Fast Logarithm function for Arbitrary Precision number.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a follow-up to a previous paper that describes the math behind arbitrary precision numbers, see [7]. First of all the original paper was written back in 2013 and quite a few things had happens since then, secondly, I have come across some other interesting methods to do the logarithm function calculation. The paper describes in more detail how to do ln(x) ($\log_e(x)$) calculation with arbitrary precision and outlines some traditional methods but also introduces an improved version that makes the calculation 10-20 times faster than the original method use in the author own arbitrary precision math packages.

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 improved the performance of the method. We will discuss the various method for calculating ln(x) and elaborate on the techniques like clever argument reduction and coefficient scaling to improve the performance of the method. Furthermore, we will analyze the Newton and Halley method for calculating ln(x) and finally go over the AGM method, which by far is the fastest method of them all.

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

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

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms

8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

Change Log

27 February 2023. Cleaning up the document and correcting minor issues.

Contents

The Arbitrary precision library

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

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

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

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

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

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

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

There is another method to manipulate the exponent of the variables. The method is called .exponent() and returns or sets the exponent as a power of two 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 the class method. .adjustExponent(). This method just adds the parameter to the internal variable that holds the exponent of the float_precision variable. E.g.

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

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

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

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

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

Internal format for float_precision variables

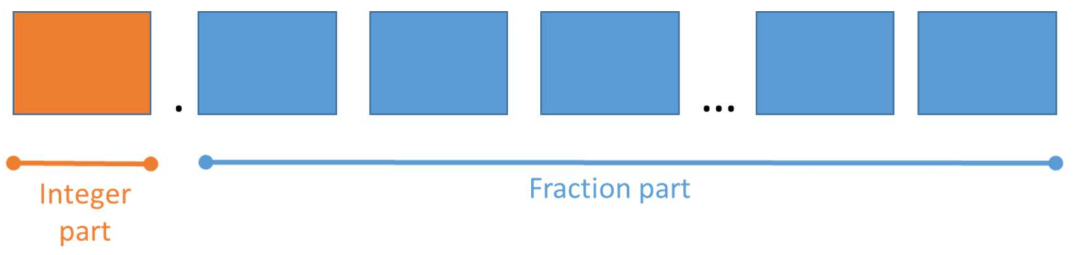
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

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

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

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

The Binary format mBinary



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

There are other internal class variables like the sign, exponent, precision, and rounding mode but these are not important to understand the code segments.

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)

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

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

The most common one for arbitrary precision libraries is the standard Taylor series expansion method but as will be shown this is not the preferred choice if you want performance. When we say log(x) with mean the natural logarithm is denoted as ln(x). For other bases, we will explicitly refer them to log10(x) or log2(x) to avoid any confusion.

Log(x) using the Taylor series

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

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

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. The series however converge slowly to ln(x) and is not suitable for arbitrary precision. Instead, most implementations use the inverse hyperbolic tangent function:

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

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

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**2**	
Taylor reductions=		**0**		
Terms	**zn**	**Term Sum**	**Ln(x)**	**Error**

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

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

Argument Reduction

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

We can use the identity:

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

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 $x \Rightarrow x^{\frac{1}{2^k}}$ and get closer to one we can then after the Taylor iterations multiply the result with 2^k to find the correct value of ln(x).

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

Example 2:
If using used the previous example 1 and reducing the argument twice from two to 1.1892… we only need 7 Taylor terms to get the same result as before, saving eight

Taylor terms but gaining two squaring and multiplication of 2^2 =4 at the end. However, overall huge savings since we have avoided eight time-consuming divisions in Taylor's terms.

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

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

Ln(x)		**Original**	**X Reduced**	
x=		**2**	**1.002711275**	
Taylor reductions=		**8**		
Terms	**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 a first glance. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result.

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

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

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:

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

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

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

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

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 it varies between 8-10 reductions.

The number of Taylor Terms.

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

Finding a reasonable reduction factor.

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

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

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

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

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

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

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

All measures are in milliseconds

Digits	100	1,000	10,000	1,000,000
Auto Red.	1.57	26	14,625	739,917
0 Red.	3.67	113	93,300	5719,74
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

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

Guard Digits

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

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

$2 + \lceil log10(precision) \rceil$ as extra guard digits as the 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 is not a lot of things you can do to improve the ln(x) algorithm. However, consider the Taylor series expansion of ln(x):

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

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

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

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

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

Moreover, group them:

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

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

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

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

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

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

Iterations Source for 5 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
       }
```

Log(x) using the Newton method

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

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

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. The Newton method has a quadratic convergence rate doubling the number of correct digits for each iteration. For precision, less than 10,000 digits the Taylor series from the previous chapter is faster but above 10,000 digits the Newton method exceeds the performance of the Taylor series. At 100,000 digits Newton's method is approximately 40% faster than the Taylor series.

Source ln_newton()

```
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);
       size_t digits, loopcnt = 1;
       double fx;
       float_precision r, x, y(a);

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

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

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

       // Now iterate using Netwon x=x(1+y-ln(x))  x=x-1+y/exp(x)
       for (digits = std::min((size_t)32, precision); ; digits =
std::min(precision, digits * 2), ++loopcnt)
              {
              // Increase precision by a factor of two for the working var. r & x.
              r.precision(digits+extra);
              x.precision(digits+extra);
              r = -c1 + y / exp(x);       // -c1+y/exp(x)
              if (digits == precision)
                     {// Reach final iteration step in regards to precision

                     r.precision(digits + 2);
                     x.precision(digits + 2);    // round to final precision
                     if (x + r == x )            // break if no improvement
                            break;
              }
              x += r;       // x=x-c1+y/exp(x)
              }

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

Log(x) using the Halley method

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

$$x_{n+1} = x_n + 2\frac{y-e^{x_n}}{y+e^{x_n}} \tag{9}$$

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

Source ln_halley()

```
float_precision lnEXP_halley_deep(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);
       size_t digits, loopcnt = 1;
       double fx;
       float_precision r, x, y(a);

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

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

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

       // Now iterate using Halley x=x-2(exp(x)-y)/(exp(x)+y)
       for (digits = std::min((size_t)48, precision); ; digits =
std::min(precision, digits * 3), ++loopcnt)
              {// Increase precision by factor two 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;
		if (digits == precision)
			{// Reach final iteration step in regards to precision
			r.precision(digits+2);
			x.precision(digits+2);
			if (x + r == x)		// break if no improvement
				break;
			}
		x += r;		// x=x+2(y-exp(x))/(exp(x)+y)
		}

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

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]:

$$\ln(x) = \pi \cdot x \frac{1+\frac{4}{x^2}(1-\frac{1}{\ln(x)})}{2\cdot AGM(x,4)} \tag{10}$$

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

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) \tag{11}$$

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

$$s = P\frac{\ln(10)}{2\cdot\ln(2)} + 1 - \frac{\ln(x)}{\ln(2)} \tag{12}$$

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

$$\ln(x) = \ln(2^s x) - s \cdot \ln(2) = \frac{\pi \cdot x^{s-2}}{2\cdot AGM(x^{s-2},1)} - s \cdot \ln(2)\text{, for } x > 1 \tag{13}$$

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

AGM Algorithm

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

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

Algorithm 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

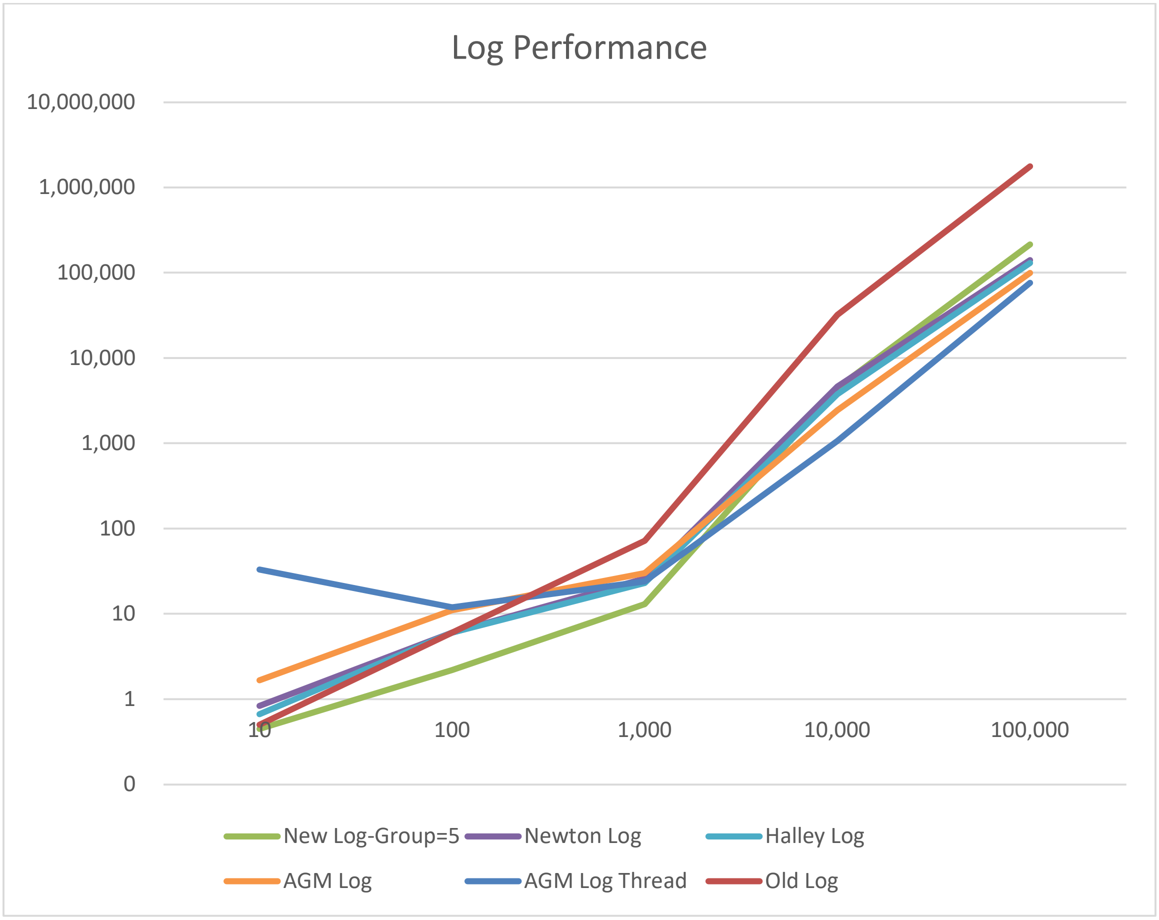


Figure 1

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

Log(x) using the AGM method and multiple threads

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

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

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

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 recommend:

- Ln(x) using Taylor series with argument reduction and coefficient scaling for precision up to approx. 4,000 digits.
- If the AGM method is available then use it above 4,000 digits.
- Moreover, use AGM in a multi-threaded version to increase performance.
- If the AGM method is not available then use either the Newton method or the better Halley method when precision exceeds 10,000 digits.
- Always use argument reduction to increase performance
- Coefficient scaling (or grouping of terms) can speed up calculation by a factor of two-three and is therefore recommended.

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)