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

By Henrik Vestermark (hve@hvks.com)

Abstract:

We are all used to the fast microprocessors available nowadays and computational speed of basic arithmetic, trigonometric or logarithms functions is done at lighting fast speed. However when building an arbitrary integer and floating point packages that can handle decimal in the range from a few to several millions digits it is s all back to the basic of math in order to build an arbitrary precision packages with reasonable speed.
This paper describes the underlying math behind this package.

Introduction:

Building an arbitrary software package that can handle all arithmetic for integers and floating point for any precision, it is down to the basic of simple math. This paper describe what formula and math that lies behind the arbitrary precision packages starting with arbitrary integer precision following with the floating point math. For the floating point math when a floating point number is broken down to its base component of <integer>, <fraction> and <exponent> it too use the integer precision math to do the calculation for floating point. After the basic floating point operators like +,-,*,/ we build on these functions to implement √, Logarithm and exponential functions continuing with Trigonometric and Hyperbolic functions. For each of these functions there is a describing on various optimizations technics to improve performance particular when needed precision in exceed 100 digits and going into the 100,000 digits precision and higher.

Contents

The Math behind arbitrary precision for integer and floating point arithmetic. 1
Abstract: 1
Introduction: 1
Integer Arithmetic 3
Addition: 3
Subtraction: 4
Multiplication: 4
Division & Remainder: 5
Integer power x^y: 7
Integer power x^y modulus z 7
Checking for prime numbers: 8
Performance of Arbitrary Integer precision: 8
Floating point arithmetic: 10
Addition: 11
Subtraction: 12
Multiplication: 13
Division: 13
Performance of Arbitrary Floating point precision: 15
Sqrt: 17
Elementary functions: 19
Logarithmic & Exponential functions: 20
$Log_e(x)$: 20
$Log_{10}(x)$: 23
Exp(x): 23
Pow(x,y): 25
Constants: $Log_e(2)$, $Log_e(10)$ & π 26
Trigonometric functions: 29
Sin(x): 29
Cos(x): 30
Tan(x): 31
ArcSin(x): 31
ArcCos(x): 34
ArcTan(x): 34
Hyperbolic functions: 36
Sinh(x): 36
Cosh(x): 36
Tanh(x): 37
ArcSinh(x): 38
ArcCosh(x): 38
ArcTanh(x): 38
Performance: 38

Integer Arithmetic

In integer arithmetic we use the notation i_n for an n digit integer number i where $n > 0$. In our description we assume the integer is in base 10 to simplify the description of the math behind the scenes. However in our actually implementation of the arbitrary precision packages we allow the integer number to be stored in any bases between base 2 and base 256 for better storage utilization. E.g. an integer number stored in base 256 requires 8 times less digits than if it was stored in base 2.
Also we denote *i[n]* as the most significant digit of *i* and *i[0]* the least significant digit of *i*. The integer i_n can also be described for any given base as:

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

For Base β=10 you get:

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

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

Addition:

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

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

$$\begin{aligned}
&for(carry = 0, i = 0..\max(n,m)) \\
&\quad c[i] = (a[i] + b[i] + carry)\%10 \\
&\quad carry = (a[i] + b[i] + carry)/10 \\
&if\,(carry != 0) \\
&\quad c[\max(n,m)+1] = carry;
\end{aligned}$$

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

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

Subtraction:

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

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

$$
\begin{aligned}
& carry = 10 \\
& for(i = 0..\max(n, m)) \\
& \quad carry = (9 + a[i] - b[i] + carry/10) \\
& \quad c[i] = carry \% 10; \\
& if\,(carry < 10\ ) \\
& \quad c\ is\ negative \\
& else \\
& \quad c\ is\ \ positive
\end{aligned}
$$

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

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

Multiplication:

Multiplication is also trivial

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

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

$$
\begin{aligned}
& for(carry = 0, i = 0..n) \\
& \quad c[i] = (a[i] * b_1 + carry)\%10 \\
& \quad carry = (a[i] * b_1 + carry) \\
& if\,(carry != 0) \\
& \quad c[n+1] = carry;
\end{aligned}
$$

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

$$
\begin{aligned}
& c_k = a_n * b[0] \\
& for(i = 1..m) \\
& \quad tmp = a_n * b[i] \\
& \quad c_k = c_k + tmp * 10^i;
\end{aligned}
$$

Notice that multiply the intermediate result with 10^i is easy since you just post fix the temp result with *i* number of zeros. The last algorithm can be enhanced in several ways. First of all you notice that the *tmp* result only can have the base β different number of outcomes. For the base 10 it is 10 different results. Instead of calculating it each time we can store the tmp results and then reused the temp result every time we encounter the same digit b[i].

$$
\begin{aligned}
& c_k = tmp_{b[0]} = a_n * b[0] \\
& for(i = 1..m) \\
& \quad if\,(tmp_{b[i]}\ does\ not\ exist) \\
& \quad\quad tmp_{b[i]} = a_n * b[i] \\
& \quad c_k = c_k + tmp_{b[i]} * 10^i;
\end{aligned}
$$

Although this improves the speed it doesn't beat the performance of using fast Fourier multiplication. It is beyond the scope of this paper to give a details explanation of the algorithm, but readers can reference [1]. However basically you convert the a_n and b_m via a series of Fourier transformation into waves then add them together and then inverse the result back via a Fourier transformation to the digital domain.

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

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

Division & Remainder:

There are several approaches you can take in order to calculate the division.
In it's simple form solving

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

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

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

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

If b_m is a single digit we do it by school book manner by dividing the single digit b_1
Into a_n starting at the most significant digit of a[n]. The result is the most significant digit of the quotient *c[k]*. Then add the remaining of that division into a's second most digit and repeat the process until all a_n digit has been divided. Now c_k is the quotient of the division and the last remaining digit is then d_j.

$$\begin{aligned} & for(rem = 0, i = 0..n) \\ & \quad c[i] = (10 * rem + a[i]) / b_1 \\ & \quad rem = (10 * rem + a[i]) \% b_1 \end{aligned}$$

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

$$\begin{aligned} & c_k = 0 \\ & while(a_n > b_m) \\ & \quad a_n = a_n - b_m \\ & \quad c_k = c_k + 1 \\ & d_j = a_n \end{aligned}$$

However we quickly find out that we will ran into problem when dividing a large number a_n that is several magnitude higher than b_m. E.g. let's assume that a_n is a number of 8 digits or a_8 magnitude is in the range of 10^8 and b_m is a two digits number of magnitude 10^2 then you will have to loop through the subtraction approx. 10^{8-2} or 10^6 times which is doable but time consuming. If instead we are dealing with the number a_n that is a 100 digit number then the looping will be in the order of 10^{98} Subtractions, even if we can do a subtraction in 10^{-6} seconds then it will still takes us 10^{+92} seconds or approx. 10^{83} years, which clearly will gets us nowhere.
Instead we use the fact that multiplication is much faster than division. Let's say that a_n is an *n* digit number and b_m is a *m* digit number and of course n>m then instead of subtracting b_m we try to subtract b_m*10^{n-m}. If b_m*10^{n-m} is less than a_n then we have replaced 10^{n-m} subtractions with one subtraction and one multiplications. This subtraction effectively ensure that the number a_n now is one digit less than *n* and the next subtraction can then be with $10*10^{n-1-m}$ Repeating this process you get an approx. number of loops and operation of the multiplication and subtraction as ~ 2(n-m). We cannot always assume that b_m*10^{n-m} is less than a_n in which case we subtract b_m*10^{n-m-1} instead. This lead to a worst case scenario that is 10 times higher than the approximation we found before or ~ 20(n-m). So instead of 10^{92} seconds we have reduced the workload to

something around ~ 0.002 seconds. If a_n is a number with 1 million digits the time will be ~ 20 seconds or 1 billion digits it will be 20,000 seconds, fast but not fast enough. So we use our last trick we have for division and that is to use the iterative method for division used with floating point division. (see floating point division). We simply convert our integer number a_n and b_m to floating point numbers with *n* decimals, do the division using the iterative division method (describe later) and then convert the division result back to a integer. Instead of a linear scaling of operations with the number of digits we get a logarithm scaling of the number of operations which is of course much faster.

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

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

Integer power x^y:

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

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

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

Integer power x^y modulus z

Where x, y and z are all integers. Instead of first calculating x^y and then take the modulus z, which can lead to very high number of digits for the interim result and then carry out the modulus z to get the answer. E.g. $2^{1000000}$ is around a number with over 300,000digits and then take modulus of z e.g. 77 can be very time consuming since we first have to build a digit with over 300,000 digits and then apply the modulus operator that is very costly operation (see discussion under division and remaining).

To avoid large number we can incorporate the modulus operator into our calculation of x^y to avoid dealing with high number of digits in the interim result and we get the following algorithm:

$$
\begin{aligned}
&ipow_\mathrm{mod}ulu(x, y, z) \\
&\quad r = 1 \\
&\quad x = x\% z \\
&\quad while(y > 0) \\
&\qquad if\,(y \ is \ odd) \\
&\qquad\quad r = r * x \\
&\qquad\quad r = r\% z \\
&\qquad x = x * x \\
&\qquad x = x\% z \\
&\qquad y = y / 2 \\
&\quad return \ r
\end{aligned}
$$

Checking for prime numbers:

Checking for prime numbers is heavily used in encryption algorithm. A prime is a number that are only divisible by 1 and the number itself. So we basically check a number by starting from 2 and try to divide it up in the number that we want to check as a prime candidate. If the division result in a remaining of zero then that number is divisible with our prime candidate and is therefore not a prime. Now to test for all numbers is again very time consuming so we take advantages that we only need to test all candidates up to $\sqrt{prime\ candidate}$ and below that number we only need to test 8 numbers for every 30 numbers.

All integers are of the form 30k + i for i = 0, 1, 2,...,29 and k an integer from 0 and up. However, 2 divides 0, 2, 4,...,28 and 3 divides 0, 3, 6,...,27 and 5 divides 0, 5, 10,...,25. So all prime numbers are of the form 30k + i for i = 1, 7, 11, 13, 17, 19, 23, 29 (i.e. for i < 30 such that gcd(i,30) = 1).

Note that if *i* and 30 are not coprime, then *30k + i* is divisible by a prime divisor of 30, namely 2, 3 or 5, and is therefore not a prime.

Performance of Arbitrary Integer precision:

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

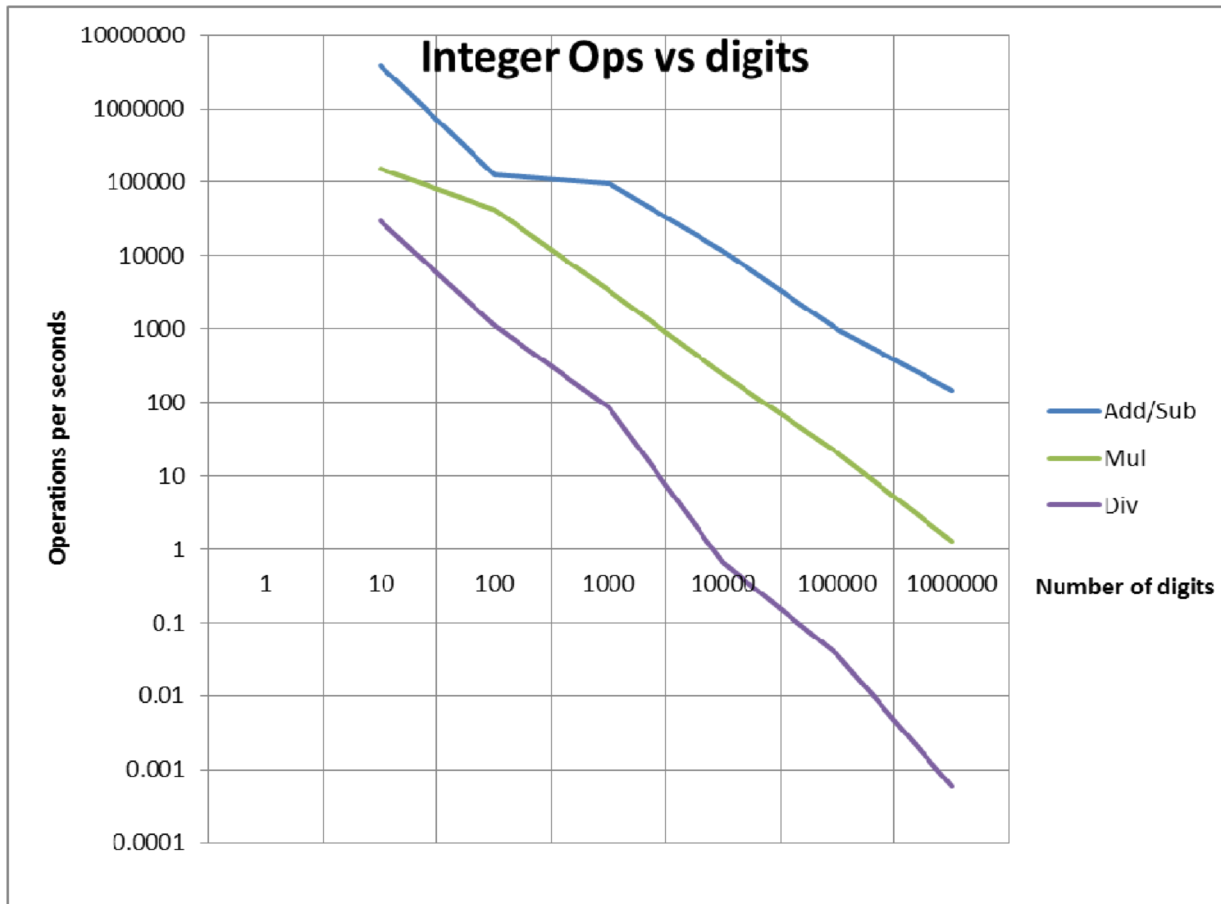


Figure 1. Integer Operation per seconds vs. number of digits

Not surpisingly the performance ratio between addition/multiplication and division increase with higher number of digits in the arithmetic operations.

Operations\digits in operand	**10**	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Addition/Subtraction	131	115	1,127	18,197	27,972	248,276
Multiplication	5	38	40	380	583	2,241
Division	1	1	1	1	1	1

The performance graph shows that it is vise to avoid the division at nearly all cost.

The performance was measure on a Windows 7, 64 bit system, Intel I7-2860QM processor with a clock speed of 2.5GHz.

Floating point arithmetic:

In arbitrary precision our floating point consist of a single integer digit i_1 follow by an arbitrary number of fraction digits (f_n) multiply to any power of the base (β) with a p digit exponent number e_p to create any arbitrary precision floating point number:

$$Floating\ \ po\text{int}\ number = i_1.f_n\beta^{e_p}$$

So it consist a single digit in front of the fraction sign and then n digits of fractions followed by the power of the base number. For base 10 you get

$$Floating\ \ po\text{int}\ number = i_1.f_n10^{e_p}$$

$$Example: 1.234 * 10^2$$
$$i_1 = 1$$
$$f_3 = 234$$
$$10^{e_1} = 2$$

For shorthand we write this example as 1.234E2 where E designates the Exponent value of 2.

In our description we assume the integer is in base 10 to simplify the description of the math behind the scenes. However in our actually implementation of the arbitrary precision packages we allow the floating point number to be stored in bases of 2, 10, 16 and 256 for better storage utilization. E.g. a floating point number stored in base 256 requires 8 times less digits than if it was stored in base 2. Furthermore the operations will also be faster in a higher base since we are dealing with fewer digits.

When the number has 1 digit in front of the fraction sign e.g. i_1 >=1 & <=9 the number is said to be normalized. If i_1 is outside this range the number is un-normalized. As a result of any arithmetic operations the intermediate result can be an un-normalized number however our arbitrary precision packages always guaranteed that the result of any arithmetic operation will always be returned as a normalized number.

Since our floating point number can be of different precision and we of course allow mixed precision for our arithmetic operators it is important to understand the precision. In any assignment statement (C language) =, +=, -=, *=, /= the resulting precision is always the precision of the variable on the left hand side of the operator and if necessary the expression on the right hand side is rounded accordingly. For binary operator like +,-,*,/

The mixed precision is handle by always aligning the argument on both side of the operator to the argument that has the highest precision. E.g. in an expression of *a+b* where *a* is a 3 digit precision number and *b* is a 5 digit precision number the operations *a+b* is carried out using 5 digit precision.

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

Addition:

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

Now we can add the two numbers together:

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

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

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

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

Here is an example of how the process is working. The number *a* is a 4 digit precision number and *b* is a 2 digits precision number.

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

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

Subtraction:

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

Subtraction	A	-	B	=	C
Number	1234.56		-2.4		
Precision	4		2		
Sign	1		-1		
Mantissa	1.235		2.4		
Exponent	3		0		

Align to max exponents	1.235	0.0024	
Align precision	1.2350	0.0024	
Subtract the two numbers			1.2374
Round to precision			1.237
Reapply exponent			1.237E+3

Multiplication:

We used fast Fourier transformation (FFT) to perform multiplication by performing a fast Fourier transformation of each digit and then add them together and then perform a inverse Fourier transformation to get the result of the multiplication. See [1] Numeric Recipes for details of how to implement it. Before we call the FTT function we strip of the sign and exponent and then use the resulting sign as follows
.

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

And for the exponent we simple add them together since:
$(a*10^{e1})*(b*10^{e2}) = (a*b)*10^{e1+e2}$

Now since we compute the FFT using IEE754 arithmetic and we know for a 64 bit floating point we have 53 bits in the floating point mantissa we can then derived a bound for how large a number we can multiplied using only 64 bit FFT transformation.

In [1] they have estimated the number of digits to N satisfying the criteria:

$$2*\log_2(base) + \log_2(N) + k*\log_2(\log_2(N)) < 53$$

For Base=10 (decimal) and k=3(conservative), we can multiply number exceeding 2 Giga digits or 2,000,000,000 digits using this arbitrary packages. This is most likely way beyond what would be needed and most likely you will have run into other constrains in the operating system you are running.

Division:

To handle floating point division we rewrite the equation a/b to a* (1/b). Multiplication is a much faster operations that division so it makes sense to do it this way. Now we only need to figure out how to quickly calculation the inverse of b=(1/b). This was the same issue that faces many microprocessor or early RISC (Reduced Instruction Set CPU) that

did not have hardware support for the division operator. Instead they use a Newton iteration using the following algorithm for calculating 1/b:

$$x_n = x_{n-1}(2 - x_{n-1}V)$$
$$where\ V = b \quad \& \quad x_0 \approx \frac{1}{b}\ (initial\ \ guess)$$
$$and\ \ x_n\ \ converged\ towards\ \frac{1}{b}$$

This can also be found the following way by restating the problem of finding $\frac{1}{x} = y$ or $f(x) = \frac{1}{x} - y = 0$. Applying it to the Newton method you get:

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

Notice the algorithm only requires us to do 1 subtraction and 2 multiplications per iteration.
To see how this algorithm works let us find the inverse of 1.6 using an initially start guess of 0.1.

1/b

b=	1.6	
x_0=	0.1	
Iteration	x_n	Error
1	0.184	4.4E-01
2	0.31383	3.1E-01
3	0.470078	1.5E-01
4	0.586598	3.8E-02
5	0.622641	2.4E-03
6	0.624991	8.9E-06

7	0.625	1.3E-10
8	0.625	0.0E+00

After 8 iterations the difference between the iteration and the build in division operator is 0 and the result of 1/1.6 is 0.625.

Now the only question remain is how to find a suitable starting point for the iteration since we can't perform an initially division as the guess of 1/b. Instead we look at how our arbitrary precision number is build up.

$$\frac{1}{b} = \frac{1}{i_1.f_n 10^{e_p}} = \frac{1}{i_1.f_n} 10^{-e_p}$$

We can extract the exponent portion and find the inverse of $\frac{1}{i_1.f_n}$ and then multiply the result with 10^{-e_p} to find our inverse of 1/b. Since we do have support of hardware division using IEE754 standard (64bit floating point number) we can get our initially start guess with approximately 15 digits accuracy and then begin to iterate towards higher number of accuracy.

This method for division is very fast and has quadratic convergence meaning that for each iteration we will double the number of correct digits. To set this into perspective, assume we have a number with 100 digits we should then expect to do only 7 iterations to get our result if we start sufficient closed to the result. For 1,000 digits it will require approx. 10 iterations and for 1,000,000 digit precision approx. 20 iterations.

Performance of Arbitrary Floating point precision:

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

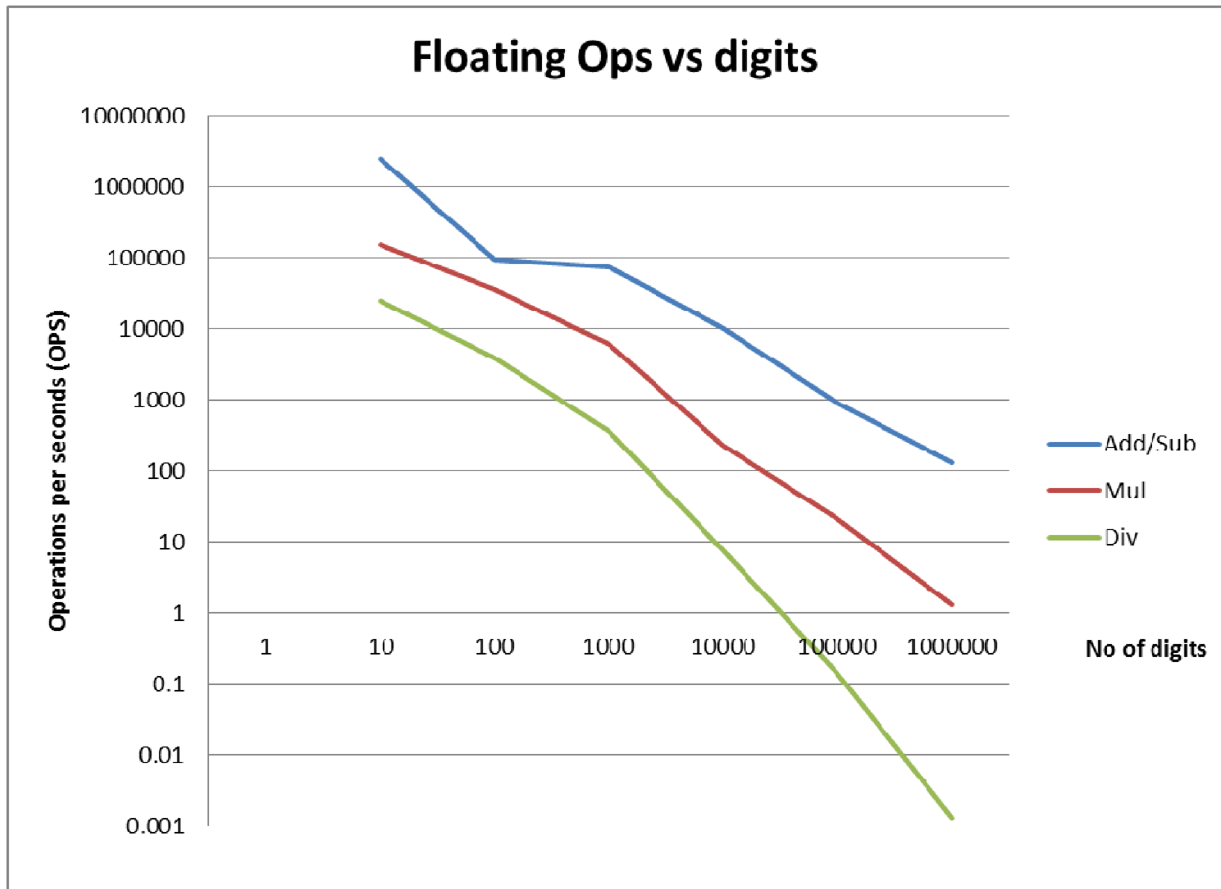


Figure 2. Floating point Operations per second versus No of Digits

Not surpisingly the performance ratio between addition/multiplication and division increase with higher number of digits in arithmetic operations.

Operations\digits in operand	10	100	1,000	10,000	100,000	1,000,000
Addition/Subtraction	101	23	211	1,334	6,543	102,308
Multiplication	6	9	17	31	150	1031
Division	1	1	1	1	1	1

The performance graph shows that it is vise to avoid the division at nearly all cost.

The performance was measure on a Windows 7, 64 bit system, Intel I7-2860QM processor with a clock speed of 2.5GHz.

Sqrt:

For sqrt(b) we again use a newton iteration algorithm to get our result. This time we iterate using this algorithm:

$$x_n = 0.5x_{n-1}(3 - V(x_{n-1})^2)$$

$$V = b \;\; Then \;\; x_n = \frac{1}{Sqrt(V)} \quad \& \quad Sqrt(V) = Vx_n$$

$$x_0 \approx \frac{1}{Sqrt(b)} \; (initial \;\; guess)$$

$$and \; Vx_n \; converged \; towards \; Sqrt(b)$$

Notice the algorithm only requires us to do 1 subtraction and 4 multiplications per iteration. Also as for Newton method we will have quadratic convergence meaning that for each iteration we will double the number of correct digits in our result.

This can also be found the following way by restating the problem of finding Sqrt(y) we instead try to find the reciprocal square root of y which is $\frac{1}{\sqrt{y}}$. Once it has been found we can find $\sqrt{y} = y\frac{1}{\sqrt{y}}$. Now to find the $\frac{1}{\sqrt{y}}$ we use the equation $\frac{1}{x^2} = y \Rightarrow \frac{1}{x^2} - y = 0$.
Using Newton method we get:

$$x_n = x_{n-1} - \frac{\frac{1}{x_{n-1}^2} - y}{(\frac{1}{x_{n-1}^2} - y)'} \Rightarrow$$

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

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

$$x_n = 0.5x_{n-1}(3 - x_{n-1}^2 y)$$

To see how this algorithm works let us find the Sqrt of 1.6 using an initially start guess of 1.6.

sqrt(b)

b=	1.6

x_0=	1.6		
Iteration	x_n	Sqrt(b)	Error
1	0.8768	1.40288	0.137968936
2	0.775948191	1.241517106	0.023393958
3	0.790166296	1.264266074	0.00064499
4	0.790569107	1.264910571	4.93246E-07
5	0.790569415	1.264911064	2.88436E-13
6	0.790569415	1.264911064	2.22045E-16
7	0.790569415	1.264911064	0

After 7 iterations the difference between the iteration and the build in Sqrt() operator is 0 and the result of Sqrt(1.6) is 1.264911064

As for the initial guess we again as for division extract the exponent out of the equation and then multiply the result with $10^{-\frac{e_p}{2}}$ after the iteration.

$$\frac{1}{Sqrt(b)} = \frac{1}{Sqrt(i_1.f_n 10^{e_p})} = \frac{1}{Sqrt(i_1.f_n)} 10^{-\frac{e_p}{2}}$$

This simplifies the iteration and allows us again to find a good initial guess using standard IEEE754 arithmetic with 15 significant decimals.

We could also instead of finding $\frac{1}{\sqrt{y}}$ go with the direct approach of finding $\sqrt{y}$ solving the equation $f(x)=x^2-y=0$. But that will change the Newton iteration to $x_n = 0.5(x_{n-1} + \frac{y}{x_{n-1}})$ Adding a dreadful division for each iteration.

Lets see how this plays out in an example:

Alternative sqrt(b)		
b=	1.6	
U0=	1.6	
Iteration	Sqrt(b)	Error
0	1.600000000	3.4E-01
1	1.300000000	3.5E-02
2	1.265384615	4.7E-04
3	1.264911153	8.9E-08
4	1.264911064	3.1E-15
5	1.264911064	0.0E+00

A little faster but comparable.

Introducing a division is a much more time consuming operator compare to just using and iteration based on addition/subtraction and multiplication. If we take the weighted relative performance between the various operators (Figure 2) used in one iteration step and apply it for the two different methods for calculating the square root we get:

Digits	**10**	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
(1) $x_n = 0.5x_{n-1}(3 - x_{n-1}^2 y)$	0.64	0.49	0.24	0.13	0.03	0.04
(2) $x_n = 0.5(x_{n-1} + \frac{y}{x_{n-1}})$	1.17	1.15	1.06	1.03	1.01	1
Ratio per iteration step	~1.7	~2.4	~4.4	~8	~37	~257

Where the ratios show that with increasing number of precision in the digits our choice of iteration (1) over (2) is approx. 8 times faster for 10,000 digits, and a whopping 257 times faster for 1,000,000 digits. A clear example of the benefit of avoiding division at nearly all cost.

Elementary functions:

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

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

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

Logarithmic & Exponential functions:

Log$_e$(x):

To find the base e logarithm to x we again resort to a Taylor series:

$$z = \frac{x-1}{x+1}$$

$$\log_e(z) = 2(z + \frac{z^3}{3} + \frac{z^5}{5} + \frac{z^7}{7} ...)$$

As usually in order to improve the effectiveness for the Taylor series we need to get the fraction z closer to 1. We do that by first taking out the exponent 10^p from z for later use. This leaves z (base 10) between 1.xxx…9.xxx. We used the identity that $\log_e(x^2) = 2\log_e(x)$ by performing k series of square root until z < 1.2. 1.2 is arbitrary chosen initially. We now have z between 1≤z≤1.2 and the Taylor series will quickly converge.
In order to compensate for the k times square root of the original z we multiply it now with 2^{k+1} which gives us the result of log(x) without the exponent of z. The effect of 10^p is now added into the result using the identity:

$$\log_e(x) = \log_e(z * 10^p) = \log_e(z) + \log_e(10^p) = \log_e(z) + p * \log_e(10)$$

Log$_e$(z) has just been calculated and log$_e$(10) is a constant (see how to calculate arbitrary precision constant) , so all we have to do is to multiply and add the numbers and we have found log$_e$(x).

Using x=1.2 we get after 8 Taylor series the result of log$_e$(1.2)=0.182322

log$_e$(x)

x=	1.2	
z=	0.09	
Iteration	x	Error
1	0.181818	5.0E-04
2	0.182319	2.5E-06
3	0.182322	1.5E-08
4	0.182322	9.5E-11
5	0.182322	6.4E-13
6	0.182322	4.5E-15
7	0.182322	2.8E-17
8	0.182322	0.0E+00

If we instead try to find $\log_e(4)$ without reducing the argument. It takes us 30 iterations to get close.

$\log_e(x)$		
x=	4	
z=	0.60	
Iteration	x	Error
1	1.2	1.9E-01
2	1.344	4.2E-02
3	1.375104	1.1E-02
4	1.383102	3.2E-03
5	1.385342	9.5E-04
6	1.386001	2.9E-04
7	1.386202	9.2E-05
8	1.386265	2.9E-05
9	1.386285	9.5E-06
10	1.386291	3.1E-06
11	1.386293	1.0E-06
12	1.386294	3.4E-07
13	1.386294	1.1E-07
14	1.386294	3.8E-08
15	1.386294	1.3E-08
16	1.386294	4.4E-09
17	1.386294	1.5E-09
18	1.386294	5.1E-10
19	1.386294	1.7E-10
20	1.386294	6.0E-11
21	1.386294	2.0E-11
22	1.386294	7.1E-12
23	1.386294	2.4E-12
24	1.386294	8.4E-13
25	1.386294	2.9E-13
26	1.386294	1.0E-13
27	1.386294	3.4E-14
28	1.386294	1.2E-14
29	1.386294	3.6E-15
30	1.386294	8.9E-16

The iteration is now much slower and demonstrates the effect of reducing the argument below 1.2 to improve the effectiveness of the Taylor series.

Can we further improve the effectiveness by significant reduce the argument well below 1.2? The answer is yes. To see the effect we have calculated the terms *n* for each sqrt()

reduction of the argument starting at 1.2 as a function of the precision *p* digits. As the table below shows if we reduced the argument by squaring it 8 times we reduced the Taylor terms from *n*=961 to *n*=291 for a 1000 digits calculation; for 100 digits we reduced *n* from 99 to 31 and for a 10 digits calculation we reduced the *n* from 13 to 5. Clearly for larger precision the benefits of reducing argument become more evident even when we consider the extra work of squaring the argument at the front-end and multiply it with a factor of 2 at the back-end.

p\x=	1.2	1.09544	1.04663	1.02305	1.0114	1.00571	1.00285	1.00142	1.00071
10	13	11	9	7	7	7	5	5	5
100	99	77	63	53	47	41	37	33	31
1000	961	747	611	515	447	395	353	319	291

Repeating the argument reduction we improve the performance with a factor 3 for a 1000 digits calculation and a factor 2 for a 100 digit calculations. Only small drawbacks is that we lose some precision by this repeated squaring but that can easily be overcome by just adding a couple of guard digits in the interim calculations of log().

So the effect of argument reduction increases with the number of digits in our calculation. 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 easily be estimated for log(). The nth -Taylor term for log() is given by

$$\frac{z^n}{n}; \quad Where\ z = \frac{x-1}{x+1}$$

And generally we stop the iteration when $\frac{z^n}{n} < 10^{-p}$; Where *p* is the precision. Now taking log on both side, rearrange and reduce we get:

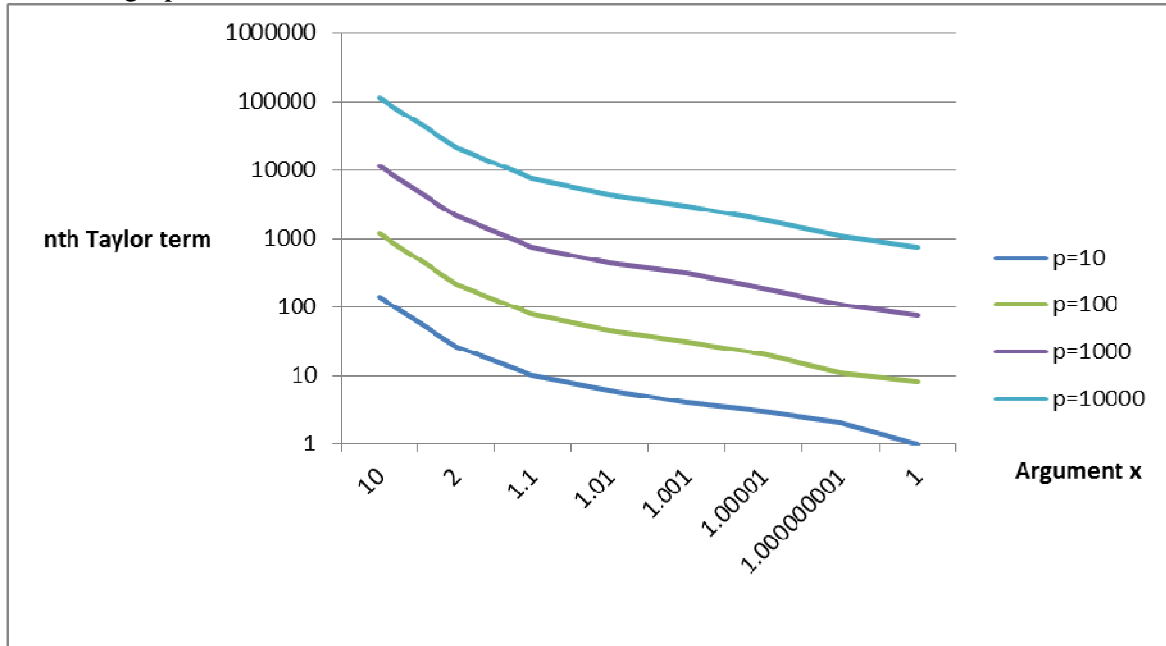
$$\log(\frac{z^n}{n}) = \log(10^{-p}) \Rightarrow$$
$$n\log(z) - \log(n) = -p\log(10) \Rightarrow$$
$$n\log(z) - \log(n) = -p$$

Log(n) can be ignore for large *p* precision so we get:

$$n\log(z) = -p \Rightarrow$$
$$n = \frac{-p}{\log(z)}$$

The picture below shows the nth Taylor terms for log() as a functions of argument (x-axis) and Precisions of the number. Roughly speaking you can reduced the needed n^{th} term of the Taylor series with a factor of 10 from an argument of 1.2 by continuing doing square

root until the argument has been reduced to $1+10^{-p}$ However that will required way to many square roots and the savings will not justified the extra work at the at front-end and back-end of the calculations. As the precision is increased we increase the number of square root reduction from the initial argument of 1.2 from 1 to 16 square roots for a 10,000 digit precisions.



$Log_{10}(x)$:

In order to calculate $Log_{10}(x)$ we use the equation: $Log_{10}(x)=Log_e(x)/Log_e(10)$ and $log_e(x)$ has been handle previously. $Log_e(10)$ is a constant see how we calculate constant to any arbitrary precision.

Exp(x):

To find the exponential for x we again resort to the Taylor series for e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \ldots$$

We eliminate $x < 0$ by using the identity: $e^{-x} = \frac{1}{e^x}$

Again to make the Taylor series converge more quickly we reduced the argument x≤0.5 using the identity: $e^x = e^{\left(\frac{x}{2}\right)^2}$ or more general we reduced the argument x for some *k*:

$\exp(x) = \exp(\frac{x}{2^k})^{2^k}$

Using x=0.5 we get after 14 Taylor series the result of Exp(0.5)=1.648721271

Exp(x)

x= 0.5

Iteration	x	Error
1	1.5	1.5E-01
2	1.625	2.4E-02
3	1.645833333	2.9E-03
4	1.6484375	2.8E-04
5	1.648697917	2.3E-05
6	1.648719618	1.7E-06
7	1.648721168	1.0E-07
8	1.648721265	5.7E-09
9	1.64872127	2.8E-10
10	1.648721271	1.3E-11
11	1.648721271	5.3E-13
12	1.648721271	2.1E-14
13	1.648721271	1.1E-15
14	1.648721271	4.4E-16

14 iterations to reach a result don't seem so bad at all. However when we are dealing with higher precisions e.g. 1000 digits, 10,000 or even 100,000 digits we suddenly has to take a lot more Taylor terms to find our result. In [5] Yacas book of algorithm they found a bound of the number of Taylor terms *n* needed as a function of the number of precision in digits *P*:

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

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

Now to see the effect of argument reduction for improving the Taylor series we have recalculated the amount of Taylor terms needed for various argument reductions from 1 to: $\frac{1}{2^k} \quad for\ k = 0,1,...,16$

No of Digits	$\frac{1}{2^0}$	$\frac{1}{2^1}$	$\frac{1}{2^2}$	$\frac{1}{2^4}$	$\frac{1}{2^6}$	$\frac{1}{2^8}$	$\frac{1}{2^{10}}$	$\frac{1}{2^{12}}$	$\frac{1}{2^{16}}$
10	15	12	10	7	6	5	4	3	3
100	71	61	53	42	35	30	26	23	18
1,000	451	405	366	307	264	231	205	184	153
10,000	3,250	2,992	2,770	2,409	2,128	1,904	1,721	1,569	1,333

With our standard argument reduction to 0.5 we need 10 Taylor terms for a 10 digits number, 61 Taylor terms for a 100 digit number, 405 for 1,000 digits and 2,992 for 10,000 digits number. Now by reducing the argument to $\frac{1}{2^{16}}$ or 0.00001526 we can reduced the number of Taylor terms needed from 2,992 to 1,333 for a 10,000 digit number. That is a huge saying considering that each extra Taylor terms need 1 addition, 2 multiplications and one inverse (division) per Taylor terms for a 10,000 digits. Multiplying this with the number of Taylor terms we save a total of 1,659 additions, 3,318 multiplications and 1,659 divisions. Now on the front end we get additional 16 extra multiplications with the argument reduction and on the backend we use an extra of 18 additions and 16 multiplications so the saving is a total of 1,643 additions, 3,286 multiplications and 1,659 divisions. However if the number of digits was only 10 we saved 9 addition, 18 multiplication and 9 division in the Taylor terms but lose 18 additions, 32 multiplications in the front and backend so the saving is insignificant.

However there is one drawback on this repeated argument reduction and that is loss of precision. If we compare exp(1) using no argument reduction and 8 times argument reductions we get a difference on the two numbers on +5E-9 for a 10 digit number and 8E-99 on a 100 digit number. With a 16 times reductions the error increase to -8.06E-7 (10digit number) and -1.748E-96 for a 100 digit number even when using Brent enhancement [6] to avoid loss of significant digits, so clearly we need to use more guard digits for the interim result when increasing the number of argument reductions. This is general not a problem since it is performance wise a lot cheaper to add a few extra guard digits in a e.g. 1000 digits precision number and save huges on reducing the number of needed Taylor terms.

Pow(x,y):

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

$$x^y = \exp(y * \log_e(x))$$

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

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

$Log_e(2)$ is calculate the same way as for $log_e(x)$ with the exception that we can determine a fix number of argument reduction using square root. Since the argument is 2 we know that squaring it twice will reduced the number below 1.19 and thereby making the Taylor series converge rapidly. After we have found the result to the given precision we then multiply the result with 2^3 to compensate for the argument reduction. However in our newer version we have applied the dynamic argument reduction to further speed up the calculation as discussed under **$Log_e(x)$.**

$Log_e(10)$ is calculated the same was as for $Log_e(2)$ except that we square 10 four times in the argument reduction to get the number below 1.16 and then use our Taylor series to quickly find the result of $Log_e(10)$ and in the back end we multiply with 2^5 instead of 2^3 However in our newer version we have applied the dynamic argument reduction to further speed up the calculation as discussed under **$Log_e(x)$.**

π is another interesting constant that there have been devoted much attention too for the last many thousand years. For a very good walk through of different algorithm to calculate π we recommend you read [4] Borwein, "PI and the AGM".
For our implementation we use one of the many algorithms for calculating π that can be found in [4].

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

The repeat for i=1,2,3,…

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

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

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

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

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

Borwein
π

Iteration	X	π	Y	Error
0	1.414214	3.41421356237309	1.189207	2.7E-01
1	1.015052	3.14260675394162	1.000673	1.0E-03
2	1.000028	3.14159266096604	1	7.4E-09
3	1	3.14159265358979	1	4.4E-16
4	1	3.14159265358979	1	4.4E-16

Another algorithm and equally as good as the Borwein algorithm is the Gauss-Legendre and the deviation also knows as Bent-Salamin.

The algorithm starts with the initial settings:

$$a_0 = 1$$

$$b_0 = \frac{1}{\sqrt{2}}$$

$$t_0 = 0.25$$

Then repeat for i=0,1,2,3,…

$$a_{i+1} = \frac{a_i + b_i}{2}$$
$$b_{i+1} = \sqrt{a_i b_i}$$
$$t_{i+1} = t_i - 2^i \left(a_i - a_{i+1)}\right)^2$$
$$\pi_{i+1} = \frac{(a_i + b_i)^2}{4t_i}$$

Gauss-Legendre

π

Iteration	a	b	t	π	Error
0	1	0.707107	0.25	2.91421356237309	2.3E-01
1	0.853553	0.840896	0.228553	3.14057925052217	1.0E-03
2	0.847225	0.847201	0.228473	3.14159264621354	7.4E-09
3	0.847213	0.847213	0.228473	3.14159265358979	8.9E-16
4	0.847213	0.847213	0.228473	3.14159265358979	8.9E-16

As for the Borwein algorithm we get quadratic convergence doubling the number of correct digits for each iteration. After 10 iterations we have more than 1000 digits and after 20 iterations more than 1 million digits. Borwein also showed a number of higher order convergence rate algorithm for finding π, with convergence rate for each iteration that multiplies number of correct digits with a factor of 3, 4, 5 and 9. However theses algorithm requires a lot more work to be done per iteration and is usually no worth implementing compare to the current one with quadratic convergence.

Trigonometric functions:

Sin(x):

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

$$Sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \ldots$$

However before we start the Taylor series we first reduce the argument x so it fall between 0 and 2π. Then we further reduce x so it is between 0..π using the identity $Sin(x+\pi) = -Sin(x)$ and finally we reduced the argument *k* number of times using the trisection identity: $Sin(3x) = 3Sin(x) - 4Sin(x)^3$ until x<0.5.
This argument reduction is done to reduce the number of Taylor iteration and to minimize the round off errors and calculation time.
After the Taylor series has converged we use the Trisection identity reverse *k* number of times to find our final result for Sin(x).

To see how this algorithm works let us find the Sin(0.7). After the 8th Taylor series the errors is zero and the result is ~ 0.644218.

Sin(x)

x= 0.7

Iteration	x	Error
1	0.7	5.6E-02
2	0.642833	1.4E-03
3	0.644234	1.6E-05
4	0.644218	1.1E-07
5	0.644218	4.9E-10
6	0.644218	1.6E-12
7	0.644218	3.7E-15
8	0.644218	0.0E+00

Again by adding a more aggressive argument reduction below 0.5 base on the actual precision we cut the number of Taylors terms needed to half which more than double the speed of Sin(x) for precision higher than 100 digits.

Cos(x):

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

$$Cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \ldots$$

However before we start the Taylor series we first reduce the argument x so it fall between 0 and 2π. Then we further reduce x so it is between 0..π using the identity $Cos(x) = Cos(2\pi - x)$ For x≥π ; and finally we reduced the argument *k* number of times using trisection identity: $Cos(3x) = -3Cos(x) + 4Cos(x)^3$ until x<0.5.
This argument reduction is done to reduce the number of Taylor iteration and to minimize the round off errors and calculation time.
After the Taylor series has converged we use the Trisection identity reverse *k* number of to find our final result for Cos(x).

To see how this algorithm works let us find the Cos(0.3). After the 7th Taylor series the errors is zero and the result is ~ 0.95534.

Cos(x)

x= 0.3

Iteration	x	Error
1	1	4.5E-02
2	0.955	3.4E-04
3	0.9553375	1.0E-06
4	0.955336488	1.6E-09
5	0.955336489	1.6E-12
6	0.955336489	1.1E-15
7	0.955336489	0.0E+00

Again by adding a more aggressive argument reduction below 0.5 base on the actual precision we cut the number of Taylors terms needed to half which more than double the speed of Sin(x) for precision higher than 100 digits.

Tan(x):

For Tan(x) we use the identity that $Tan(x) = \frac{Sin(x)}{\sqrt{1 - Sin(x)^2}}$

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

ArcSin(x):

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

Restating the problem as f(x)=Sin(y)-x=0 and applying the Newton method we get:

$$y_n = y_{n-1} - \frac{Sin(y_{n-1}) - x}{(Sin(y_{n-1}) - x)'} \Rightarrow$$
$$y_n = y_{n-1} - \frac{Sin(y_{n-1}) - x}{Cos(y_{n-1})}$$

And stop when $y_n = y_{n-1}$ for any given precision of the number. To speed up the iteration and to ensure convergence we repeatedly reduced the argument x below 0.5 by using the identity:

$$ArcSin(x) = 2 * ArcSin(\frac{x}{\sqrt{2}\sqrt{1 + \sqrt{1 - x^2}}})$$

Now the x argument will always per definition be -1≤x≥1, so we will only need a maximum of two argument reduction to get below 0.5.

Where v_0=x

$v_m = \frac{v_{m-1}}{\sqrt{2}\sqrt{1 + \sqrt{1 - v_{m-1}^2}}}$ until v_m is <0.5. Now we know the argument is less than 0.5 and we can start with an initially guess of ArcSin(v) using standard IEEE754. This gives us a starting guess for the Newton iteration with a least 15 significant digit and the Newton iteration will convergence pretty quickly. After we find the new y_n we will need to multiply the result with $y = y_n * 2^m$ to reverse the argument reduction we did prior to the Newton iteration.

To see how this algorithm works let us find the ArcSin(0.3). After only 3 iteration the errors is zero and the result is ~ 0.304693.

ArcSin(x)

x= 0.3

Iteration	x	Error
1	0.304689	3.4E-06
2	0.304693	1.8E-12
3	0.304693	0.0E+00

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

ArcSin(x)

x= 1

Iteration	x	Error
1	1.293408	2.8E-01
2	1.432998	1.4E-01
3	1.502007	6.9E-02
4	1.536415	3.4E-02
5	1.553607	1.7E-02
6	1.562202	8.6E-03
7	1.566499	4.3E-03
8	1.568648	2.1E-03
9	1.569722	1.1E-03
10	1.570259	5.4E-04
11	1.570528	2.7E-04
12	1.570662	1.3E-04
13	1.570729	6.7E-05
14	1.570763	3.4E-05
15	1.57078	1.7E-05
16	1.570788	8.4E-06
17	1.570792	4.2E-06
18	1.570794	2.1E-06
19	1.570795	1.0E-06
20	1.570796	5.2E-07
21	1.570796	2.6E-07
22	1.570796	1.3E-07
23	1.570796	6.6E-08
24	1.570796	3.2E-08
25	1.570796	1.5E-08

26	1.570796	7.5E-09

Even after 26 iterations we only get a decent result with an error margin of 7.5E-9, while with argument reduction we have the result with only 3 iterations. This example clearly demonstrates the benefit of using argument reduction prior to applying the Newton iterations.

Using Newton iteration gives the result in relative few iterations however still not very fast even if we replaced cos() by using the identity:

$$\cos(y_{n-1}) = \sqrt{1 - Sin^2(y_{n-1})} \Rightarrow$$
$$y_n = y_{n-1} - \frac{Sin(y_{n-1}) - x}{\sqrt{1 - Sin^2(y_{n-1})}}$$

So we only have to calculate Sin(y_{n-1}) once. Instead we can use the Taylor Series for ArcSin(x):

$$ArcSin(x) = x + \frac{x^3}{2*3} + \frac{3x^5}{2*4*5} + \frac{(3*5)x^7}{2*4*6*7} \dots$$

This gives us a more direct approach to ArcSin(x) and applied together with the dynamic argument reductions we see a speed up in the calculation with 50% for 10 digits numbers, 200% for 100 digits number and 500% for 1000 digits numbers.

Below is an example of using Taylor Series for calculating ArcSin(x). R is the residual of the Taylor terms that are added up in the x column.

ArcSin(x) Using Taylor series
x= 0.2

Iteration	R	x	Error
1	2.0000E-01	0.200000000	0.001358
2	1.3333E-03	0.201333333	2.46E-05
3	2.4000E-05	0.201357333	5.87E-07
4	5.7143E-07	0.201357905	1.6E-08
5	1.5556E-08	0.201357920	4.73E-10
6	4.5818E-10	0.201357921	1.47E-11
7	1.4215E-11	0.201357921	4.73E-13
8	4.5760E-13	0.201357921	1.57E-14
9	1.5141E-14	0.201357921	5.27E-16
10	5.1179E-16	0.201357921	2.78E-17

11	1.7596E-17	0.201357921	0

ArcCos(x):

To find ArcCos(x) we used the identity:

$$ArcCos(x) = \frac{\pi}{2} - ArcSin(x)$$

ArcTan(x):

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

$$ArcTan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \ldots$$

However before we start the Taylor series we first reduce the argument x so it is less than ≤0.5. We use the identity: $ArcTan(x) = 2ArcTan(\frac{x}{1+\sqrt{1+x^2}})$ k number of times until x≤0.5.
This argument reduction is done to reduce the number of Taylor steps and to minimize the round off errors and calculation time and of course to ensure that our Taylor series is stable.
After the Taylor series has converged we multiply the result with 2^k to find our final result for ArcTan(x). Now looking closer to the argument reduction you will notice that we never need more than two argument reduction. The first reduction will give us a max of ±1 Since:

$$\lim_{x\to\infty}(\frac{x}{1+\sqrt{1+x^2}}) = 1 \text{ or}$$

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

And the second reduction with x=±1 gives us x=±0.41.

To see how this algorithm works let us find the ArcTan(0.3). After 11th Taylor series the errors does not get lower and the result is ~ 0.291456794.

ArcTan(x)

x= 0.3

Iteration	x	Error
1	0.3	0.008543206
2	0.291	0.000456794
3	0.291486	2.92055E-05
4	0.291454757	2.03734E-06
5	0.291456944	1.49665E-07
6	0.291456783	1.13777E-08
7	0.291456795	8.86286E-10
8	0.291456794	7.0308E-11
9	0.291456794	5.65681E-12
10	0.291456794	4.60354E-13
11	0.291456794	9.58456E-13

Now assuming that we will continue our argument reduction to e.g. below ≤0.1 we can further reduced the number of Taylor steps taken. E.g.ArcTan(0.1) gives the result after only 6 Taylor steps.

ArcTan(x)

x= 0.1

Iteration	x	Error
1	0.1	3.3E-04
2	0.099666667	2.0E-06
3	0.099668667	1.4E-08
4	0.099668652	1.1E-10
5	0.099668652	9.0E-13
6	0.099668652	7.6E-15

So I guess it could be argued that the argument reduction should be used to further reduce the argument to below 0.1 to minimize the number of Taylor steps needed.

Hyperbolic functions:

Sinh(x):

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

$$Sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!}...$$

However before we start the Taylor series we first reduce the argument x so it is less than 0.5 using the identity: $Sinh(3x) = Sinh(x)(3 + 4Sinh^2(x))$ until x<0.5.
This argument reduction is done to reduce the number of Taylor iteration and to minimize the round off errors and calculation time.
After the Taylor series has converged we use the Trisection identity reverse *k* number of times to find our final result for Sinh(x).

To see how this algorithm works let us find the Sinh(0.7). After 8th Taylor series the errors is zero and the result is ~ 0.75858370184.

Sinh(x)
x= 0.7

Iteration	x	Error
1	0.700000000000	5.9E-02
2	0.757166666667	1.4E-03
3	0.758567250000	1.6E-05
4	0.758583590139	1.1E-07
5	0.758583701343	5.0E-10
6	0.758583701838	1.6E-12
7	0.758583701840	3.7E-15
8	0.758583701840	0.0E+00

Cosh(x):

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

$$Cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} ...$$

However before we start the Taylor series we first reduce the argument x, *k* number of times using trisection identity: $Cosh(3x) = Cosh(x)(4Cos^2(x) - 3)$ until x<0.5.
This argument reduction is done to reduce the number of Taylor iteration and to minimize the round off errors and calculation time.
After the Taylor series has converged we use the Trisection identity reverse *k* number of to find our final result for Cosh(x).

To see how this algorithm works let us find the Cosh(0.7). After 9th Taylor series the errors is not getting any lower and the result is ~ 1.25516900563094.

Cosh(x)

x= 0.7

Iteration	x	Error
1	1.00000000000000	2.6E-01
2	1.24500000000000	1.0E-02
3	1.25500416666667	1.6E-04
4	1.25516756805556	1.4E-06
5	1.25516899781771	7.8E-09
6	1.25516900560197	2.9E-11
7	1.25516900563086	7.8E-14
8	1.25516900563094	4.4E-16
9	1.25516900563094	2.2E-16

Tanh(x):

For Tanh(x) we use the identity that

$$Tanh(x) = \frac{Sinh(x)}{Cosh(x)} \quad =>$$

$$Tanh(x) = \frac{Exp(x) - Exp(-x)}{Exp(x) + Exp(-x)} \quad =>$$

$$Tanh(x) = \frac{Exp^{2x}(x) - 1}{Exp^{2x}(x) + 1}$$

ArcSinh(x):

ArcSinh(x) is giving by:

$$ArcSinh(x) = \log_e(x + \sqrt{x^2 + 1})$$

ArcCosh(x):

ArcCosh(x) is giving by:

$$ArcCosh(x) = \log_e(x + \sqrt{x^2 - 1})$$

ArcTanh(x):

ArcTanh(x) is giving by::

$$ArcTanh(x) = 0.5\log_e(\frac{1+x}{1-x})$$

Performance:

Below performance is based on a Intel I7 CPU at 2.50Mhz. Notice that addition/Subtraction is not surprisingly the fastest operations. Thanks to the FFT multiplication is also considered a very fast operation followed by Division and Square root as shown earlier. We notice in below graph that Sqrt() is considerable faster than Log() and Exp() .

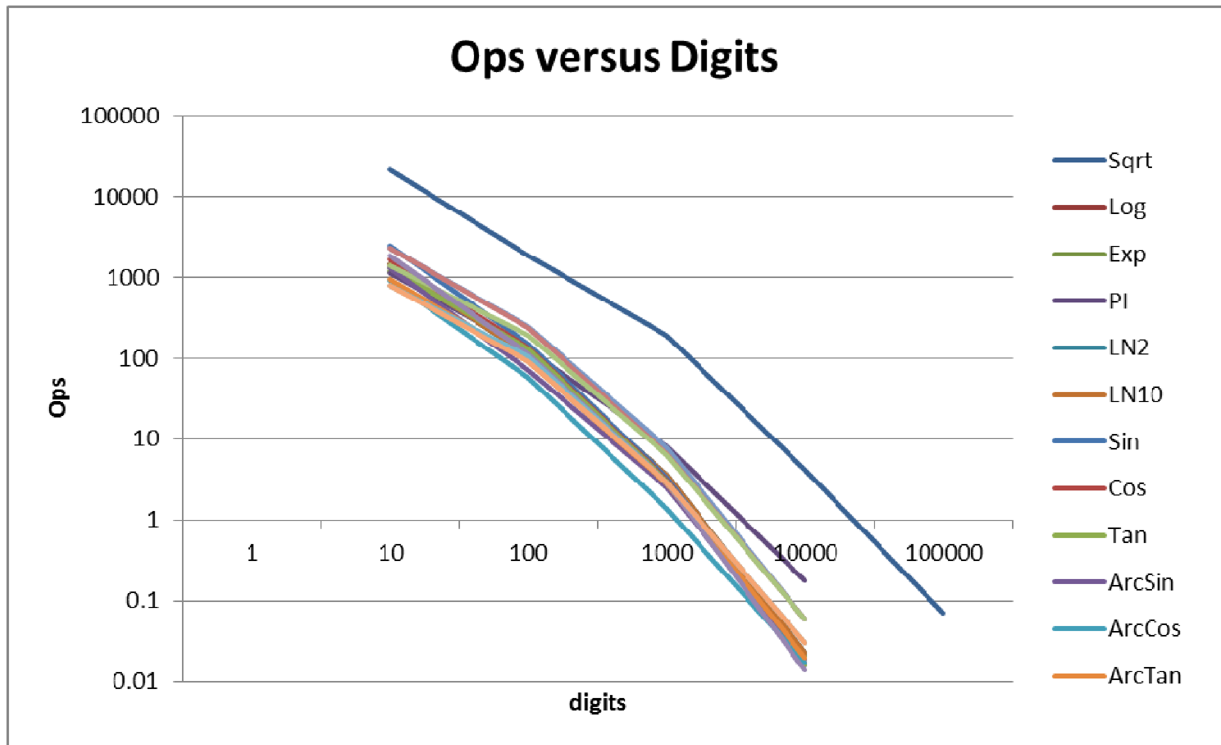


Figure 3Relative performance for functions

Reference

1. Numerical recipe's in C++, 3rd edition, Cambridge University press, New York, NY 2007
2. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood cliffs, NJ 1963
3. Methods of Computing square roots; May 17-2013; http://en.wikipedia.org/wiki/Methods_of_computing_square_roots
4. Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
5. The Yacas book of algorithm, 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