated to the same precision, P, and we need to perform approximately $2\frac{\ln(P)}{\ln(2)}$ iterations to calculate the AGM value outperformed any other methods presented here for precision exceeding approximately 200 digits. See the log(x) performance chart.

AGM Algorithm

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

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

Algorithm 1

In arbitrary precision, the source would look like this:

Source AGM()

```
float_precision AGM(const float_precision a, const float_precision b)
	{
	const int guard = 0;
```

```
	size_t precision = std::max(a.precision(), b.precision())+guard;
	size_t loopcnt;
	float_precision x(a), y(b), xnew(a), ynew(b);
	float_precision diff;

	x.precision(precision);
	y.precision(precision);
	xnew.precision(precision);
	ynew.precision(precision);
	for (loopcnt=1;;++loopcnt)
		{
		xnew = 0.5*(x + y);
		ynew = sqrt(x*y);
		diff = xnew - ynew;
		if (diff.iszero() || xnew==x||ynew==y)
			break;
		x = xnew;
		y = ynew;
		}

	xnew.precision(precision - guard);
	return xnew;
	}
```

Source ln_AGM()

```
float_precision lnAGM(const float_precision& x)
	{
	const size_t guard = 5;
	const size_t precision = x.precision() +
(size_t)ceil(log10(x.precision()))+guard;
	const uintmax_t s = (uintmax_t)ceil(precision*log(10) / (2 * log(2)) + 1 -
log((double)x) / log(2));
	// slost is loss of precision
	const uintmax_t slost = (uintmax_t)ceil(log((double)s) / log(10.0));
	const float_precision c1(1);
	float_precision logx, z(x), agm;

	if (x <= float_precision(0))
		{
		throw float_precision::domain_error();
		}

	// Adjust to working precision
	agm.precision(precision);
	logx.precision(precision);
	z.precision(precision);

	if(z < c1)
		z = 1 / z; // Now z >= 1

	logx = _float_table(_PI, precision);
	z.adjustExponent(s-2);      //z=x^(s-2)
	logx *= z;                  //PI*x^(s-2)
	agm = AGM(z, 1);
```

```
	agm.adjustExponent(+1);		//2*AGM
	logx /= agm;			//(PI*x^(s-2))/(2*AGM)
	// Increase precision to avoid loss of significant
	// when subtracting two large numbers
	logx.precision(precision + slost);
	logx -= float_precision(s,precision+slost) *_float_table(_LN2,
precision+slost);

	// Round to the same precision as argument and rounding mode
	if (x < c1)
		logx.change_sign();
	logx.mode(x.mode());
	logx.precision(x.precision());
	return logx;
	}
```

Log(x) performance

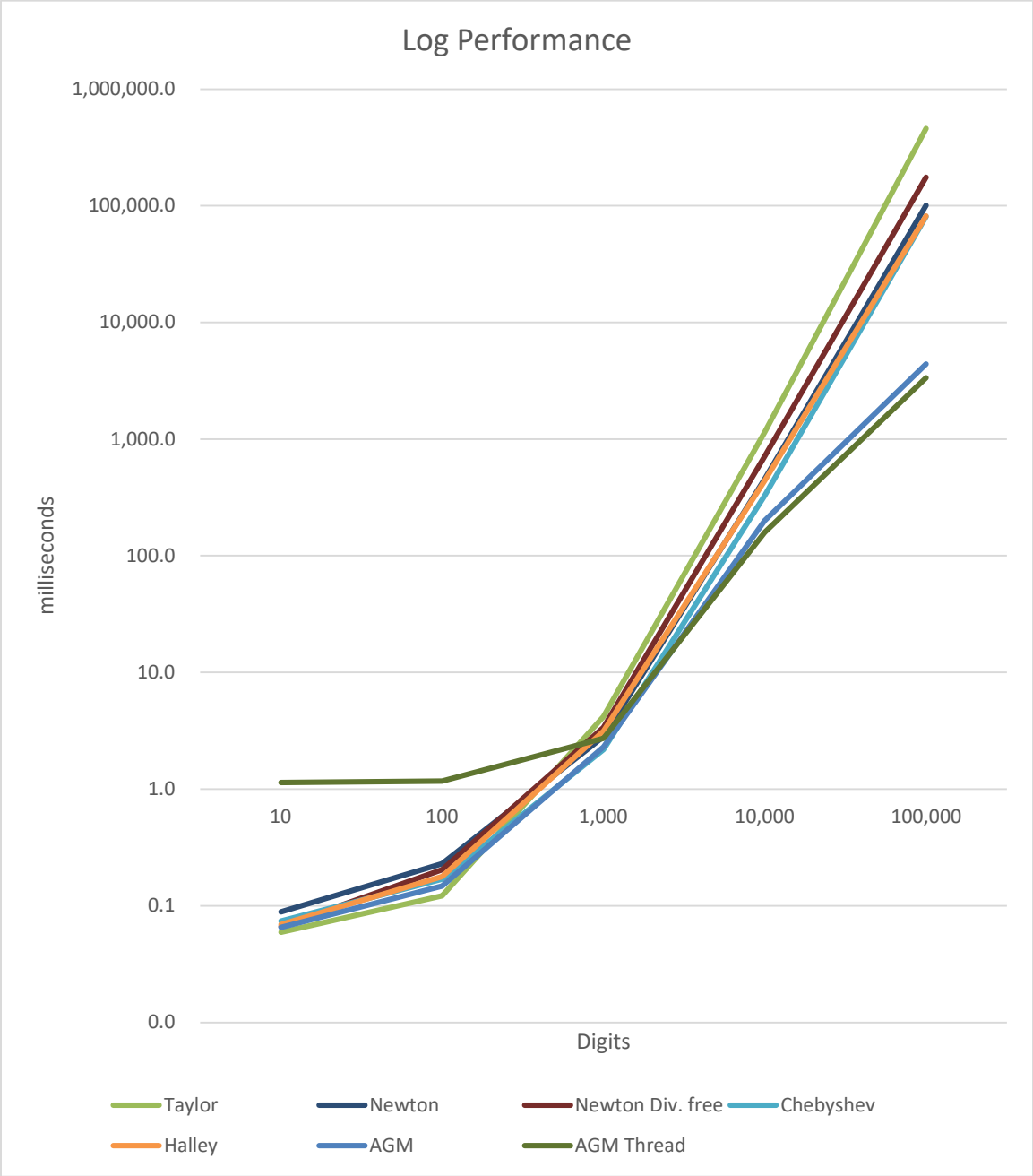


Figure 1. Log(x) performance chart.

Digits	10	100	1,000	10,000	100,000
Taylor	0.06	0.12	4.18	1,145	461,310
Newton	0.09	0.23	2.81	454	100,840
Newton Div. free	0.07	0.20	3.38	710	175,472
Chebyshev	0.07	0.17	2.19	328	80,832

Halley	0.07	0.18	3.14	440	82,082
AGM	0.07	0.15	2.30	200	4,403
AGM Thread	1.14	1.17	2.73	158	3,363

Table 8. All times in milliseconds

Taylor is fastest up to roughly 200–300 digits; the exact crossover is within measurement noise. AGM is fastest above that, and the threaded version takes over at around 2,500 digits. Every exp()-based method overtakes Taylor between 100 and 1,000 digits. Among them, Chebyshev is fastest from 100 digits upward: about 10% ahead of Newton at 1,000 digits and 26% at 10,000. Halley catches up with Chebyshev at 100,000 digits.

Log(x) using the AGM method and multiple threads.

The AGM method lends itself to being implemented using threads. There are three essential 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.

The threaded version of lnAGM becomes:

Threaded lnAGM() source

```
float_precision logAGMThread(const float_precision& x)
	{
	const size_t guard = 5;
	const size_t precision = x.precision() + (size_t)ceil(log10(x.precision()))
+ guard;
	const uintmax_t s = (uintmax_t)ceil(precision*log(10) / (2 * log(2)) + 1 -
log((double)x) / log(2));
	const uintmax_t slost = (uintmax_t)ceil(log((double)s) / log(10.0));  //
Loss of precision
	const float_precision c1(1);
	float_precision logx, z(x), agm, ln2;

	if (x <= float_precision(0))
		{
		throw float_precision::domain_error();
		}

	// Adjust to working precision
	agm.precision(precision);
	logx.precision(precision);
	z.precision(precision);
	ln2.precision(precision);

	if (z < c1)
```

```
            z = 1 / z;          // Now z >= 1
      z.adjustExponent(s - 2);    // z/=4

      // First thread calculates PI
      std::thread first([=, &logx]()
            { logx = _float_table(_PI, precision); });
      // Second thread calculate ln(2)
      std::thread second([=, &ln2]()
            { ln2=_float_table(_LN2, precision + slost); });
      // Third thread calculate AGM(z,1)
      std::thread third([=, &agm, &z]()
            { agm = AGM(z,1);
            agm.adjustExponent(+1);  //2*AGM
            });
      // Wait for threads 1 & 3 to finish
      first.join();
      third.join();

      logx *= z;    //PI*x^(s-2)
      logx /= agm;  // PI*x^(s-2)/AGM

      // Wait for ln(2) thread to finish
      second.join();
      // Increase precision to avoid loss of significance when subtracting two
large numbers
      logx.precision(precision + slost);
      logx -= float_precision(s, precision + slost) * ln2;

      // Round to the same precision as argument and rounding mode
      if (x < c1)
            logx.change_sign();
      logx.mode(x.mode());
      logx.precision(x.precision());
      return logx;
      }
```

Recommendation for calculating log(x)

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

- If the AGM method is unavailable, use Chebyshev's method at any precision. It is the fastest of the exp()-based methods from 10 digits upward; it is division-free, and it is a two-line change from the Newton iteration. Halley is an acceptable alternative and ties Chebyshev above roughly 50,000 digits, but it is never faster below that.
- Surprisingly, the Newton division-free method performs worse than the highly optimized Newton or Chebyshev methods
- Use the Taylor series with argument reduction and coefficient scaling up to approximately 200–300 digits. Coefficient scaling (grouping of terms) remains worth a factor of two to three, and argument reduction should always be applied.

- Use the AGM method above for approximately 300 digits. AGM already matches the Taylor series at a few hundred digits and is 3× faster by 2,000 digits.
- Use the multi-threaded AGM above for approximately 2,500 digits, where the thread overhead of about 1 ms has been amortized.
- Do not use the division-free Newton iteration in preference to Chebyshev. It is included in this paper because the reason it fails is instructive, not because it is competitive.
- Always use the top-down precision ladder with an exponent-based stopping test. It is worth 44-54% in performance improvement for every iterative method here, and the same technique applies to the Newton and Halley iterations used for the inverse and for sqrt().

For a library that dispatches automatically, the resulting hybrid is: the Taylor series below about 300 digits, single-threaded AGM from there to about 2,500 digits, and multi-threaded AGM above that. Note that this moves the Taylor-to-AGM switch in the existing hybrid implementation down from 2,500 digits to about 300, which accounts for the hybrid being slower at 2,000 digits than at 3,000 in the table above.

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 [3] presents a method utilizing a spigot-like algorithm, which is considerably faster for precision up to 10,000 digits. However, [8] and [9] 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 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 [3], 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. ln(2) with 100 digits precision you call spigot_lnxy_64(2,1,100,4); for ln(10) you call spigot_lnxy_64(10,1,100,4); and finally if you want a fraction e.g. ln(1.25) you called it as spigot_lnxy_64(10,8,100,4);

Source spigot_lnxy_64()

```
//64-bit version of spigot algorithm for LN(x/y) fraction
// It has automatic 64-bit integer overflow detection in which case the result
starts with the string "Overflow...."
// A Column: x-1,x-1,x-1,...,x-1
// B Column: x,x,x,x,x,...,x
// Initialization values: (x-1)/(x(n+1))...
```

```
std::string spigot_lnxy_64(unsigned int x, unsigned int y, int digits, int no_dig
= 1)
	{
	static unsigned long f_table[] = { 1, 10, 100, 1000, 10000, 100000,
1000000, 10000000, 100000000 };
	bool first_time = true;		// First iteration of the algorithm
	bool overflow_flag = false;	// 64bit integer overflow flag
	bool bit32_overflow = false;
	char buffer[32];
	std::string ss;			// The std::string that holds the ln(x)
	int dig;
	unsigned int car, no_carry = 0;
	unsigned int no_terms;		// No of terms to complete as a function
of digits
	unsigned long f;		// New base 1 decimal digits at a time
	unsigned long dig_n;		// dig_n holds the next no_dig digit to
add
	unsigned _int64 carry;
	unsigned _int64 tmp_n, tmp_dn;
	ss.reserve(digits + 16);
	int factor;

	if (x < y) return std::string("Domain Error of argument.Required x>y");
	if (x <= 0) return std::string("Domain Error of argument. Required x>0");

	if (no_dig > 8) no_dig = 8;		// ensure no_dig<=8
	if (no_dig < 1) no_dig = 1;		// Ensure no_dig>0
	// Since we do it in trunks of no_dig digits at a time we need to ensure
digits is divisble with no_dig.
	dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
	dig += no_dig;					// Extra guard
digits
	// Calculate the number of terms needed
	factor = (int)ceil(10 * log(0.5) / log((double)(x - y) / (double)x));
	no_terms = (unsigned int)(factor * dig / 3 + 3);
	// Allocate the needed accumulators
	unsigned _int64 *acc_n = new unsigned _int64[no_terms + 1];
	unsigned _int64 *acc_dn = new unsigned _int64[no_terms + 1];
	f = f_table[no_dig];		// Load the initial f
	carry = 0;			// Set carry to 0
	//Loop for each no_dig
	for (int i = dig; i >= 0 && overflow_flag == false; i -= first_time == true
? 1 : no_dig, first_time = false)
		{
		// Calculate new number of terms needed
		no_terms = (unsigned int)(factor * i / 3 + 3);
		// Loop for each no_terms
		for (int j = no_terms; j>0 && overflow_flag == false; --j)
			{
			if (first_time == true)
				{// Calculate the initialize value
				tmp_dn = (j + 1) * x;
				tmp_n = (x - y);
				}
			else
				{
				tmp_n = acc_n[j];
				tmp_dn = acc_dn[j];
```

```
                        }
                if (tmp_n > (ULLONG_MAX) / f)
                        overflow_flag = true;
                tmp_n *= f;             // Scale it
                // Check for 64bit overflow. Not very likely
                if (carry > 0 && tmp_dn > (ULLONG_MAX - tmp_n) / carry)
                        overflow_flag = true;
                tmp_n += carry * tmp_dn;
                carry = (tmp_n / (x * tmp_dn));
                carry *= (x - y);
                acc_n[j] = tmp_n % (tmp_dn * x);
                acc_dn[j] = tmp_dn;
                }

        if (first_time == true)
                {
                tmp_n = (x - y) * f;
                if (carry > 0 && tmp_n > (ULLONG_MAX - carry * x))
                        overflow_flag = true;

                acc_n[0] = (tmp_n + carry*x);
                acc_dn[0] = x;
                dig_n = (unsigned)(acc_n[0] / (f*acc_dn[0]));
                }
        else
                {
                if (acc_n[0] > (ULLONG_MAX - carry * acc_dn[0]) / f)
                        overflow_flag = true;
                dig_n = (unsigned)((acc_n[0] * f + carry * acc_dn[0]) /
(f*acc_dn[0]));
                }
        car = (unsigned)(dig_n / f);
        dig_n %= f;
        // Add the carry to the existing number for ln(x/y) calculated.
        if (car > 0)
                {
                ++no_carry;
                for (int j = ss.length(); car > 0 && j > 0; --j)
                        {
                        int dd;
                        dd = (ss[j - 1] - '0') + car;
                        car = dd / 10;
                        ss[j - 1] = dd % 10 + '0';
                        }
                }
        (void)sprintf(buffer, "%0*lu", first_time == true ? 1 : no_dig,
dig_n);
        ss += std::string(buffer);
        if (first_time == true)
                acc_n[0] %= f*acc_dn[0];
        else
                {
                acc_n[0] = acc_n[0] * f + carry *acc_dn[0];
                acc_n[0] %= f  * acc_dn[0];
                }
        carry = 0;
        }
```

```
	ss.insert(1, ".");// add a come after the first digit to create 2.30...
	if (overflow_flag == false)
		ss.erase(digits + 1); // Remove the extra digits that we didnt
requested.
	else
		ss = std::string("Overflow:") + ss;

	delete[] acc_n, acc_dn;
	return ss;
	}
```

Xiao binary splitting for log(2)

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

$$log(2) = \frac{1}{4}\sum_{k=1}^{n} \frac{3927264b^3-4300512b^2+1209726b-8189}{k(2k-1)(4k-1)(4k-3)} \prod_{i=1}^{k} \frac{i(2i-1)(4i-1)(4i-3)}{1440(10i-1)(10i-3)(10i-7)(10i-9)} \quad (21)$$

Not as voluminous as some of the previous Zuniga series.

Algorithm: Xiao Binary splitting method for log(2)

$set\ m = \frac{a+b}{2}\ integer\ division$
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)=3927264b^3-4300512b^2+1209726b-81891
Q(b-1,b)=1440 (10b-1)(10b-3)(10b-7)(10b-9)
R(b-1,b)=b(2b-1)(4b-1)(4b-3)

Algorithm 2. Xiao Log(2) Algorithm.

And then

$$\log(2) = \frac{1}{4}\frac{P(0,n)}{Q(0,n)} + O(450000^{-n}) \quad (22)$$

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:

$$n = \left\lceil P \frac{\ln(10)}{\ln(45000)} \right\rceil \quad (16)$$

Source Xiao binary splitting method

```
// Now define computeLn2Xiao with an embedded lambda
//
static float_precision computeLn2Xiao(const uintmax_t precision)
{
        // Decide how many terms, etc.
        const intmax_t kmax = intmax_t(std::ceil(precision * std::log(10) /
std::log(450000)));
        const size_t   workprec = size_t(std::ceil(precision + 1 + 1 * std::log(kmax)));

        // Prepare accumulators
        int_precision p, q, r;
        float_precision fp, fq;

        // 1) Define the recursive lambda as a std::function so it can call itself
        std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)>
                binarysplittingXiao;

        binarysplittingXiao =
                [&](uintmax_t a, uintmax_t b, int_precision& pRef, int_precision& qRef,
int_precision& rRef)
        {
                if (a + 1 == b)
                {
                        const uintmax_t b4 = 4 * b;
                        const uintmax_t b10 = 10 * b;
                        const int_precision bb(b);

                        qRef = int_precision(1440) * int_precision((b10-1)*(b10-3)) *
int_precision((b10-7)*(b10-9));
                        rRef = int_precision(b*(2*b-1)) * int_precision((b4 - 1) * (b4 - 3));
                        pRef = ((int_precision(3927264) * bb
                                        - int_precision(4300512)) * bb
                                        + int_precision(1209726)) * bb
                                        - int_precision(81891);
                        return;
                }

                // If you had the special (a + 2 == b) case, you can put it here

                // 2) Recurse: split [a..b] into [a..mid] and [mid..b]
                const uintmax_t mid = (a + b) / 2;
                int_precision pp, qq, rr;

                binarysplittingXiao(a, mid, pRef, qRef, rRef);
                binarysplittingXiao(mid, b, pp, qq, rr);

                // 3) Merge results (combine partial results)
                pRef = pRef * qq + pp * rRef;
                qRef *= qq;
                rRef *= rr;
        };

        // 4) Actually use the recursive lambda
        binarysplittingXiao(0, kmax, p, q, r);
        q *= int_precision(4);
        // 5) Convert to float_precision and finish
        fp.precision(workprec);
        fq.precision(workprec);

        // Move q into a float_precision
        fp = float_precision(p, workprec);
        fq = float_precision(q, workprec);
        fp /= fq;
        fp.precision(precision);

        return fp;
```

```
}
```

Zuniga's binary splitting method for log(2)

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

$$log(2) = \frac{1}{2}\sum_{k=1}^{n} \frac{1794k-297}{k(2k-1)} \prod_{i=1}^{k} \frac{i(2i-1)}{216(6i-1)(6i-5)} \quad (23)$$

It is more or less as complex as the Xiao method.

Algorithm: Zuniga Binary splitting method for Log(2)

$set\ m = \frac{a+b}{2}\ integer\ division$
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)=1794b-297
Q(b-1,b)=216(6b-1)(6b-5)
R(b-1,b)=b(2b-1)

Algorithm 3. Xuniga Log(2) method.

And then

$$\log(2) = \frac{1}{2}\frac{P(0,n)}{Q(0,n)} + O(3888^{-n}) \quad (24)$$

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:

$$n = \left\lceil P\frac{\ln(10)}{\ln(3888)} \right\rceil \quad (25)$$

Source Zuniga-II binary splitting method.

```
// Now define computeLn2Zuniga with an embedded lambda
//
static float_precision computeLn2ZunigaII(const uintmax_t precision)
{
        // Decide how many terms, etc.
        const intmax_t kmax = intmax_t(std::ceil(precision * std::log(10) /
std::log(3888)));
        const size_t   workprec = size_t(std::ceil(precision + 1 + 1 * std::log(kmax)));

        // Prepare accumulators
        int_precision p, q, r;
        float_precision fp, fq;
```

```
        // 1) Define the recursive lambda as a std::function so it can call itself
        std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)>
                binarysplittingZuniga;

        binarysplittingZuniga =
                [&](uintmax_t a, uintmax_t b, int_precision& pRef, int_precision& qRef,
int_precision& rRef)
        {
                if (a + 1 == b)
                {
                        const uintmax_t b6 = 6 * b;

                        qRef = int_precision(216 * (b6 - 1) * (b6 - 5));
                        rRef = int_precision(b * (2 * b - 1));
                        pRef = int_precision(1794 * b - 297);
                        return;
                }

                // If you had the special (a + 2 == b) case, you can put it here

                // 2) Recurse: split [a..b] into [a..mid] and [mid..b]
                const uintmax_t mid = (a + b) / 2;
                int_precision pp, qq, rr;

                binarysplittingZuniga(a, mid, pRef, qRef, rRef);
                binarysplittingZuniga(mid, b, pp, qq, rr);

                // 3) Merge results (combine partial results)
                pRef = pRef * qq + pp * rRef;
                qRef *= qq;
                rRef *= rr;
        };

        // 4) Actually use the recursive lambda
        binarysplittingZuniga(0, kmax, p, q, r);
        q *= int_precision(2);
        // 5) Convert to float_precision and finish
        fp.precision(workprec);
        fq.precision(workprec);

        // Move q into a float_precision
        fp = float_precision(p, workprec);
        fq = float_precision(q, workprec);
        fp /= fq;
        fp.precision(precision);

        return fp;
}
```

Comparison of the Log(2) methods

Having introduced each specialized approach—spigot, Xiao, and Zuniga—and the more general log(x) routine, we can now compare them in terms of accuracy, convergence rates, and practical runtime. Each approach has its sweet spot regarding precision needed and implementation cost.

The four Log(2) methods are listed below.

Method	Implementation	Error	N(P), *P=precision*
Log(x)	Combination of the Taylor series and the AGM	Type equation here.	

Spigot Lnxy_64	Spigot		
Xiao	Binary-Splitting	$O(450000^{-n})$	0.18P
Zuniga II	Binary-Splitting	$O(3888^{-n})$	0.27P

Table 9. 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.

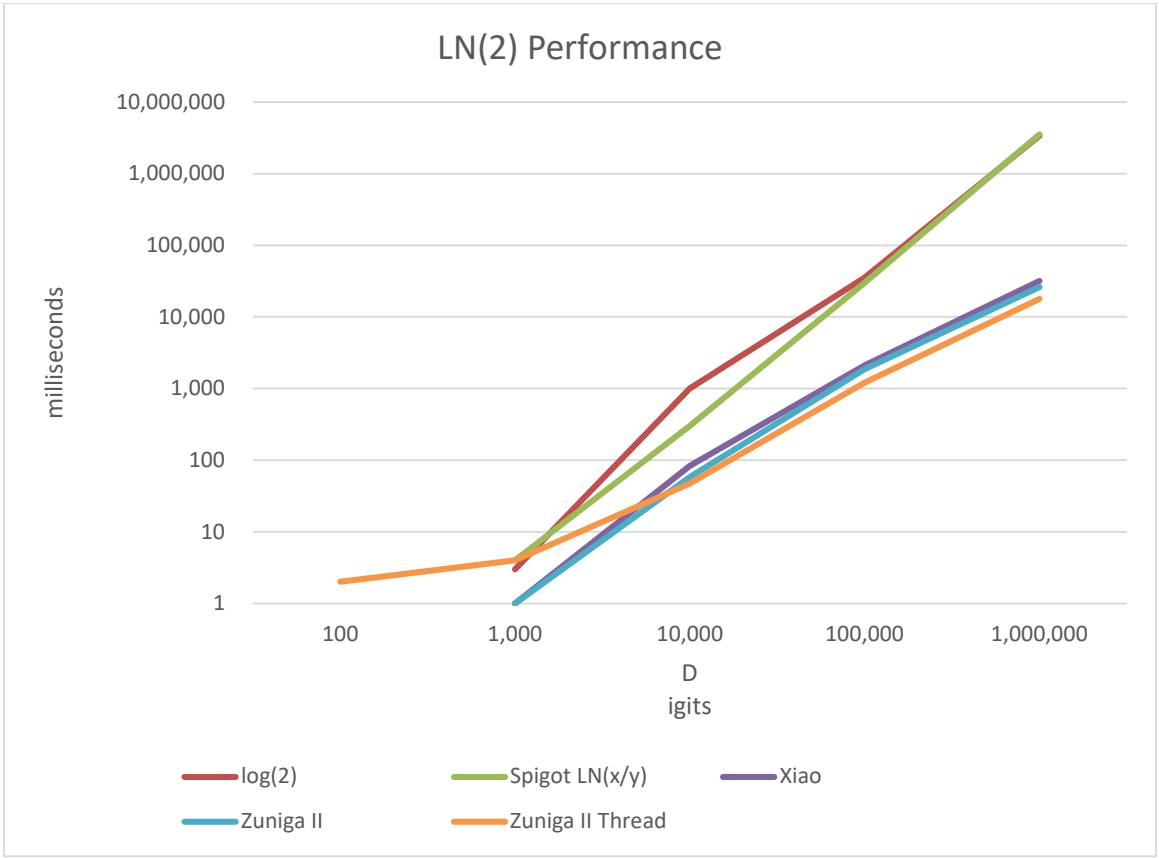


Figure 2. Log(2) performance chart.

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

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
Zuniga II Thread	8	2	4	47	1,204	17,872

Table 10. Performance of the 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. A threaded version of the Zuniga method is approximately 30% faster in line with expectations. Although the Spigot LN(x/y) only uses native integer arithmetic, it quickly falls behind.

Recommendation for the log(2) constant

I recommend the following:

1. It is clear that if you are serious, you would 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.
3. Implement the threaded version of the binary splitting method if speed is of the essence, achieving a 30% performance increase above 10,000 digits.

The constant Log(3)

Log(3) using:

- The general Log(x) algorithm is described in the previous section.
- The spigot-like method lnxy_64.
- Zuniga-I binary splitting method.
- Zuniga-IG4 binary splitting method.

The lnxy_64 method has already been described under log(2). Therefore, we will proceed directly to Zuniga's two methods for log(3). Although the lnxy_64 spigot-like approach can handle ln(3), its performance does not match the specialized methods described below when high precision is required. In particular, Zuniga's binary splitting techniques leverage series expansions tailored to ln(3), resulting in significantly faster convergence. We now explore two such variants, both designed for efficient high‑precision computation.

Zuniga's binary splitting method for log(3)

The constant ln(3) has applications in combinatorics, information theory, and other domains, making fast computation useful in certain specialized fields. Zuniga's interest in ln(3) led to multiple series and product expansions, each balancing complexity against convergence speed. We begin with the simplest of these methods. From [9], we get, after changing the more efficient notation using the Pochhammer symbol into the series below, suitable for implementation using binary splitting.

$$log(3) = \sum_{k=1}^{n} \frac{176k-}{2k(2k-1)} \prod_{i=1}^{k} \frac{2i(2i-1)}{27(6i-1)(6i-5)} \quad (26)$$

The heart of the method is a series expansion that expresses ln(3) as a rational combination of polynomials and Pochhammer products. This structure enables us to apply the same divide-and-conquer merging strategy used for ln(2) but with different polynomial and product definitions.

Algorithm: Zuniga Binary splitting method for Log(3)

$set\ m = \frac{a+b}{2}\ integer\ division$
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)=176b-28
Q(b-1,b)=27(6b-1)(6b-5)
R(b-1,b)=2b(2b-1)

Algorithm 4. Zuniga Log(3) method.

And then

$$\log(3) = \frac{P(0,n)}{Q(0,n)} + O(243^{-n}) \quad (27)$$

These methods have a linearly convergent cost of ~1.46. it is straightforward to implement but is not as efficient as the second of Zuniga methods.

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(243)} \right\rceil \quad (28)$$

The above equation tells us you get approximately. 2.5 decimal digits accuracy per split.

Source Zuniga I

```
// Now define computeLn3Zuniga with an embedded lambda
//
static float_precision computeLn3ZunigaI(const uintmax_t precision)
```

```
{
	// Decide how many terms, etc.
	const intmax_t kmax = intmax_t(std::ceil(precision * std::log(10) / std::log(243)));
	const size_t   workprec = size_t(std::ceil(precision + 1 + 1 * std::log(kmax)));

	// Prepare accumulators
	int_precision p, q, r;
	float_precision fp, fq;

	// 1) Define the recursive lambda as a std::function so it can call itself
	std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)>
		binarysplittingZuniga;

	binarysplittingZuniga =
		[&](uintmax_t a, uintmax_t b, int_precision& pRef, int_precision& qRef,
int_precision& rRef)
	{
		if (a + 1 == b)
		{
			const uintmax_t b6 = 6 * b;

			qRef = int_precision(27 * (b6 - 1) * (b6 - 5));
			rRef = int_precision(2 * b * (2 * b - 1));
			pRef = int_precision(2* 88 * b - 2* 14);
			return;
		}

		// If you had the special (a + 2 == b) case, you can put it here

		// 2) Recurse: split [a..b] into [a..mid] and [mid..b]
		const uintmax_t mid = (a + b) / 2;
		int_precision pp, qq, rr;

		binarysplittingZuniga(a, mid, pRef, qRef, rRef);
		binarysplittingZuniga(mid, b, pp, qq, rr);

		// 3) Merge results (combine partial results)
		pRef = pRef * qq + pp * rRef;
		qRef *= qq;
		rRef *= rr;
	};

	// 4) Actually use the recursive lambda
	binarysplittingZuniga(0, kmax, p, q, r);

	// 5) Convert to float_precision and finish
	fp.precision(workprec);
	fq.precision(workprec);

	// Move q into a float_precision
	fp = float_precision(p, workprec);
	fq = float_precision(q, workprec);
	fp /= fq;
	fp.precision(precision);

	return fp;
}
```

While the first Zuniga method for ln(3) is relatively straightforward, it converges at a moderate rate. By contrast, Zuniga's second approach, sometimes labeled "I-G4," uses a more complex polynomial but can deliver higher digits of accuracy per term. Below, we outline how that method is constructed and how its complexity compares in practice.

$$log(3) = \sum_{k=1}^{n} \frac{P(k)}{128k(2k-1)(4k-1)(4k-3)(8k-1)(8k-3)(8k-5)(8k-7)} \prod_{i=1}^{k} \frac{128i(2i-1)(4i-1)(4i-3)(8i-1)(8i-3)(8i-5)(8i-7)}{Q(i)} \quad (29)$$

where P(k)= $2659024319283200k^7 - 8084203069505536k^6 + 9949321961406464k^5 - 6349069378355200k^4 + 2229033948170240k^3 - 417891652274944k^2 + 36428229376416k - 935651486700$

And

$Q(i)=531441(24i-1)(24i-5)(24i-7)(24i-11)(24i-13)(24i-17)(24i-19)(24i-23)$

Algorithm: Zuniga Binary splitting method for Log(3)

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

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)=$2659024319283200b^7 - 8084203069505536b^6 + 9949321961406464b^5 - 6349069378355200b^4 + 2229033948170240b^3 - 417891652274944b^2 + 36428229376416b - 935651486700$

Q(b-1,b)=531441(24b-1)(24b-5)(24b-7)(24b-11)(24b-13)(24b-17)(24b-19)(24b-23)

R(b-1,b)=128b(2b-1)(4b-1)(4b-3)(8b-1)(8b-3)(8b-5)(8b-7)

Algorithm 5. Xuniga (iiG4) Log(3) method.

And then

$$\log(3) = \frac{P(0,n)}{Q(0,n)} + O(3486784401243^{-n}) \quad (30)$$

This method also has a linearly convergent cost of ~1.46. It is way more complex to implement.

For *n* terms, the error is $O(3486784401243^{-n})$ And for a given precision, *P,* you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(3486784401243)} \right\rceil \quad (31)$$

The above equation tells us you get approximately. 9.5 decimal digits accuracy per split. The convergence cost is the same for the two methods. One requires less computation per split, but on the other hand, it requires nearly four times as many splits as the above method.

Source Zuniga IG4

```
// Now define computeLn3Zuniga with an embedded lambda
//
static float_precision computeLn3ZunigaIG4(const uintmax_t precision)
{
        // Decide how many terms, etc.'
        const intmax_t kmax = intmax_t(std::ceil(precision * std::log(10) /
std::log(3'486'784'401)));
        const size_t   workprec = size_t(std::ceil(precision + 1 + 1 * std::log(kmax)));

        // Prepare accumulators
        int_precision p, q, r;
```

```
        float_precision fp, fq;

        // 1) Define the recursive lambda as a std::function so it can call itself
        std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)>
                binarysplittingZuniga;

        binarysplittingZuniga =
                [&](uintmax_t a, uintmax_t b, int_precision& pRef, int_precision& qRef,
int_precision& rRef)
        {
                if (a + 1 == b)
                {
                        const uintmax_t b4 = 4 * b;
                        const uintmax_t b8= 8* b;
                        const uintmax_t b24 = 24 * b;
                        const int_precision bb(b);

                        qRef = int_precision(531'441ull) *
                                int_precision((b24 - 1) * (b24 - 5)) *
                                int_precision((b24 - 7) * (b24 - 11)) *
                                int_precision((b24 - 13) * (b24 - 17)) *
                                int_precision((b24 - 19) * (b24 - 23));
                        rRef = int_precision(128 * b * (2 * b - 1)) *
                                int_precision((b4 - 1) * (b4 - 3)) *
                                int_precision((b8 - 1) * (b8 - 3)) *
                                int_precision((b8 - 5) * (b8 - 7));
                        pRef = ((((((int_precision(2659024319283200ull) * bb
                                        - int_precision(8'084'203'069'505'536ull) ) * bb
                                        + int_precision(9949321961406464ull)) * bb
                                        - int_precision(6349069378355200ull)) * bb
                                        + int_precision(2229033948170240ull)) * bb
                                        - int_precision(417891652274944ull)) * bb
                                        + int_precision(364282293764l6ull) ) * bb
                                        - int_precision(935651486700ull);
                        return;
                }

                // If you had the special (a + 2 == b) case, you can put it here

                // 2) Recurse: split [a..b] into [a..mid] and [mid..b]
                const uintmax_t mid = (a + b) / 2;
                int_precision pp, qq, rr;

                binarysplittingZuniga(a, mid, pRef, qRef, rRef);
                binarysplittingZuniga(mid, b, pp, qq, rr);

                // 3) Merge results (combine partial results)
                pRef = pRef * qq + pp * rRef;
                qRef *= qq;
                rRef *= rr;
        };

        // 4) Actually use the recursive lambda
        binarysplittingZuniga(0, kmax, p, q, r);

        // 5) Convert to float_precision and finish
        fp.precision(workprec);
        fq.precision(workprec);

        // Move q into a float_precision
        fp = float_precision(p, workprec);
        fq = float_precision(q, workprec);
        fp /= fq;
        fp.precision(precision);

        return fp;
}
```

Performance of log(3)

Having introduced both the more straightforward Zuniga I method and the more involved I - G4 variant, we now compare their practical performance alongside more general alternatives. As before, we measure the time required for each approach to achieve a specific precision, focusing on input sizes ranging from 10 to 1,000,000 digits.

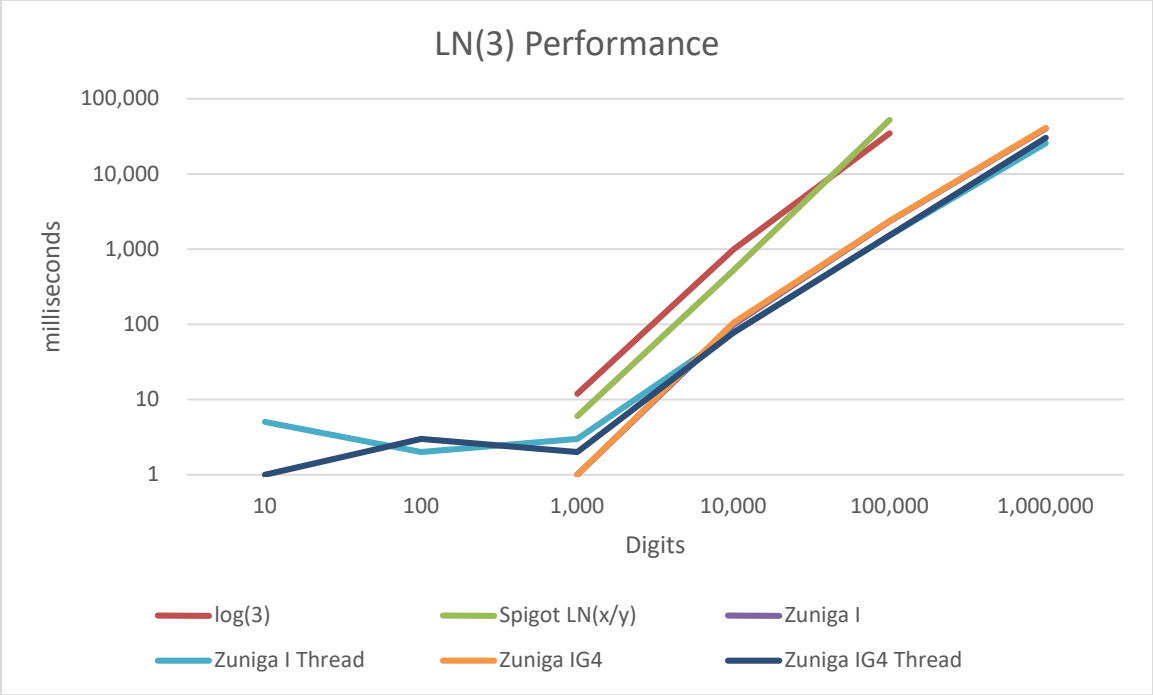


Figure 3. Log(3) performance chart.

As the performance chart above shows, the Zuniga methods are an order of magnitude faster than the others.

LN(3) Performance Result. All times are in msec.						
Digits	10	100	1,000	10,000	100,000	1,000,000
log(2)	-	-	12	986	34,829	
Spigot LN(x/y)	-	-	6	523	52,497	
Zuniga I	-	-	1	99	2,339	40,190
Zuniga I Thread	5	2	3	78	1,521	25,669
Zuniga I-G4	-	-	1	105	2,352	41,192
Zuniga I-G4 Thread	1	3	2	77	1,514	30,315

Table 11. Log(3) performance for different methods.

As the table above shows, the scalability of the Zuniga method is much better than that of any other method. The two Zuniga methods are similar, with a slight edge to the Zuniga I method over the Zuniga I-G4.

Recommendation for the log(3) constant

1) The traditional log(3) and the Spigot Lnxy are significantly slower than the Zuniga methods and are therefore not recommended.
2) The two Zuniga methods yield the same performance. However, the simplex method is easier to implement and, therefore, is recommended.
3) Add Threading to increase the performance of each method, typically yielding a 30% performance gain.

The constant Log(5)

Besides the previous general method for log(5), not many other methods can be helpful. Expect the relatively new method by Zuniga in 2023.

In comparison to log(2) and log(3), there has historically been less focus on log(5) within the numerical analysis community, likely because log(10) is more directly tied to decimal systems. The result is fewer specialized formulas specifically targeting log(5). Nevertheless, as we will see, recent work by Zuniga has yielded methods that converge quickly and are well-suited for binary splitting.

For Log(5) we will examined:

- The general Log(x) algorithm is described in the previous section.
- The spigot-like method lnxy_64.
- Zuniga-I binary splitting method.

Each method has distinct complexity, memory usage, and convergence speed trade-offs. As with log(2) and log(3), we aim for a high-precision approach that can scale to millions of digits while minimizing runtime and code complexity. Below, we assess the classical log(x) function, the spigot-like approach, and two Zuniga options, clarifying where each shines.
The spigot-like method will perform slower since it is less efficient when the argument is between one and two.
Zuniga also has another method that requires more computational work per split but also less than the one below. Since that method also has a computational cost of ~ 1.23, the speed will be equivalent. I have always strongly preferred using the most straightforward function when the two methods have the same computational cost.

Zuniga binary splitting method for log(5)

While the spigot-like approach can handle log(5), it suffers from slow convergence once x exceeds 2. This limitation becomes apparent as precision requirements grow. Zuniga's method, by contrast, relies on a carefully constructed series that significantly boosts

convergence, particularly at higher precisions. We now present the details of that binary splitting approach.
Zuniga has made several methods for computing the constant log(5). From [9], we get, after changing the more efficient notation using the Pochhammer symbol into the series below, suitable for implementation using binary splitting.

$$log(5) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}(728n-124)}{2n(2n-1)} \prod_{k=1}^{n} \frac{2k(2k-1)}{75(6k-1)(6k-5)} \quad (32)$$

Algorithm: Zuniga Binary splitting method for Log(5)

$set\ m = \frac{a+b}{2}\ integer\ division$
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)^n+(728b-124)
Q(b-1,b)=75(6b-1)(6b-5)
R(b-1,b)=2b(2b-1)

Zuniga Log(5) method.

And then

$$\log(5) = \frac{P(0,n)}{Q(0,n)} + O(675^{-n}) \quad (33)$$

These methods have a linearly convergent cost of ~1.23. It is straightforward to implement but less efficient than the second of Zuniga's methods.

For *n* terms, the error is $O(675^{-n})$ And for a given precision, *P,* you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(675)} \right\rceil \quad (34)$$

The above equation tells us you get approximately. 2.8 decimal digits accuracy per split.

Source Zuniga

```
// Now define computeLn5Zuniga with an embedded lambda
//
static float_precision computeLn5ZunigaI(const uintmax_t precision)
{
        // Decide how many terms, etc.
        const intmax_t kmax = intmax_t(std::ceil(precision * std::log(10) / std::log(675)));
        const size_t   workprec = size_t(std::ceil(precision + 1 + 1 * std::log(kmax)));

        // Prepare accumulators
        int_precision p, q, r;
        float_precision fp, fq;

        // 1) Define the recursive lambda as a std::function so it can call itself
```

```
        std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)>
                binarysplittingZuniga;

        binarysplittingZuniga =
                [&](uintmax_t a, uintmax_t b, int_precision& pRef, int_precision& qRef,
int_precision& rRef)
        {
                if (a + 1 == b)
                {
                        const uintmax_t b6 = 6 * b;

                        qRef = int_precision(75 * (b6 - 1) * (b6 - 5));
                        rRef = int_precision(2 * b * (2 * b - 1));
                        pRef = int_precision(728 * b - 124);
                        if(b+1 &0x1) // Odd
                                pRef.change_sign();
                        return;
                }

                // If you had the special (a + 2 == b) case, you can put it here

                // 2) Recurse: split [a..b] into [a..mid] and [mid..b]
                const uintmax_t mid = (a + b) / 2;
                int_precision pp, qq, rr;

                binarysplittingZuniga(a, mid, pRef, qRef, rRef);
                binarysplittingZuniga(mid, b, pp, qq, rr);

                // 3) Merge results (combine partial results)
                pRef = pRef * qq + pp * rRef;
                qRef *= qq;
                rRef *= rr;
        };

        // 4) Actually use the recursive lambda
        binarysplittingZuniga(0, kmax, p, q, r);

        // 5) Convert to float_precision and finish
        fp.precision(workprec);
        fq.precision(workprec);

        // Move q into a float_precision
        fp = float_precision(p, workprec);
        fq = float_precision(q, workprec);
        fp /= fq;
        fp.precision(precision);

        return fp;
}
```

In parallel with the described method, Zuniga explored an alternate series that requires fewer terms overall by packing more factors into each iteration. However, this introduces a heavier computational load per step. Below, we summarize how this second method compares in terms of complexity and speed.

Zuniga has also published another similar method for log(5). It has the same computational cost as the previous method. It is a more complex formula but requires fewer splits. However, the computational cost and performance are the same for both methods.

Performance of log(5)

Having outlined the primary approaches to log(5), we turn our attention to real-world performance. As with log(2) and log(3), we measure wall-clock times for each method at various digit precisions, ensuring consistent big-integer operations and hardware settings. The table below illustrates the dramatic outperformance of the specialized Zuniga approach over more generic routines.

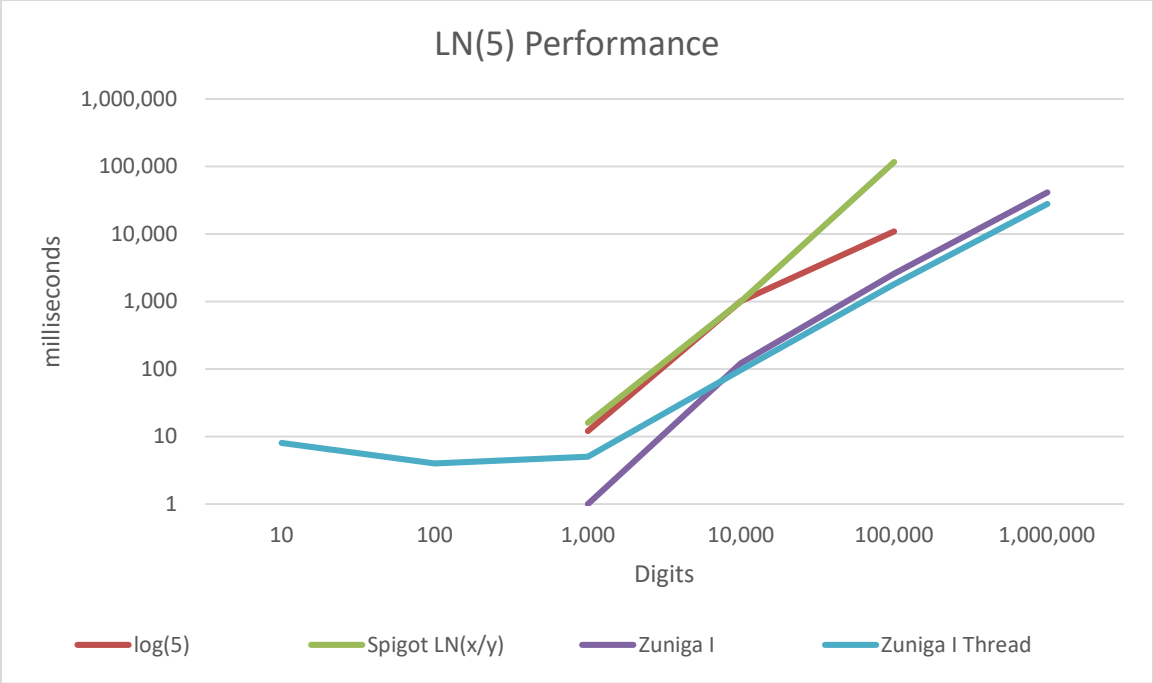


Figure 4. Log(5) performance chart.

The Zuniga binary splitting method outperforms the classic log(10) and the Spigot-like lnxy_64 method.

LN(5) Performance Result. All times are in msec						
Digits	10	100	1,000	10,000	100,000	1,000,000
log(5)	-	-	12	1,014	11,002	
Spigot LN(x/y)	-	-	16	1,001	117,122	
Zuniga I	-	-	1	122	2,572	41,689
Zuniga I Thread	8	4	5	96	1,806	27,929

Table 12. Log(5) performance for different methods.

Recommendation for log(5)

1) The performance chart clearly indicates that Zuniga's binary splitting method is significantly faster than any other method and is recommended.
2) Adding a threaded version of the method increases the performance by approximately 30%.

The constant Log(10)

There are not that many contenders to the log(10) constant. To my knowledge, there is no binary splitting version available. One reason is that specialized algorithms for log(10) can be easily derived from the more thoroughly studied constants 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(2·5)=log(2)+log(5). This leads us to examine these three versions.

For Log(10) we will examined:

- The general 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.

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 at 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(8·10/8)=log(8)+log(10/8)=3·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 LN(x/y) graph.

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.

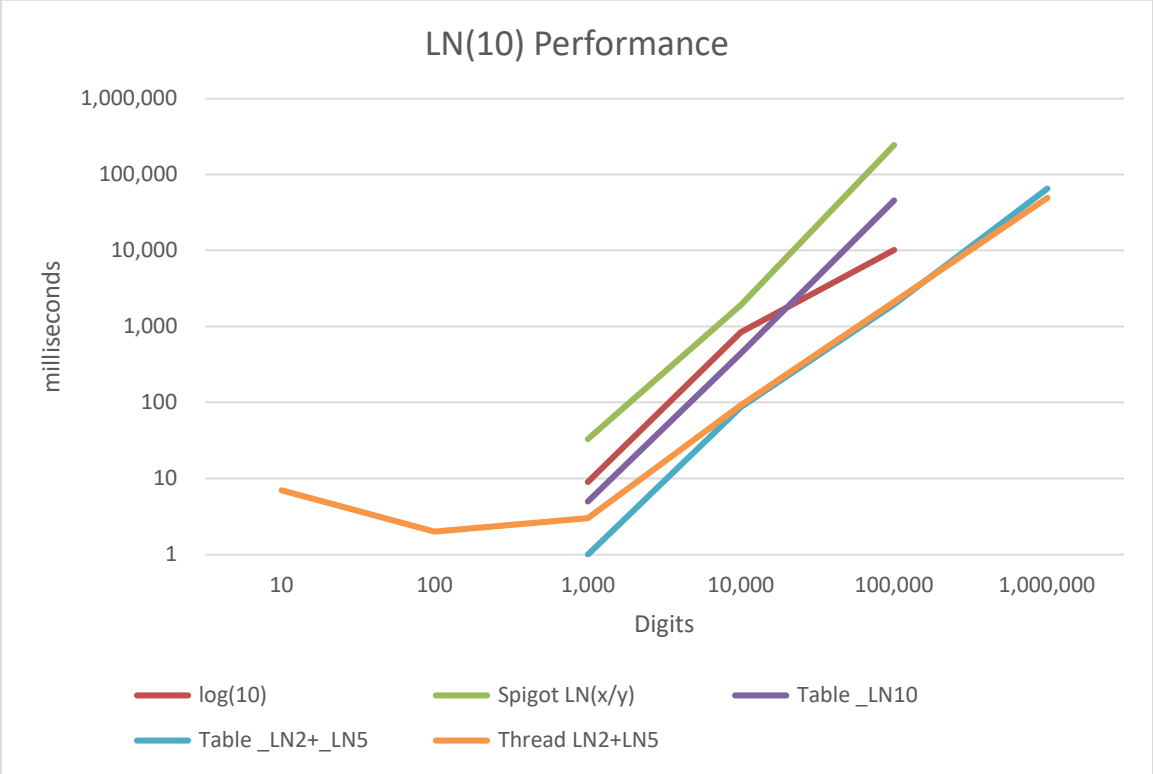


Figure 5. Log(10) performance chart.

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 13. 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.
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 precisions.

Conclusion for log() constants

There is no doubt that, for the time being, using binary splitting methods for log(2), log(3), log(5), and log(10) yields a significant performance gain over any other methods.

.

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) Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants (hvks.com)
4) HVE Fast Exp() calculation for arbitrary precision; Fast Exp() calculation for arbitrary precision (hvks.com)
5) The Yacas book of algorithms, Version 1.3.3, April 1 2013 by the Yacas team
6) 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
7) The Math behind arbitrary precision for integer and floating-point arithmetic. The Math behind arbitrary precision (hvks.com)
8) Alexander Yee, Binary Splitting Recursion Library. Binary Splitting Recursion Library
9) Github.com y-cruncher-Formulas/Official Formulas at master · Mysticial/y-cruncher-Formulas · GitHub

Appendix

The table below summarizes each log(x) approach, including its convergence rate and complexity, its main advantages and drawbacks, and the precision range in which it is the preferred choice. All the iterative methods are assumed to use the top-down precision ladder described earlier; without it, every one of them is roughly twice as slow, and the ranges below do not apply.

Method	Convergence & Complexity	Pros	Cons	Recommended Precision Range
Taylor series + argument reduction	Linear, with heavy argument reduction. The number of terms grows with the precision.	- Easy to implement - No dependency on exp(), π or ln(2) - Straightforward, minimal overhead	- Slow at large precisions - One division per term - Requires repeated square roots for argument reduction	Up to ~200–300 digits
Newton's method	Quadratic (doubling digits per step). About 1.5 full-precision exp() evaluations in total with the ladder.	- Simple iteration, self-correcting - y·exp(−x) avoids the division and is ~15% faster than y/exp(x)	- One exp() evaluation per level - Superseded by Chebyshev at every precision, at no extra implementation cost	Not recommended; retained as the baseline of the family
Newton, division free (Brent)	Quadratic. About 2.5 full-precision exp() evaluations, because of the extra reduction stage.	- Removes every division from the loop - Free argument reduction when x is already close to 1	- The naive propagated form loses self-correction and silently caps the accuracy - The corrected form costs one extra full-precision exp()	Not recommended, except possibly for x in [0.5, 1.5)
Halley's method	Cubic (tripling digits per step). About 1.21 full-precision exp() evaluations with the ladder.	- Fewer levels than Newton - Best error constant of the cubic methods ($d^3/12$)	- One exp() and one division per level - The division costs it ~25% against Chebyshev below ~50,000 digits	Only where a division is cheaper than a squaring; otherwise use Chebyshev
Chebyshev's method	Cubic (tripling digits per step). About 1.21 full-precision exp() evaluations with the ladder.	- Fastest of the exp()-based methods at every precision above 10 digits - Division free: one exp() and one squaring per level - Two-line change from Newton	- Still one exp() per level, so it inherits the growth rate of exp() - Cannot approach AGM at high precision	The preferred fallback at any precision when AGM is unavailable
AGM method	O(M(P)·log P). The iteration count grows only logarithmically with the precision.	- Fastest known approach above a few hundred digits - Easy to parallelize (π, ln(2) and the AGM in separate threads)	- Relatively complex to implement - Requires π and ln(2) to the same precision - Cancellation in the final subtraction needs guard digits	Above ~300 digits
AGM method, multi-threaded	As AGM, with π, ln(2) and the AGM computed concurrently.	- 20–30% faster than single-threaded AGM at high precision	- About 1 ms of thread overhead, which dominates at small precisions	Above ~2,000–3,000 digits
Binary splitting (log constants)	Quasi-linear for the specific constant, via specialized rapidly	- Ideal for ln(2), ln(3), ln(5), ln(10)	- Only applies to "nice" constants - Separate code for each constant	For those constants, at any precision

	converging expansions.	- Far faster than the general algorithm for these values		

Table . Summary of the log(x) methods.

Measured performance

All timings are in milliseconds, measured on the author's machine with each case averaged over a 0.5 second run for digits less than 10,000. Every iterative method uses the top-down ladder and the exponent-based stopping test.

Method\Digits	10	100	1,000	10,000	100,000
Hybrid (Taylor/AGM)	0.06	0.12	2.34	205	4,582
Taylor	0.06	0.12	4.18	1,145	461,310
Newton	0.09	0.23	2.81	454	100,840
Newton Division free	0.07	0.20	3.38	710	175,472
Chebyshev	0.07	0.17	2.19	328	80,832
Halley	0.07	0.18	3.14	440	82,082
AGM	0.07	0.15	2.30	200	4,403
AGM Thread	1.14	1.17	2.73	158	3,363

Table . Log(x) performance in milliseconds as a function of precision.

Four observations about the measurements themselves are worth recording, since they affect how the table should be read.

Below about 300 digits, the differences between Taylor, Chebyshev, and AGM are within a few percent and close to the resolution of the measurement. Any of the three is a defensible choice there, and the crossover between Taylor and AGM should be regarded as lying somewhere between 150 and 400 digits rather than at a sharp value.

The methods that depend heavily on sqrt(), AGM, which calls it once per iteration, and the Taylor series, whose argument reduction is a chain of square roots, both show a jump of 60–80% between 900 and 1,000 digits, where the exp()-based methods grow by only 13–14%. This points to a threshold inside sqrt() rather than to anything in the logarithm algorithms.

Similarly, all the exp()-based methods jump by a factor of 2.5–3 between 4,000 and 5,000 digits, while AGM grows by 1.7. That is a threshold inside exp(), and it affects every routine in the library that calls exp() in that range, not only the logarithm.

Finally, the growth rates explain why the ranking at 100,000 digits is not a matter of tuning. For a tenfold increase in digits, from 10,000 to 100,000, AGM grows by a factor of 23 while the exp()-based methods grow by 190–250. Every method in that family costs

a fixed multiple of one full-precision exp() and therefore inherits its growth rate; no change of convergence order can close that gap.