Implementing mathematical functions in Unum library

Sidong Feng (u6063820)

May 25, 2018

Course: COMP3740

Supervisor: Dr. Josh Milthorpe

Australian National University

I declare that except where otherwise indicated, this report is my own original work.

Sidong Feng
May 25, 2018

Abstract

Unum is a new number format, which is similar to floating point in IEEE. Computing with unums provides correct answers without rounding errors. Operations with unums are associative, expressions can be evaluated in parallel for an increase in performance. Crafting a sequence of operations to minimize round-off error, as with IEEE floats, is no longer necessary. Logarithm, exponential and power functions have many advantages in analytical world and much progress to scientific arithmetic has been made in theory. It needs more sophisticated mathematical algorithm to find unum than a floating-point arithmetic or transcendental functions. This report presents a methodology for the easy implementation of arithmetic function and transcendental functions. The approach presented is based on the mathematical inductions and arithmetical relations. It is capable of generating correct, highly precise unum results. Although the current implementations are several thousand times slower than IEEE, it seems reasonable. It caused by the concept of unum, conversion between three layers and bitwise notation. Meanwhile, massive use of GMP library leads to the speed down. We expect that in the future, an improvement on the arithmetic calculations in unum will be developed efficiency.

Contents

List of Figures

1. Introduction

Unum (the universal number), encompasses all standard floating point formats, as well as fixed point and exact integer arithmetic. Unums get more accurate answers than floating point arithmetic, yet use fewer bits in many cases, which saves memory, bandwidth, energy, and power. Unlike floating point numbers ("floats"), unums make no rounding errors, and cannot overflow or underflow. That may seem impossible for a finite number of bits, but that is what unums do. Unums are to floats what floats are to integers: A superset.

Unum arithmetic has more rigor than interval arithmetic, but uses far fewer bits of storage. A unum computing environment dynamically and automatically adjusts precision and dynamic range so programmers need not choose which precision to use; it is therefore easier and safer to use than floating point arithmetic. Unlike floats, unum arithmetic guarantees bitwise identical answers across different computers. Its main drawback is that it requires more gates to implement in a chip, but gates are something we have in abundance. Memory bandwidth, power efficiency, and programmer productivity are precious and limiting. Unums help with all three issues.

1.1 Contributions

By exploiting knowledge on the information of unum, we designed a set of mathematical algorithms that enable the arithmetic computations to be feasible in the real life. We implemented logarithm, exponential and power functions using LLNL unum library in C and C++, and evaluated their performance on a small set of arbitrary numbers and intervals.

1.2 Limitations

Our research and implementations focused on the mathematical algorithm by using GMP, but the efficiency of GMP is idealization. The baseline of the efficiency of GMP is doubtable. More research on GMP needs to be done. The LLNL unum library implemented the conversion between a high precision calculated number and human readable and representable unum or ubound. Accuracy, precision and efficiency are lost. Sections 5.3.1 and Section 6 describes how those identities are lost.

1.3 Structure of this report

The organization of this report is as follows. In Chapter 2 we overview the concepts of unum, the implementations in LLNL unum version, the benefit of using GMP, as well as an intuitive guide to the mathematical functions. Chapter 3 describes the algorithms

used, and Chapter 4 provides the implementation for functions. Chapter 5 provides some experimental data on the performance of our implementation. Chapter 6 draws some conclusions from the results obtained, and addresses possible future work to further this research topic.

2. Background

2.1 Overview of unum

There are three layers in the concept of unum. The h-layer is where numbers are represented in a form understandable to humans. The u-layer is the level of computer arithmetic where all the operands are unums and data structures made from unums, like ubounds. The g-layer is the scratchpad where results are computed by arithmetic such that they are always correct to the smallest representable uncertainty when they are returned to the u-layer.

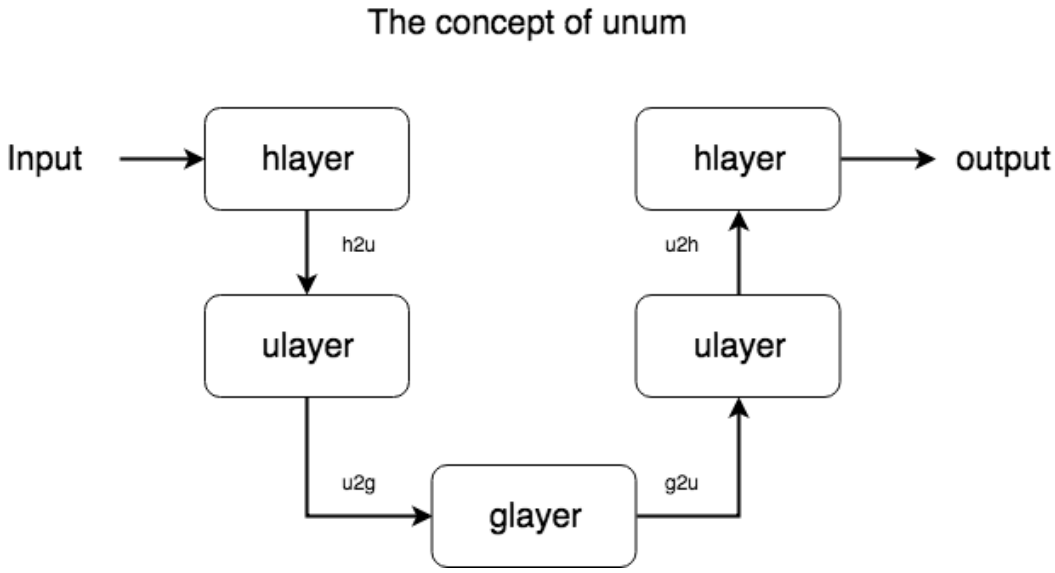


Figure 2.1 the concept of unum

A gbound is the data structure used for temporary calculations at higher precision than in the unum environment, which is similar to scratchpad in IEEE.

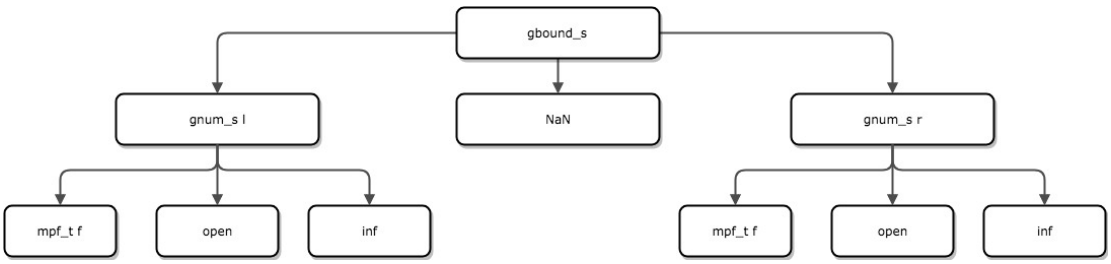


Figure 2.2, data structure for gbound

2.2 Lawerance Livermore National Laboratory version

Lawrence Livermore National Laboratory (LLNL)[1] developed as part of unum and implemented a high-level dynamic programming language in C and C++. It is functional and has been released as open source for further research. There are conversion functions that convert between unums and primitive "C" types such as integers and floats, but it may lose precision. There are few arithmetic operators have been implemented such as add, subtract, multiply, divide, square and square root. Additional relational operations are available to test whether unum intervals are contained or are completely disjointed. Several print functions are available to display values in interval notation. The unum library uses the GNU Multiple Precision Arithmetic Library for the low-level arithmetic since the primitive C types do not give enough precision.

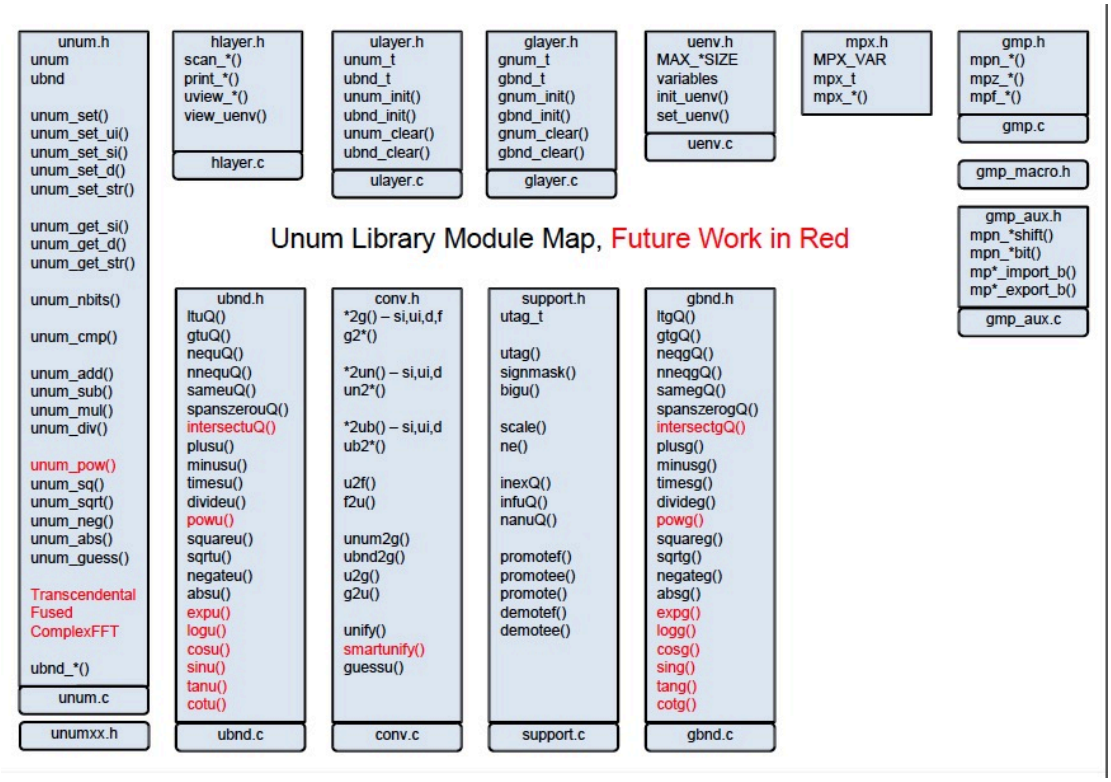


Figure 2.3, Future Work of unum library in LLNL version. Picture is released by LULESH Excetion with Unums in C/C++ in October 2016

2.3 GNU Multiple Precision Arithmetic Library

GMP[2] is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision expect the ones implied by the available memory in the machine GMP runs on. GMP is carefully designed to be as fast as possible. The speed is achieved by using

[1] Lawrence Livermore National Laboratory
https://en.wikipedia.org/wiki/Lawrence_Livermore_National_Laboratory
[2] GNU Multiple Precision Arithmetic Library, GMP
https://en.wikipedia.org/wiki/GNU_Multiple_Precision_Arithmetic_Library

fullwords as the basic arithmetic type, by using fast algorithms, with highly optimized assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed. Besides, GMP has a rich set of functions. There are several categories of functions in GMP.

mpz: high-level signed integer arithmetic functions
mpq: high-level rational arithmetic functions
mpf: high-level floating-point arithmetic functions

2.4 Intuitive guide to mathematical functions

Logarithm is a convenient way to express large numbers. For example the base_10 logarithm of a number is roughly the number of digits in that number. Lots of things "decay logarithmically". For example, hot objects cool down, cold objects warm up. And things in motion experience friction and drag and gradually slow down. In computing environment, you can probably write a computer program where the number of steps required to solve the problem is logarithmic. That is, the time taken depends on the logarithm of the amount of data to be processed. Exponential function is not just a number to the power. Exponential function is the best approach to describe the base rate of growth shared by all continually growing processes. Just like every number can be considered a scaled version of 1, every circle can be considered a scaled version of the unit circle radius 1, and every rate of growth can be considered a scaled version of e. It is widely use in computer science as well. So exponential function is not an obscure, seemingly random function. Exponential represents the idea that all continually growing systems are scaled versions of a common rate. Power function is significantly important in real life. It relates many mathematical applications, physical observables to many mechanics, and analysis, theory in computer science.

3. Design Architecture and Algorithms

Our approach is as follows: first, we implement corresponding GMP arithmetic functions due to the lack of the implementations of functions or the insufficiency of functions in GMP. Then, according to mathematical algorithm, we implement arithmetic functions in unum.

3.1 Taylor Expansion and Piecewise algorithm in logarithm

Gbound is an interval that has left and right endpoint, we need to handle all the possible numbers inside this interval. Since the logarithmic function is an increasing function with positive gradient for all the positive numbers, so that it means that for all the positive numbers if a is small than b, log(a) is always small than log(b). Therefore the logarithmic interval is actually the logarithmic value of left and right endpoint.

log(gbound(a,b)) = gbound(log(a),log(b)) for all positive a and b

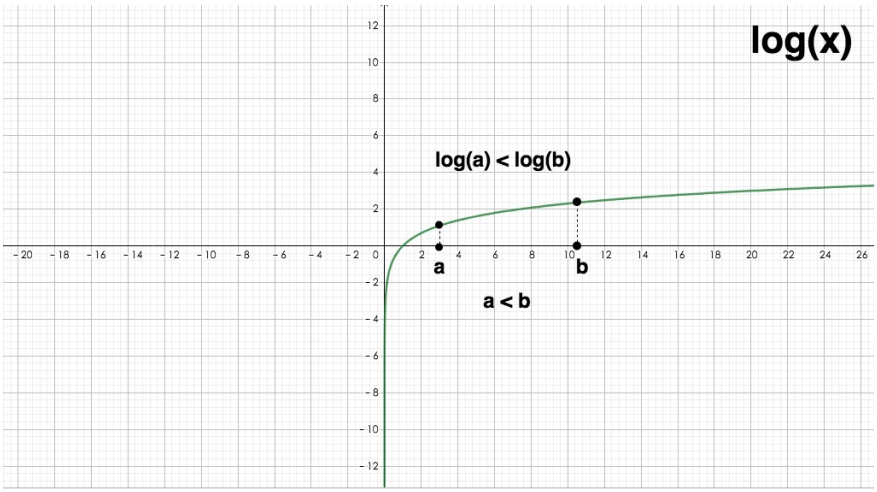


Figure 3.1, logarithm function with comparing two points

To calculate the logarithmic value of the endpoint, piecewise is one approach to improve efficiency. This is because, this method calculates the value by the most efficient way based on the subdomain it locates. And more importantly, we can handle special values, NaN, Infinity. GMP does not implement infinity and NaN, therefore, I need to implement it, thus, it supports for the whole domain.

Meanwhile, I decide to use Taylor Expansion to find out the logarithmic value, since it is the most efficient approach compare to the other methods.

3.1.1 Milestone to design logarithm in unum

- Method 1: Piecewise [3]

I design to use piecewise to evaluate the value, hence improving the efficiency. There are few cases need to be handle.

$$\log(x) = \begin{cases} 0 & \text{if } x = 1 \\ \text{NaN} & \text{if } x < 0 \\ \text{NaN} & \text{if } x = \text{NaN} \\ -\infty & \text{if } x = 0 \\ +\infty & \text{if } x = +\infty \\ \log(x) & \end{cases}$$

If the value of x is exactly 1 then it is always 0. This is the only value which has exact value in the logarithmic arithmetic. Furthermore, if the value of x is less than 0 or NaN, it always returns NaN due to the mathematical algorithm. If the value of x is positive infinity, it returns to positive infinity, and if it is 0, it is negative infinity. Otherwise, for all the values that do not satisfy these special cases, evaluate trivially to the logarithmic input value by using mpf_log.

In order to evaluate a logarithmic function at a given input value, the appropriate subdomain needs to be chosen in hence to produce the correct output value. Therefore, it improves the efficiency. For example, if the input value is 1, we can straightforward to output 0 instead of doing extremely complicated computations.

3.1.2 Milestone to design logarithm in GMP

- Method 1: Taylor Series Approximation

For log(x) where x > 1, the closer the value of x is to 1, the faster the rate of convergence [4]. The identities associated with the logarithm can be leveraged to exploit this:

$$\begin{aligned} \log(x) &= \log(10^n \cdot a) && \text{where } x = 10^n \cdot a \text{ and } n \in \text{Integer}, a \in \text{one digit in real} \\ &= \log(10^n) + \log(a) && \text{by logarithmic product identity} \\ &= n \cdot \log(10) + \log(a) && \text{by logarithmic power identity} \\ &= n \cdot \log(10) + (\frac{a}{p} - 1) && \text{by } \log(1+x) \approx x \text{ if } 1+x \text{ is close to } 1 \end{aligned}$$

[3] In mathematics, a piecewise function is a function defined by multiple sub-functions, each sub-function applying to a subdomain
https://en.wikipedia.org/wiki/Piecewise

[4] Convergence speed：In numerical analysis, the speed at which a convergent sequence approaches its limit is called the rate of convergence
https://en.wikipedia.org/wiki/Rate_of_convergence#Convergence_speed_for_iterative_methods

It is a way to solve logarithmic operation with high efficiency. This is because, there are only few calculations needed to do to figure out the result. And the approximation is not complex, literally finding the decimal number.

But is it accurate? The approximation works for the numbers really close to 1, but if the number is far away from 1.

- Method 2: Gauss-Legendre 2-point quadrature and find prime

The algorithm is to first turn the number into natural number and then turns the natural number into the multiplication, which represented by few prime numbers and a number, which is close to 1. The algorithm can be represented by:

$\log(x) = \log(10^n \cdot a)$ where $x = 10^n \cdot a$, $n \in$ Integer, $a \in$ Natural number

$= \log(10^n) + \log(a)$ by logarithmic product identity

$= n \cdot \log(10) + \log(a)$ by logarithmic power identity

$= n \cdot \log(10) + \log(p \cdot \frac{a}{p})$ where p =multiplication with primes factors

$$\text{and } \frac{a}{p} \approx 1$$

$= n \cdot \log(10) + \log(p) + \log(\frac{a}{p})$ by logarithmic product identity

$= n \cdot \log(10) + \log(p) + \frac{6(\frac{a}{p}-1)+3(\frac{a}{p}-1)^2}{6+6(\frac{a}{p}-1)+(\frac{a}{p}-1)^2}$ by $\log(1+x) \approx \frac{6x+3x^2}{6+6x+x^2}$ if 1+x is close to 1

An example of log(11.3):

$\log(11.2) = \log(10^{-1} \cdot 113)$

$= \log(10^{-1}) + \log(113)$

$= -1 \cdot \log(10) + \log(113)$

$= -1 \cdot \log(10) + \log(2^4 \cdot 7 \cdot \frac{113}{2^4 \cdot 7})$

$= -1 \cdot \log(10) + \log(2^4 \cdot 7) + \log(\frac{113}{112})$

This algorithm is an improvement of Method 1. The most important improvement is it works for all numbers. This is because, the number always convert to a number, which is close to 1. Also, it makes the results more accurate and precise. This is because; first it finds out all the possible representable accurate and precise values, therefore, there is less error prone on these values. Secondly, it improves the approximation. Instead of using Taylor Series approximation, we use Gauss-Legendre 2 point quadrature. It has less error; hence the result is more accurate.

The fewer parts need approximation, the less effect on the result. And also, more accurate approximation means the less error prone. Hence, the result is more accurate than Method 1.

It is a large work to find all the prime factors. To find all the prime factors, we can implement Sieve of Eratosthenes algorithm [5], which is the most efficient approach to find all the factors, O(n log logn) as time complexity. But, how can we get logarithmic values on these prime numbers? We could implement by build in, but it is brute force especially for finding prime factors for large number. For example, to find factors for 99999, it can be factored into $3 \cdot 3 \cdot 41 \cdot 271$, thus we need all the logarithmic values for these numbers.

- Method 3 Euler's Algorithm

The algorithm is to calculate log using square roots and the other identities of log. The general idea is to pin the logarithm down into two logarithms that are easy to calculate or known, then keep narrowing the range of the endpoints to figure out the logarithm value of the number. It starts at the range between 1 and 10, since $\log_{10}(1) = 0$ and $\log_{10}(10) = 1$.

$\log(x) = \log(10^n \cdot a)$ where $x = 10^n \cdot a$ and $n \in$ Integer, $a \in$ one digit in real

$= n \cdot \log(10) + \log(a)$ by logarithmic product and power identity

$= n \cdot \log(10) + \frac{log_{10}(a)}{log_{10}(e)}$ by logarithmic change of base

If $n < a < m$, this means that $\log_{10}(n) < \log_{10}(a) < \log_{10}(m)$, then using

$$\log_{10}(\sqrt{mn}) = \frac{\log10(m)+\log10(n)}{2}$$

This is because,

$\log_{10}(\sqrt{mn}) = \log_{10}(\sqrt{m}) + \log_{10}(\sqrt{n})$

$= \log_{10}(m^{\frac{1}{2}}) + \log_{10}(n^{\frac{1}{2}})$

$= \frac{\log10(m)+\log10(n)}{2}$

Keep looping to narrow the range until the range is precise.

It is an improvement of Method 2. Compare to Method 2, with the prerequisite of accuracy and precision, method 3 improves the efficiency. We only need to know few log values, $\log(10)$ $\log_{10}(e)$, rather than all the prime numbers as well as their log values. Therefore, this method is achievable with efficiency and accuracy.

But it takes a lot of iterations to pin the endpoint down; even it can certainly get the result. The accuracy is another problem. Square root of a number is normally not an

[5] Sieve of Eratosthenes: It is a simple, ancient algorithm for finding all prime numbers up to any given limit https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

exact number, hence rounding all the time when narrowing the range. And also, converting from log to $\log_{10}$ loses accuracy, as well as converting it back, since all the numbers are normally irrational.

- Method 4 Taylor Expansion

$$\ln(x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(x-1)^n}{n}$$

Figure 3.2 Equation for Taylor Expansion for natural logarithm

This has one major caveat, Taylor Expansion only converges if a ∈ (0,2)[6]. Therefore, I design to pin the number down to less or equal to 2. And then implement Taylor expansion to figure out the number. This can be exploited as:

$\log(x) = \log(2^n \cdot a)$ where $x = 2^n \cdot a$ and n ∈ integer, a ∈ (0,2)

$= n \cdot \log(2) + \log(a)$ by logarithmic product and power identity

$= n \cdot \log(2) + \sum_{k=1}^{\infty} (-1)^{k+1} \frac{(a-1)^k}{k}$ by Taylor Expansion

We can now expand the Taylor series above to as many digits as needed to achieve the required accuracy. Hence, it is most accurate one compare to the previous methods, no conversion, no approximation. Get as precise as we want. Meanwhile, it has a high efficiency as well. If calculate $\log(\frac{1}{3})$ in 5 significant digits, it needs 22 iterations to get -1.0986 accurately. In a sense, 22 iterations are efficient.

There is still limitation on this. The closer number gets to 0 or 2, the longer it takes to converge to something useful. For example, while calculating $\log(\frac{1}{30})$ in 5 significant digits accurately, it needs 211 iterations to get -3.4011. Therefore, I design to pin the number close to 1 to solve this problem.

Comparison Table

The comparison between method 1,2,3,4 to calculate log(17.1) in 4 significant digits

	Method 1	Method 2	Method 3	Method 4

[6] Doug Glasshod. Taylor polynomials, 2006. Note that, beyond 2 or less than 1, Taylor polynomials of higher degree are increasing worse approximations
https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1014&context=mathmidexppap

Simplify	1·log(10)+log(1.71)	-1·log(10)+log(2·3⁴)+log($\frac{19}{18}$)	1·log(10) + $\frac{log_{10}(1.71)}{log_{10}(e)}$	4·log(2) + $\sum_{k=1}^{\infty}(-1)^{k+1}\frac{0.06875}{k}$
Approx	1·log(10)+0.71	-1·log(10)+log(2·3⁴)+ 0.05406721	1·log(10) + 0.53647112	4·log(2) + 0.06649503
Iteration	1+0	1+45+0	1+14	4+3
Result	3.0125	2.8390	2.8390	2.8390
log(17.1)-result	-0.1735	0	0	0
Error	6.11%	0%	0%	0%

Figure 3.3, A comparison between Method 1,2,3,4 in 4 significant digits

The comparison between method 1,2,3,4 to calculate log(17.1) within 20 iterations for approximation and 20 decimals

	Method 1	Method 2	Method 3	Method 4
Simplify	1·log(10)+ log(1.71)	-1·log(10)+ log(2·3⁴)+log($\frac{19}{18}$)	1·log(10) + $\frac{log_{10}(1.71)}{log_{10}(e)}$	4·log(2) + $\sum_{n=1}^{\infty}(-1)^{n+1}\frac{0.06875}{n}$
Approx	1·log(10)+0.71	No result	1·log(10) + 0.53649528010962 60251918235	4·log(2) + 0.06648974126883 29211125979
Iteration	1+0	20	1+20	4+20
Result	3.0125850929940 4568401	No result	2.83908037310367 170920	2.83907846350861 415878
log(17.1) -result	-0.173506629485 43152522	No result	-1.90960×10^{-6}	1.52645×10^{-21}
Error	6.11×10^{-2}	No result	6.72613×10^{-7}	5.27657×10^{-22}

Figure 3.4, A comparison between method 1,2,3,4 in 20 iterations

From comparison in 4 significant digits (Figure 3.3), we can see that Method 1 returns to an incorrect result. From comparison in 20 iterations (Figure 3.4), with the same iterations, Method 4 gets more accurate answer. Therefore, Method 4 is our approach to implement logarithm, which is Taylor Expansion.

3.2 Taylor Series and Piecewise algorithm in exponential

The exponential function is continuous. It is a monotonically increasing function with positive gradient all the time, so that it means if a is smaller than b, e^a is certainly smaller than e^b. Therefore, the exponential interval of an interval is an interval, which takes exponential of both endpoints in the interval.

$e^{(a,b)} = (e^a, e^b)$ for any value of a and b

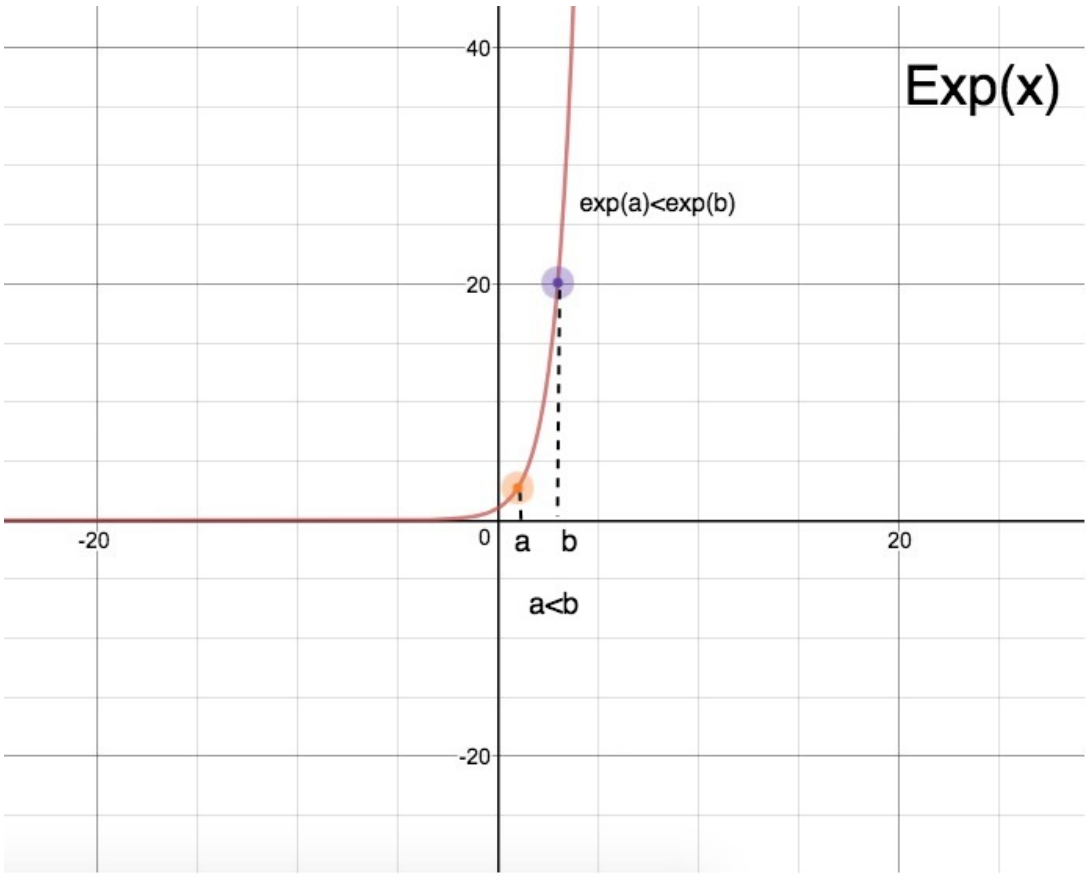


Figure 3.5, Exponential function with comparing two points

To calculate the interval, I design to use piecewise method to calculate each endpoint, which is similar to logarithmic function. While, I design to use Taylor Series to evaluate exponential value.

3.2.1 Milestone to design exponential in unum

- Method 1: Piecewise

I design to use piecewise to evaluate the value, hence improving the efficiency. There are few cases need to be handle.

exp(x) = | 1 if x = 0

| NaN if x = NaN
| 0 if x = $-\infty$
| $+\infty$ if x = $+\infty$
| exp(x)

If the value of x is 0 then it is always 1, which is the special value in the exponential operation. If the value of x is NaN, it always returns NaN due to mathematical algorithm. If the value of x is positive infinity, it returns positive, and if the value is negative infinity, it is 0. Otherwise, evaluate trivially to the exponential function (mpf_exp) with the input value.

3.2.2 Milestone to design exponential in GMP

- Method 1: Taylor Series

Although we do have *mpf_pow* function as a built-in function in GMP, it provides convenience for us. However, there is still insufficiency. This power function only supports for the arbitrary number to the unsigned integer. What if it is a signed integer? This could be solved by using the identity $x^{-a} = 1 / x^a$. But how can we solve the problem when the exponent is a decimal number? For example, $2.1^{1.3}$ is not supported. I design to use Taylor Series.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Figure 3.6 Equation for Taylor Series for natural exponential

To improve the accuracy, I design to separate the input into two parts, real and decimal.

$e^x = e^a \cdot e^b$ where a ∈ Integer, x = a + b

This is because, Taylor Series is close enough to the actual result, but it is not the exactly result. The smaller part that we need to approximate, the more accurate the result is.

3.3 Transcendental functions and Remedy in power

Instead of use e as the base, we can use any arbitrary number as base. Encountering the same issue as discussed in exponential function. I design to use transcendental functions to implement *mpf_pow*.

We do know that there is some loss precision in the *mpf_exp* and *mpf_log* due to they are using precise enough fixed value like log(2) and e, and a precise enough approximation, however, it is not the exact value. Hence, to make sure that the result is accurate, I design to enlarge a relative precise width of interval depends on the environment, instead of generating meaningless interval like, $(-\infty, +\infty)$. Therefore, it balances precision and correctness.

3.3.1 Milestone to design power in unum

- Method1: Piecewise

  Unlike logarithm and exponential implementation in unum, power is sophisticated, since there is no predicated range of interval. Firstly evaluate any possible result without any computation, especially infinity and NaN. Note that there is one particular NaN in power due to the limitation of the concept of unum, if x^y and x is negative and y is an interval, then it returns NaN, due to there are lots of possible clauses as result. Instead of the special cases, I design to calculate the Cartesian Product[7], which is $left^{left}$, $left^{right}$, $right^{left}$, $right^{right}$. Return a largest interval, hence contains all the possible results.

Here is a table to represent all the piecewise cases for all the values.

y \ x	$-\infty$	Negative	0	Positive	$+\infty$	NaN
$-\infty$	NaN	NaN	NaN	0	0	NaN
Negative	0	$(-1)^{Positive}$mpf_pow	NaN	mpf_pow	0	NaN
0	1	1	NaN	1	1	NaN
Positive	$(-1)^{Positive}\infty$	$(-1)^{Positive}$mpf_pow	0	mpf_pow	$+\infty$	NaN
$+\infty$	NaN	NaN	0	$+\infty$	$+\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 3.7, Special values and domains in x^y

A further check is needed to ensure the complete interval. It causes when the result which both endpoints are greater than 0. For example, $(-2,3)^2 = (4,9)$ by our design, however, the result should include zero.

3.3.2 Milestone to design power in GMP

- Method 1: Shifting nth root algorithm

  According to mathematical identities,

[7] In set theory, a Cartesian product is a mathematical operation that returns a product set from multiple sets. https://en.wikipedia.org/wiki/Cartesian_product

$m^n = m^{\frac{numerator}{denominator}}$ where $n = \frac{numerator}{denominator}$

$= \sqrt[denominator]{m^{numerator}}$

Since we have built in function *mpf_pow_ui*, we can figure out the value of $m^{numerator}$, then use shifting nth root algorithm to find out the $\sqrt[denominator]{}$. To achieve this method, we need to find out $n = \frac{numerator}{denominator}$ first. It can be easily find by taking $n = \frac{10^{decimal\ digits} \cdot n}{10^{decimal\ digits}}$. Then, find out the greatest common divisor [8].

$$n = \frac{10^{decimal\ digits} \cdot n/gcd}{10^{decimal\ digits}/gcd} = \frac{numerator}{denominator}$$

For example, if we have n = 11.2,

$n = 11.2$

$= \frac{112}{10}$

$= \frac{112/2}{10/2}$ due to gcd = 2

$= \frac{56}{5}$

And then

$m^{11.2} = \sqrt[5]{m^{56}}$

Shifting nth root algorithm is an algorithm for extracting the nth root of a positive real number which proceeds iteratively by shifting n digits of the radicand, starting with the most significant, and produces one digit of the root on each iteration, in a manner similar to long division.

1. Initialize r and y
2. While not ((not desired precision is obtained) -> (exact))
    1. Let α be the next aligned block of digits from the radicand.
    2. Let β be the largest possible value of
    $$(10 \cdot y + \beta)^{\text{radicand}} \leq 10^{\text{radicand}} \cdot (y^{\text{radicand}} + r) + \alpha$$
    3. $y' = 10 \cdot y + \beta$
    4. $r' = 10^{\text{radicand}} \cdot r + \alpha - (y')^{\text{radicand}} + 10^{\text{radicand}} \cdot y^{\text{radicand}}$
    5. Assign $y \leftarrow y'$ and $r \leftarrow r'$

[8] In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.
https://en.wikipedia.org/wiki/Greatest_common_divisor

6. A series of β is the exact value

Take an example

Find $\sqrt[2]{2}$:

```
      1.   4   1   ...
     ________________
2 / 02.   00  00  00
\/
```

Loop 1:

$\alpha = 02$

$\beta = 1, \quad (10 \cdot 0 + 1)^2 \leq 10^2 \cdot (0^2 + 0) + 2$

$y' = 1 = 10 \cdot 0 + 1$

$r' = 1 = 10^2 \cdot 0 + 2 - (1)^2 + 10^2 \cdot 0^2$

$y = 1$

$r = 1$

Loop 2:

$\alpha = 00$

$\beta = 4, \quad (10 \cdot 1 + 4)^2 \leq 10^2 \cdot (1^2 + 1) + 0$

$y' = 14 = 10 \cdot 1 + 4$

$r' = 4 = 10^2 \cdot 1 + 0 - (14)^2 + 10^2 \cdot 1^2$

$y = 14$

$r = 4$

Loop 3:

$\alpha = 00$

$\beta = 1, \quad (10 \cdot 14 + 1)^2 \leq 10^2 \cdot (14^2 + 4) + 0$

$y' = 141 = 10 \cdot 14 + 1$

$r' = 119 = 10^2 \cdot 4 + 0 - (141)^2 + 10^2 \cdot 14^2$

$y = 141$

$r = 119$

This method can figure out the exact result of the root, as well as power.

However, there are some limitations. For example, GMP needs to be initialized to an amount of precision bits. y value and r value grows sufficiently fast, and power grows exponentially as well. It is hard to determine how many precision bits it requires. If it is extend the maximum number in declared precision bits, it returns infinity, in which leads to chaos.

- Method2: Transcendental functions

The method is in the form of

$$m^n = e^{n \cdot log(m)}$$

This follows from the fact that exponentials and logarithms are inverse to each other, and that the exponentials and logarithms have the property that

$$m^n = e^{log(m^n)} = e^{n \cdot log(m)}$$

However, there are a few caveats, this method is based on exponential and logarithm. Log(m) is undefined for m<0, while, power allows computation with negative base m. Exponential suffers from error magnification. I design to solve this by enlarging the relative width. This idea comes from double-float computation. [9][10] Double-float is a technique that uses pairs of single-precision numbers to achieve almost twice the precision of single precision arithmetic accompanied by a slight reduction of the single precision exponent range. I design to return an interval for power to make the result accurate without losing too much precision by using ulp [11].

$$m^n = (e^{n \cdot log(m)}, e^{n \cdot (log(m\_ulp) + ulp)})$$

[9] Extended-precision floating-point numbers for GPUComputation
http://andrewthall.org/papers/df64_qf128.pdf

[10] Implemention of float-float operators on graphics hardware
https://hal.archives-ouvertes.fr/file/index/docid/63356/filename/float-float.pdf

[11] In computer science and numerical analysis, unit in the last place is the spacing between floating-point numbers
https://en.wikipedia.org/wiki/Unit_in_the_last_place

4. Implementation

4.1 Logarithm

4.1.1 Logarithm in unum

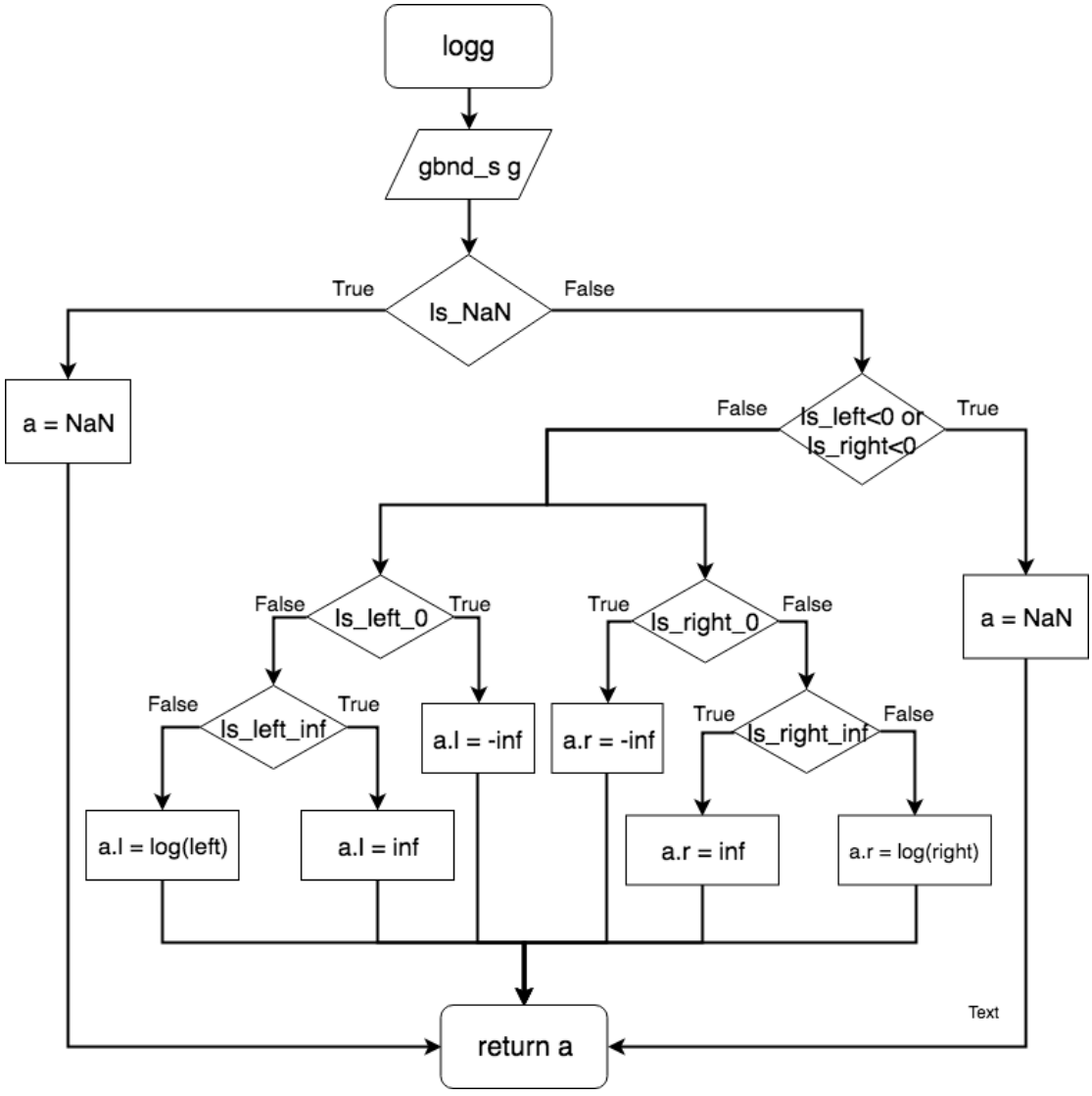


Figure 4.1, Control flow diagram of implementing logg

I implement the idea to check the interval whether it is NaN or any endpoints are less than 0 first.

Is_NaN can be checked by *g->nan*, 0 indicates not NaN

Is_<0 can be found by *mpf_cmp_ui(g->l.f , 0)*,

return positive if the value of left interval of g > 0
return negative if the value < 0
return 0 if the value = 0

There is no need to do computations any further when it satisfies the conditions, and we set *a->nan=1* to indicate it is a NaN.

Then, I evaluate for both endpoints. If the number is 0, it returns to negative infinity.

Set_Infinity: *a->l.inf = 1*, 1 indicates the value of left interval of a is infinity
Set_Negative: *mpf_set_si(a->l.f , -1)*, set value -1 to the left interval of a

If the number is positive infinity, it returns to positive infinity.
Check_Positive: *mpf_sgn(g->l.f)*,
return positive if the value of left interval of g is positive
return negative if the value negative
return 0 if the value 0

Otherwise, calculate the number by logarithmic (*mpf_log*).

Since, the precision I set is in decimal, I set *(PBITS+4)/3.322* as precision. *PBITS* is the maximum bits that we can represent in the environment. The ratio of bits to decimal digits is round 3.322.[12] While ensuring accuracy, I implement to calculate one more decimal digit (*+4*); it may be an excess of the maximum number in the environment. But it improves accuracy.

For instance, assume that the environment is able to represent 3 decimal digits
log([2,3]) = [0.693,1.098] without extending
= [0.693,1.098] as result

But log(3) = 1.098612 which is not in the result interval, hence the result is wrong. With one more decimal digit, it will enlarge the interval, which contains all the value.

For instance, with the same environment
log([2,3]) = [0.6931,1.0986] with extending
= [0.693,1.099] as result

It contains all the possible logarithmic values, which is an accurate evaluation.

In unum, the domain is slightly different. unum has closed and open infinity. Closed infinity represents exactly infinity and the open infinity represents a really large number, which could not be represented by the number of bits, but it is not the same as infinity. Since, log is an increasing function, the output domain depends on

[12] Ratio of Bits to Decimal Digits
http://www.exploringbinary.com/ratio-of-bits-to-decimal-digits/

the input domain. If it is open interval, the logarithmic is an open interval. This can be done by

a->l.open = g->l.open, the bracket of left interval of a = the bracket of left interval of g

It works for infinity and 0 as well, if the domain includes 0(close bracket), it returns closed infinity.

According to the piecewise design, I consider the number 1, it is a special value in logarithmic, however, I did not implement that in unum implementation. This is because, unum is to evaluate the interval, not the one that calculates logarithm, and thus, I implement it in *mpf_log*.

4.1.2 Logarithm in GMP

mpf_log (mpf_t log, const mpf_t x, unsigned int prec)

mpf_log takes three parameters, log(output), x(input) and prec(precision). The precision is in decimal digits.

If x is less or equal to 0, it should return NaN, since we can't represent NaN in mpf, then I implement it by returning nothing and considering it in unum implementation. If x is equal to 1, return 0 directly. Otherwise, we do the algorithm:

```
mpf_log:
        int n
        while x > 2:
            x = x ÷ 2
            n += 1
        while x < 1
            x = 2 · x
            n -= 1
        log = n · log(2) – Taylor_expansion(1 - x)
        return log
```

Figure 4.2: mpf_log algorithm

To find $x \div 2$ or $2 \cdot x$, by using *mpf_div_ui* or *mpf_mul_ui*.

mpf_div_ui(tmp,tmp,2): store tmp divided by 2 in tmp

In addition, since gmp does not support for any multiplication between value and signed integer, I implement *mpf_mul_si*.

```
mpf_mul_si(factor a,b):
    if (any factor is 0) return 0
    result = abs(a) * abs(b)
    if (a,b in the same sign)
        return result
    else
        return –result
```

Figure 4.3: mpf_mul_si algorithm

Then to implement Taylor expansion:

```
Taylor_expansion(x):
    minterm = find smallest term
    result = 0
    n = 1
    while > minterm:
        result = result + x^n / n
    return result
```

Figure 4.4: Taylor Expansion for logarithm algorithm

I implement Taylor expansion in another way which saves more computations.

Taylor expansion:

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

$$= -(1-x) - \frac{(1-x)^2}{2} + \frac{-(1-x)^3}{3} - \ldots + \frac{(1-x)^n}{n} \quad \text{for x >=1}$$

$$= -[(1-x) + \frac{(1-x)^2}{2} + \frac{(1-x)^3}{3} + \ldots + \frac{(1-x)^n}{n}] \quad \text{for x>=1}$$

Therefore, we do not need to consider any subtractions. That is why I implement to find an x which is greater than 1.

4.2 Exponential

4.2.1 Exponential in unum

It is similar to logarithm implementation in unum. The process and control flow in logarithm support exponential function as well. According to the domain, exponential function has the same implementation as logarithm.

4.2.2 Exponential in GMP

mpf_exp(mpf_t exp, const mpf_t x, unsigned int prec)

mpf_exp takes three parameters, exp(output), x(input) and prec(precision). The precision is in decimal digits.

It is a continuous function for all domains. To make it more piecewise, if x is equal to 0, return 1 straightaway without doing any further computations. Otherwise, we do the algorithm:

```
mpf_exp:
    real = real part in x
    decimal = decimal part in x
    exp = e^real · e^Taylor_Series(decimal)
    return exp
```

Figure 4.5: mpf_exp algorithm

```
Taylor_Series(x):
    minterm = find smallest term
    result = 1
    n = 1
    while > minterm:
        fact = n!
        result = result + x^n / fact
    return result
```

Figure 4.6: Taylor Seies for exponential algorithm

If real is negative, due to there is no implementation for the power to the negative integer, I implement $e^{-x} = \frac{1}{e^x}$ to solve this problem.

To find factorial of n, I implement the idea of by using *mpz_fac_ui*, it is a built-in function for mpz.

mpz_fac_ui(fact,n): store n! in fact

However, the factorial value may extend the maximum representable value according to the declared precision bits. Therefore, I implement the idea to find the maximum digits to represent a factorial number, and set it as precision bits.

Find_factorial_digits: $n! = \sum_{k=1}^{n} log_{10}(k) + 1$

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

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

And add one extra digit as insurance and also to remedy the inaccuracy of ln(10).

4.3 Power

4.3.1 Power in unum

To find the power of the Cartesian Product, which is $left^{left}$, $left^{right}$, $right^{left}$, $right^{right}$ in unum.

*void power(gnum_s *a, gnum_s *b, const gnum_s *x, const gnum_s *y, int *nan)*

x, y is the base and exponent. a, b is the minimum and maximum value of *mpf_power*. Nan indicates whether not a number. This function is computed according to piecewise design (Figure 3.7).

Combined these minimum and maximum value together to form a largest representable interval by using *combinegb*, hence all the possible values are in this interval.

*combinegb(gbnd_s *interval, const gbnd_s *x, const gbnd_s *y)*

The open or close interval depends on the base and exponent, but differently for infinity. If there is any infinity, the open or close bracket only depends on the infinity number.

a->l.open = x->l.open && y->l.open if no infinity
= x->l.open if left of base is infinity
= y->l.open if left of exponent is infinity

Since, it may cause incomplete result when the interval is in positive. Therefore, I implement to check whether the result contained zero by using *spanszerogQ* when the left endpoint of the result is greater than 0. If left is positive, it always a positive interval.

spanszerogQ(a): test if interval spans zero

4.3.2 Power in GMP

mpf_pow(mpf_t min, mpf_t max, const mpf_t base, const mpf_t exponent, unsigned int prec)

mpf_exp takes five parameters, min(output), max(output), base(input), exponent(input) and prec(precision). The precision is in decimal digits.

There are few subdomains needs to consider.

x^y = | 1 if x != 0 and y = 0
| x if y = 1
| 0 if x = 0 and y >0
| NaN if x = 0 and y<0 or 0
| NaN if x <0 and y != int
| x^y if y = int by using *mpf_pow_ui*
| $x^{floor(y)} * \sqrt{x}$ if y = 0.5 * n by using *mpf_sqrt*

| $(-1)^y e^{y \cdot log(x)}$

To check whether it is an interger or float,

mpf_integer_p(y): it returns 0 if y is an integer, otherwise it is float

As designed, I implement *mpf_pow* with two outputs, minimum and maximum. For the one that has exact value, set minimum and maximum to be the same. For the one uses transcendental functions, I consider the minimum and maximum in these cases:

x / y	< 0	> 0
< 0 and even	Min = $e^{y \cdot log(-x+ulp)}$ Max = $e^{y \cdot log(-x)+ulp}$	Min = $e^{y \cdot log(x)}$ Max = $e^{y \cdot log(x)+ulp}$
< 0 and odd	Min = $-e^{y \cdot log(-x)+ulp}$ Max = $-e^{y \cdot log(-x+ulp)}$	Min = $e^{y \cdot log(x)}$ Max = $e^{y \cdot log(x)+ulp}$
> 0 and even	Min = $e^{y \cdot log(-x)}$ Max = $e^{y \cdot (log(-x+ulp)+ulp)}$	Min = $e^{y \cdot log(x)}$ Max = $e^{y \cdot (log(x+ulp)+ulp)}$
> 0 and odd	Min = $-e^{y \cdot (log(-x+ulp)+ulp)}$ Max = $-e^{y \cdot log(-x)}$	Min = $e^{y \cdot log(x)}$ Max = $e^{y \cdot (log(x+ulp)+ulp)}$

Figure 4.7, maximum and minimum value by using transcendental functions

Thus, the interval between minimum and maximum contains all the value.

To find ulp, since it is in decimal, I implement to find the smallest decimal number depends on the environment.

ulp(u, prec): set u be the ulp base on the environment precision.

5. Performance Evaluation

Our approach is as follows: first, we test the accuracy, which is the most important for the concept of unum. Then, test the performance by relative width of ubound and time efficiency.

5.1 Accuracy

Running 20 arbitrary numbers in {3,4}, {3,5}, {4,6} environment. These environments contains half, single and double precision. 20 arbitrary numbers are definitely not enough to test correctness. Few environments are not enough to verify the conclusion. Since time constraint, assume that 20 arbitrary numbers are able to give us conclusion of correctness.

By testing, all the results are correct.

5.2 Relative width of ubound

The relative width of ubound can reflect the precision of the result. If the result is more precise, it is a better result.

Running 20 arbitrary numbers in {3,4}, {3,5}, {4,6} environment. Since time constraint, assume that 20 arbitrary numbers are able to give us an approximately relative width of ubound.

5.2.1 Logarithm

Corresponding data is in logarithm.xlsx

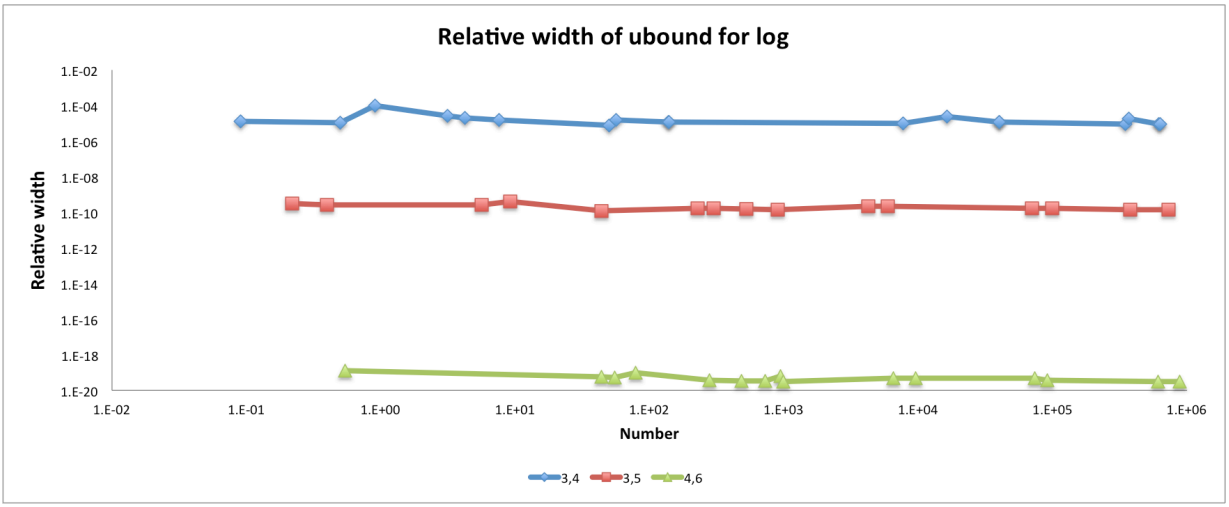


Figure 5.1, Relative width of ubound for logarithm function

	3,4	3,5	4,6
Average	1.89E-05	2.03E-10	5.28E-20
Standard Deviation	2.12608E-05	8.13758E-11	2.85017E-20

Figure 5.2, Average and Standard Deviation for relative width of ubound for logarithm function

From the data of logarithm result, we can see that there are few that calculate to the maximum interval (maximum, infinity) and minimum positive interval (0, minimum real). Those are not considered as a valid data to analysis for the relative width of ubound, this is because the width is fixed and it only depends on the true value, therefore, the result is an out of ordinary number. For example, assume we have a minimum positive representable number is 1 in the unum environment. The true value is 10^{-10}. If the result by calculation is 10^{-10}, but base on there is no representable for this result, it returns (0,1) in ubound. The relative width is $\frac{1-0}{abs(10^{-10})} = 10^{10}$ which is not supposed.

The graph shows the relative width of ubound in different environments by calculating logarithm. It can clearly see that the relative width is steady for each environment due to the floating rate is small and a small standard deviation as evidence. This assumes that the data is reliable and we can convince ourselves that the conclusion stands. The relative width of ubound decreases when the precision in environment increases. This can be proved by, on average, the relative width of ubound is 1.89E-05 in {3,4} environment and it decreases to 2.03E-10 in {3,5} environment. The difference between {3,4} and {3,5} is $16(2^5-2^4)$ bits in fraction. More bits in fraction means that more precision in environment, and leads to less relative width of ubound, hence more precise. This conclusion is supported by the evidence of {4,6} environment as well.

5.2.2 Exponential

Corresponding data is in exponential.xlsx

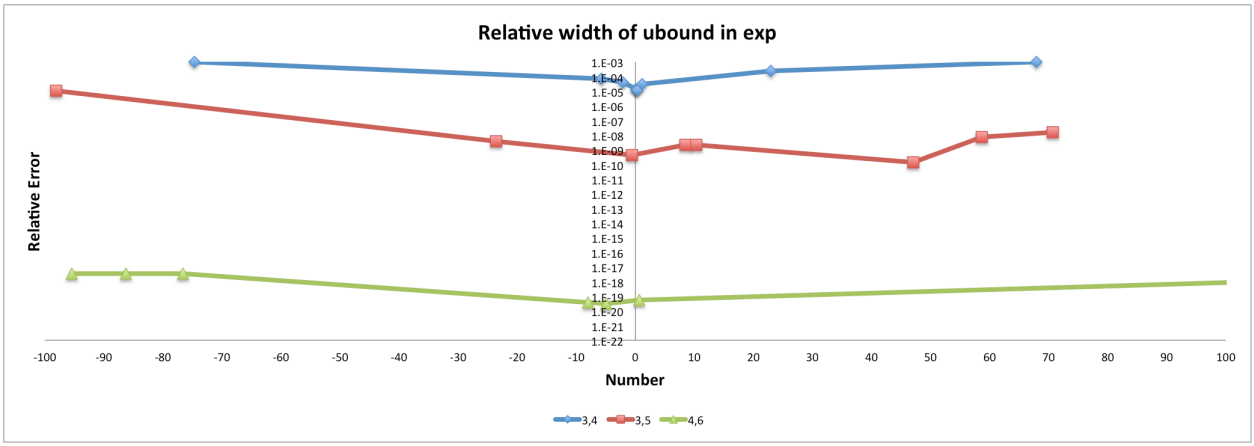


Figure 5.3, Relative width of ubound for exponential function

	3,4	3,5	4,6
Average	2.67E-04	1.33E-06	1.09E-16
Standard Deviation	4.14E-04	3.74E-06	1.35E-16

Figure 5.4, Average and Standard Deviation for relative width of ubound for exponential function

The graph shows the relative width of ubound in different environments by calculating exponential. From the graph, we can observe that it is steady for each environment. I assume the data is valid to analysis. On average, the relative width of ubound for {3,4}, {3,5}, {4,6} is 2.67E-04, 1.33E-06, 1.09E-16. By comparison, more precisions in the environment, more precise for the result.

5.2.3 Power

Corresponding data is in power.xlsx

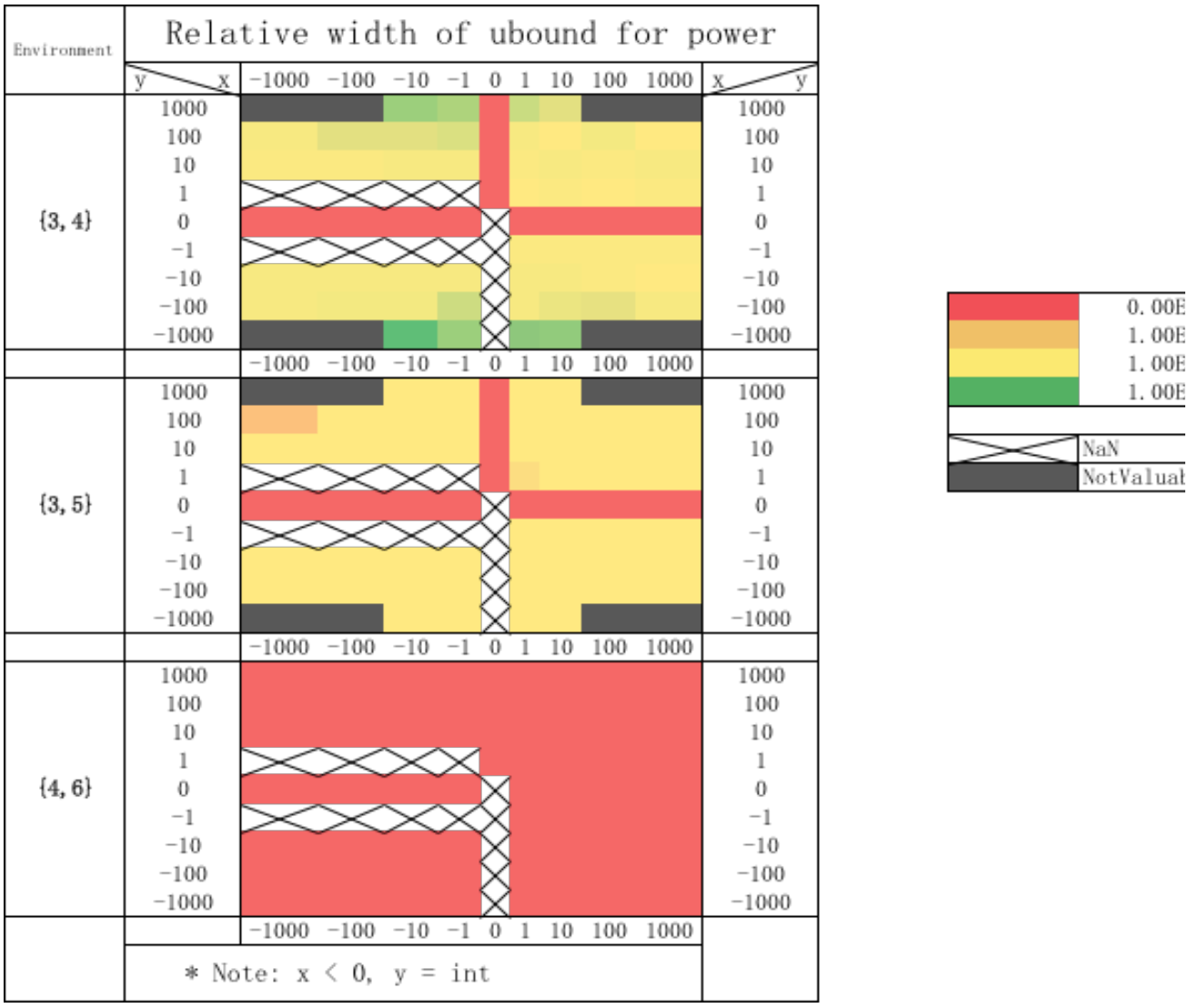


Figure 5.5, Relative width of ubound for power function

I produce a heatmap to observe the relative width of ubound in different environments by power function.

From the {3,4} environment, four corners are not valuable in this case as mentioned in logarithm operation. We can see that relative widths is relatively large when the exponent is large in positive and small in negative, especially when the power result is close to corners. So that, closer to the maximum or minimum value, less precision for the relative width. Most of the relative width is close to 1E-05. And there are few exact results, which is when x is equal to 0 or y is equal to 0. This is because, any value except 0 to the power of 0 is 1, and 0 to the power of any value except 0 is 0. We can see that there are few NaN when x is 0 and y is between 0 and 1 or between -1 and 0. It does tally with the mathematical theory, negative base to the power of non integer is NaN.

From the {3,5} environment, it is similar to the {3,4} environment, but with more precise relative width. Most of the relative width is 1E-09. It is reasonable due to there are more fraction bits in {3,5}. Therefore, the more fraction bits, the more precise relative width.

From the {4,6} environment, we can see that the relative width is as close as to 0, about 1E-20, which is reasonably small. That means the interval is in fact close to the exact value. Furthermore, since more exponent and fraction bits, there is a representation for the large or tiny number, thus four corners is valuable.

In conclusion, the more precise environment, the more precise relative width.

5.3 Time efficiency

I decide to test the time cost, thereby analysis the efficiency. The shorter time it takes, the implementation is more efficient.

Randomly generating 10000 numbers and calculating these numbers by functions, record the time for {3,4}, {3,5}, {4,6}, {4,7} and IEEE environment for 10 times in 2.5GHz Intel core i5 Mac 2012. These environments contains half, single, double and quadrant precision in IEEE. I decide to record the time in five decimal digits in milliseconds. To choose milliseconds as measurement is because of 10-millisecond computing [13]. In order to choose five digits is because IEEE has valid time testing until five decimal digits. 10000 random numbers and 10 trials are definitely not enough to test the efficiency. Few environments are not enough to verify the conclusion. Since time constraint, assume that 10 trials are able to give us an approximately time efficiency.

[13] 10-millisecond Computing. Beijing Academy of Frontier Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences. Mar 2017 https://pdfs.semanticscholar.org/df9b/1dbc3b62f5d082e515d92296a8ed11cd70b2.pdf

5.3.1 Logarithm

Calculate logarithm in different environments in milliseconds						
		Environment				
		3,4	3,5	4,6	4,7	IEEE
Trials	1	0.21136	0.21494	0.18110	0.25890	0.00001
	2	0.12037	0.22426	0.15217	0.24553	0.00001
	3	0.13061	0.12118	0.22251	0.22230	0.00001
	4	0.23316	0.11718	0.21261	0.30770	0.00001
	5	0.16139	0.15871	0.27556	0.23508	0.00001
	6	0.11993	0.31150	0.21181	0.30077	0.00001
	7	0.12035	0.12790	0.19343	0.43923	0.00001
	8	0.11479	0.12438	0.24727	0.28108	0.00001
	9	0.22881	0.14458	0.20279	0.30403	0.00001
	10	0.14054	0.12980	0.19567	0.31912	0.00001
Box Plot						
	Q1	0.12035	0.12526	0.19399	0.24887	0.00001
	MIN	0.11479	0.11718	0.15217	0.22230	0.00001
	MEDIAN	0.13558	0.13719	0.20730	0.29092	0.00001
	MAX	0.23316	0.31150	0.27556	0.43923	0.00001
	Q3	0.19886	0.20088	0.22004	0.30678	0.00001
Slows down of Unum by IEEE						
	Average	0.15813	0.16744	0.20949	0.29137	0.00001
	Log/IEEE	2.E+04	2.E+04	2.E+04	3.E+04	1.E+00

Figure 5.6, Logarithm functionn in different environments in millisececonds

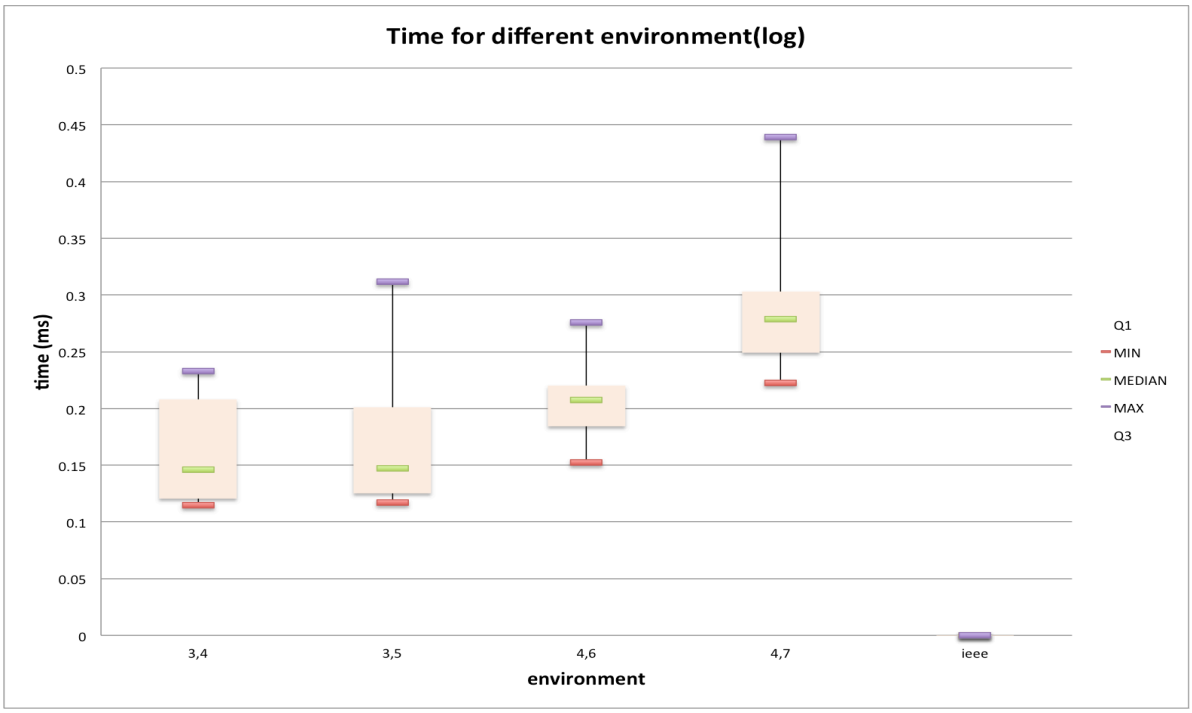


Figure 5.7, Time cost for logarithm function in different environment

The graph is a box plot for calculating logarithm. The box plot is highly visually effective way of viewing a clear summary of data.

We can observe that the box plots are comparatively short for {4,6}, {4,7} and IEEE, this suggests that the trials takes a relative steady time. This may due to since the precision increases; more representable numbers occurs, therefore, more computations allowed instead of skipping computations and returning infinity.

We can observe that box plot for Unum is much higher than IEEE. This suggests that Unum takes longer time than IEEE. This can be proved by the data from table which comparing Unum and IEEE. And from the table, we can see that on average, Unum takes more than 10000 times slower than IEEE. This may due to several reasons. Unum is represented as an interval (ubound). Thus, two endpoints (unum) are calculated separately. Besides, one endpoint (unum) is represented as a small interval (gbound) as well. Thus, double times to calculate endpoints (gnum) again. Calculating each endpoint (gnum) by using GMP. However, we do not actually know how long it takes for GMP arithmetic. Also, since glayer is a small interval (gbound), we return a small interval (gbound) rather than straightforward returning result after calculation. In logarithm function, we need to consider which one is the left endpoint of interval (gbound) and right endpoint. In particular, it takes more computations in power function. In addition, interval includes closure. Computation closure for interval takes few computations. Apart from that, Unum conversion takes amount of time as well. It needs to convert form hlayer to ulayer and to glayer, then convert it backwards. I assume that the conversion between ulayer and glayer takes lots of time. This is because; the idea of glayer is a scratchpad, which compute the arithmetic in higher precision than environment. Therefore, ulayer needs to convert the higher precision gbound to a representable ubound, as well as the closure. Furthermore, ulayer achieves an idea of unifying[14] to shorten the bits. This idea is to convert an interval (ubound) to a number (unum) if possible. This can be done when [1,1] returns to 1 or (1.1,1.3) returns to inexact 1.2. Thus, unify takes time. It is reasonable that Unum takes ten thousand longer than IEEE.

From the boxplot, we can see that time has some connection with environment. More precision in the environment, longer time it takes. It is rational. This is because, more precision in the environment means that more bits need to be calculated, hence longer time.

[14] John L. Gustafson. “The end of error” Section “Unifying a ubound to a single unum” P86

5.3.2 Exponential

Calculate exponential in different environments in milliseconds						
		Environment				
		3,4	3,5	4,6	4,7	IEEE
Trials	1	0.12083	2.19428	3.05184	4.49597	0.00001
	2	0.10588	2.16492	3.05082	4.51009	0.00001
	3	0.10909	2.17172	3.08429	4.47757	0.00001
	4	0.10574	2.17148	3.06770	4.47344	0.00001
	5	0.10783	2.17919	3.05931	4.55678	0.00001
	6	0.10783	2.17340	3.12628	4.47454	0.00001
	7	0.10742	2.16097	3.05831	4.42702	0.00001
	8	0.10684	2.17750	3.08016	4.46351	0.00001
	9	0.10092	2.16927	3.05750	4.43541	0.00001
	10	0.10910	2.16498	3.07829	4.42549	0.00001
Box Plot						
	Q1	0.10612	2.16605	3.05770	4.44244	0.00001
	MIN	0.10092	2.16097	3.05082	4.42549	0.00001
	MEDIAN	0.10762	2.17160	3.06351	4.47399	0.00001
	MAX	0.12083	2.19428	3.12628	4.55678	0.00001
	Q3	0.10877	2.17648	3.07969	4.49137	0.00001
Slows down of	Average	0.10815	2.17277	3.07145	4.47398	0.00001
Unum by IEEE	Exp/IEEE	1.E+04	3.E+05	4.E+05	5.E+05	1.E+00

Figure 5.8, Exponential function in different environments in milliseconds

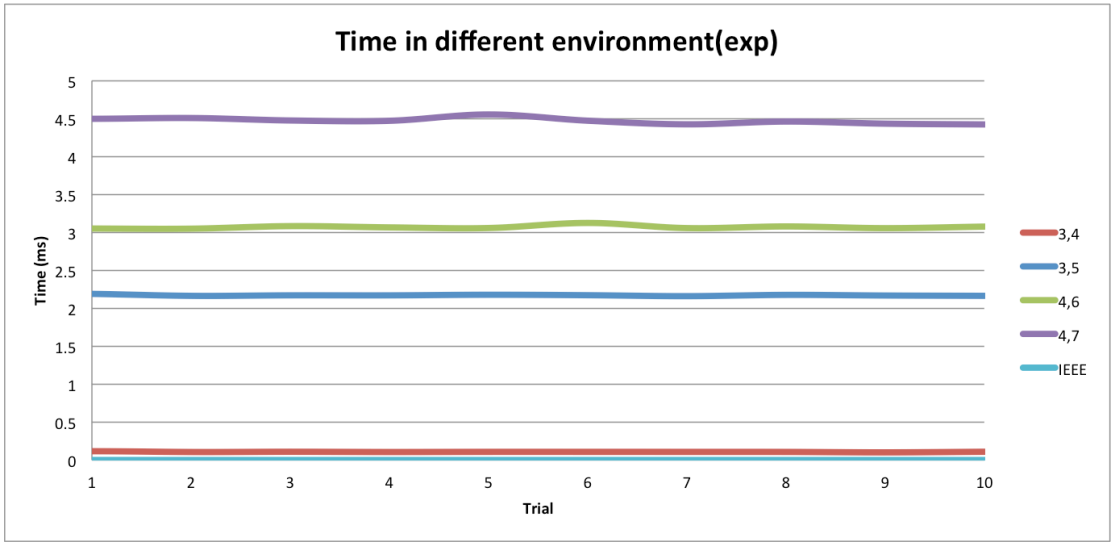


Figure 5.9, Time cost for exponential function in different environment in line chart

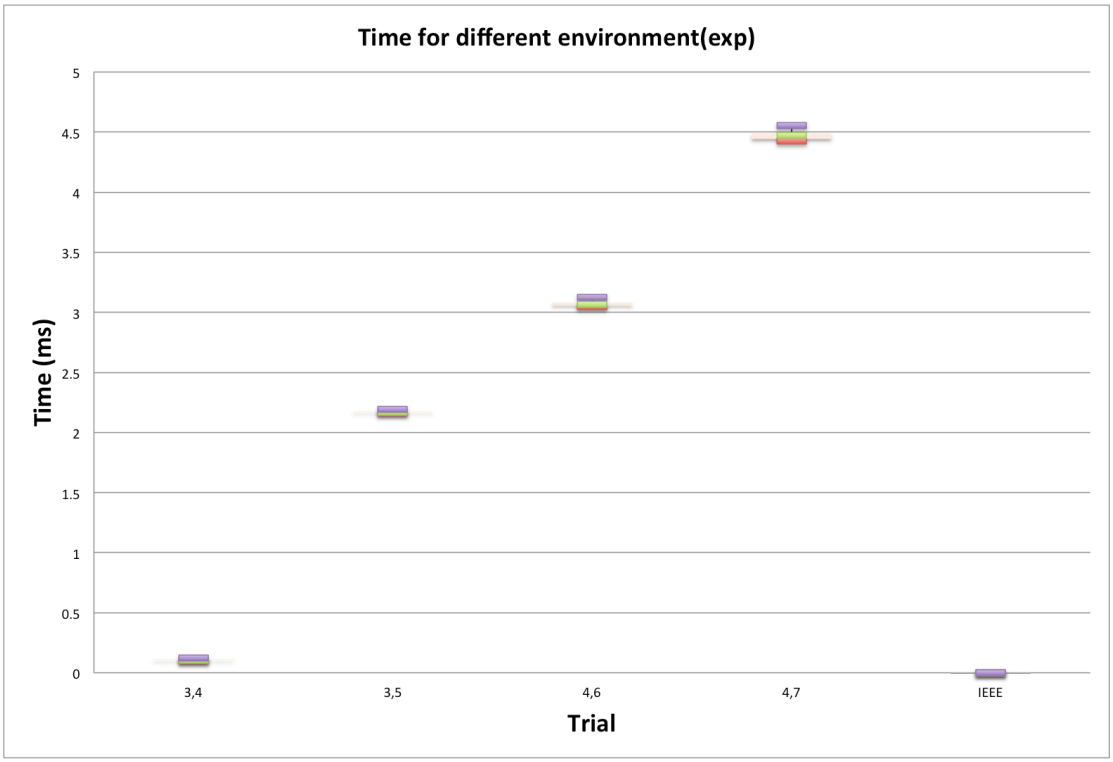


Figure 5.10, Time cost for Exponential function in different environment

We can observe that all the box plots are comparatively short. This suggests that the time for calculating by exponential takes a steady or nearly constant time. This can be also proved by the line chart. We can see that in the line chart, all the lines are steady without any floating rate. Apart from {3,4} box plot in Unum, others are much higher than IEEE. This means that except {3,4}, others environment takes longer than IEEE. This may due to since the precision increases; thus more bits. Due to more bits, more representable numbers occurs, therefore, more computations allowed instead of returning infinity. Hence, longer time.

5.3.3 Power

Calculate power in different environments in milliseconds						
		Environment				
		3,4	3,5	4,6	4,7	IEEE
Trials	1	13.40240	14.17750	19.85850	38.17790	0.00004
	2	13.49180	15.23470	20.30090	39.66120	0.00004
	3	13.57140	14.58120	20.42300	59.13860	0.00004
	4	12.60500	15.03090	21.41580	39.88540	0.00004
	5	12.82130	14.83100	19.29680	47.32710	0.00005
	6	13.12350	15.24050	20.55630	41.03270	0.00004
	7	13.56250	14.51040	21.09330	38.42530	0.00004
	8	13.78810	14.54890	20.22850	36.49940	0.00004
	9	13.54340	14.45260	21.43850	41.25100	0.00005
	10	12.91440	13.93420	19.71820	39.13560	0.00004
Box Plot						
	Q1	12.96668	14.46705	19.95100	38.60288	0.00004
	MIN	12.60500	13.93420	19.29680	36.49940	0.00004
	MEDIAN	13.44710	14.56505	20.36195	39.77330	0.00004
	MAX	13.78810	15.24050	21.43850	59.13860	0.00005
	Q3	13.55773	14.98093	20.95905	41.19643	0.00004
Slows down of Unum by IEEE						
	Average	13.28238	14.65419	20.43298	42.05342	0.00004
	Pow/IEEE	3.E+05	3.E+05	5.E+05	1.E+06	1.E+00

Figure 5.11, Power function in different environments in milliseconds

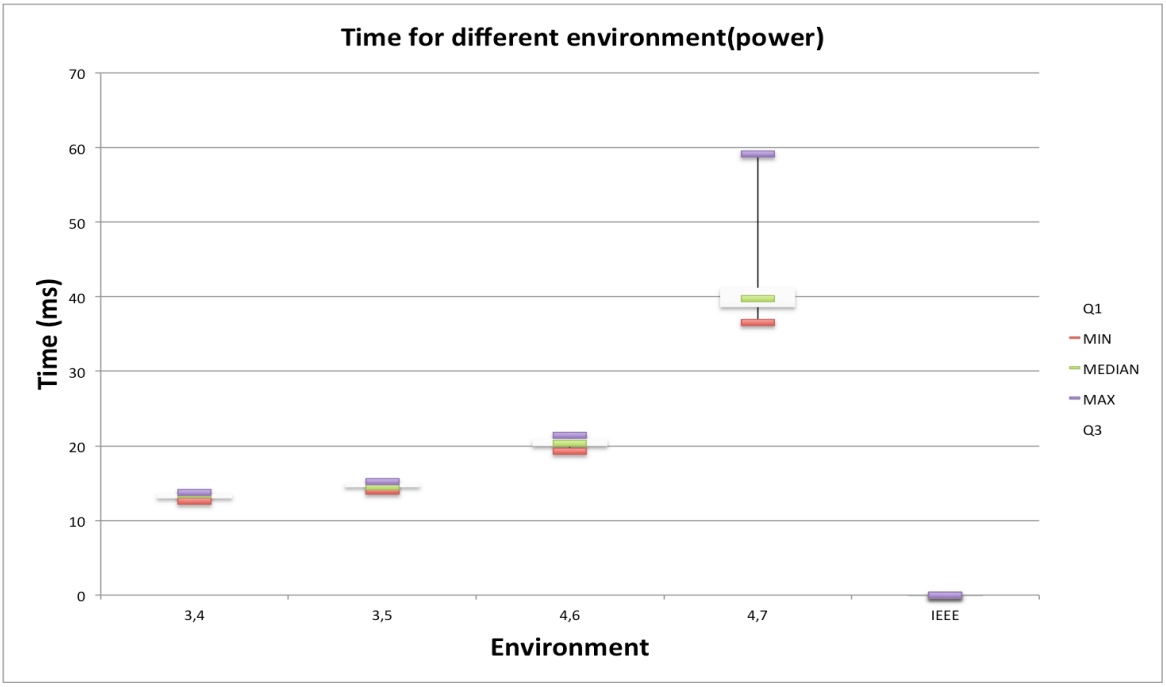


Figure 5.12, Time cost for power function in different environment

The box plot of {4,7} indicates it is a right skewed, which the top whisker is much longer than the bottom whisker ad the line is gravitating towards the bottom of the box. In other word, the maximum is close to an outlier. This may due to if there are any threads and processes interrupt (delay) the computations.

We can observe that in small precision, like {3,4}, fraction bits increase; time increases slowly. A great increase in time for large precision, like {4,6}. The difference between {3,4} and {3,5} is 16 (2^5-2^4) bits in fraction, and the time difference is on average 1.4msec. Thus, on average, to compute power in {3,5} environment takes 1.4msec longer than {3,4}. In contrast, the difference between {4,6} and {4,7} is 64 (2^7-2^6) bits in fraction, and the time difference is on average 21.7 msec. 64 bits is 4 times 16 bits, but time is way more than 4 times, about 15.5 times. Therefore, as fraction bits increase, time increases, moreover the increasing speed in regards to the fraction size.

5.3.4 Unum and IEEE

Slows down of Unum functions comparing to IEEE				
	3,4	3,5	4,6	4,7
Log/IEEE	1.52E+04	1.61E+04	2.02E+04	2.80E+04
Exp/IEEE	1.26E+04	2.52E+05	3.57E+05	5.20E+05
Pow/IEEE	3.02E+05	3.33E+05	4.64E+05	9.56E+05

Figure 5.13, Slow down of Unum functions comparing to IEEE

By summarizing the data together, we can see that generally logarithm function in Unum is 10000 times slower than IEEE, exponential and power functions in Unum is 100000 times slower than IEEE. Since power function is implemented by using exponential function, time to calculate power is related to exponential. Therefore, time to calculate exponential function increases, time to calculate power function increases.

6. Conclusion and Future Work

In regard to Unum obeying the associative law, logarithm and exponential, power and root is inverse function to each other, they will not necessarily give the exactly same answer with different computation sequence, but the result always correct, Even though the resulting intervals are not exactly same, but the correct answer always be contained. But as trade-off, unum is several thousands slower than IEEE. The more precision bits in the unum environment, the slower it would be. The rate of speed is regard to the precision bits as well. More precision bits in the unum environment, the slower the speed is. The efficiency needs to be improved to have more performance.

There are some limitations on calculating by power function. First limitation is, in mathematic, it does exist some valid values for negative number to the power of non integer. For example, $-32^{0.2} = -2$. But in my implementation, it does not support this as well as in IEEE. But, I think that it can be solved by implementing complex number[15]. Complex number can calculate the exact value for any negative base. Second limitation is, unum returns an interval for that number that cannot be represented. But the power to an interval returns a slightly different result under some circumstances, especially in negative base. For example, assume that 10 is not representable in unum environment, and hence it is represented by (9,11). -1^{10} should be 1, however, since the exponent is an interval and the base is a negative number, it returns NaN. But due to the limitation of time and knowledge of unum, this issue has not been fixed. It can be regard as a future discovery.

I think there are few approaches that can implement those functions to the true value. Shifting nth root algorithm [16] is a design I mentioned to solve power and exponential function. For now, the implementation of my power function is highly connected with logarithm and exponential function, therefore, efficiency and accuracy depends on the transcendental functions. This has been proved in discussion. Also, these functions are good at approximation, but not the exact result. In particular, it suffers from error magnification in exponential. Enlarge the interval provides correctness, but in return losing precision in relative width. Shifting nth root algorithm returns an exact result without any transcendental functions. The efficiency is doubtable, but it is the most accurate and precise result. Hence, it is attemptable approach. I have implemented part of this algorithm, but due to the limitation of time and knowledge of GMP. I am getting trouble with how to power a large number instead of returning infinity. In Unum, there is no rounding rule and most of the decimal number converts to a precise interval. For example, 2.1 in environment {3,4} becomes (2.0999755859375,2.100006103515625), and in rational form $(\frac{17203}{8192},\frac{68813}{32768})$. Therefore, $2^{2.1}$ is converted to $(\sqrt[8192]{2^{17203}}, \sqrt[32768]{2^{68813}})$. 2^{68813} is generally returned to infinity.

[15] A complex number is a number that can be expressed in the form a + bi, real and imagery number. https://en.wikipedia.org/wiki/Complex_number

[16] Shifting nth root algorithm. https://en.wikipedia.org/wiki/Shifting_nth_root_algorithm

Besides, $\sqrt[8192]{\ }$ is hard to compute as well. There is a prototype in unum library and a test in java. Java one uses BigDecimal and works. Unum one works for the numbers which do not exceed the maximum or minimum.

Golden Ratio improves efficiency Taylor Expansion for logarithm.[17] However, it encounters the same situation as power function. Taylor Expansion is a good approximation, but it is not the exact result. Therefore, there is an anonymous approach can solve the exact result.[18] The correctness of the approach can be proved. The efficiency is doubtable but it improves accuracy. But due to time limitation and the issue above, I did not implement this approach.

Further data mining and analysis need to be done. By testing and analyzing, I find the time to calculate function is around 1E4 and 1E5 slower than IEEE. As discussed, it may due to implementation of arithmetic operations; it may due to conversion between ulayer and glayer, as well as hlayer; it may due to GMP. There are several possibilities that cause the time slows down. If the purpose is to test the implementation of arithmetic functions, it should test the function individually without any conversions in Unum. Thus, it provides a more accurate and valid data on time efficiency. This could be done by profiling in LLVM [19], to see the exact amount of time it takes on functions. In addition, more data on relative width need to be discovered. The testing is about relative width of ubound for each function. However, there are conversions between each layer. Therefore, the conversions may loss precision. For example, $0.0625^{\frac{1}{4}} = 0.5$, assume that 0.0625 cannot be represented in unum, thus, in return it turns to (0.0624,0.0626) to glayer. Hence, $(0.0624^{\frac{1}{4}}, 0.0626^{\frac{1}{4}})$, and it calculates to (0.4997998,0.5001998) in glayer. But, since there is not representation for this interval, it converts to (0.49,0.51). As a result, it losses too many precisions. I think this could be done by testing the result on glayer separately. Furthermore, I think it is valuable to discover the relative precision loss between ulayer and glayer to gain more information on the trade-off between layers.

[17] Hiroaki Kimpara Sept 12,2011. Logarithmic spiral based on the golden ratio http://en.orion-metaphysics.com/wp-content/uploads/2012/06/Logarithmic-spiral-based-on-the-Golden-Ratio-Golden-Squared-Ratio-Silver-Ratio-and-Silver-Squared-Ratio-Apr22-12.pdf

[18] Solve logarithm by hand https://math.stackexchange.com/questions/820094/what-is-the-best-way-to-calculate-log-without-a-calculator

[19] Arun Ramachandran, Feb 11. Profiling Code with LLVM. https://medium.com/delta-force/profiling-code-with-llvm-f32c5292750a

Appendix A Project Description

The project will explore the unum library in LLNL version. The library supports creation of unums and printing of unum values. It also supports variable sized unums from a few bits up to thousands. Functions are available that convert between unums and primitive 'C' types. The relational operators, four arithmetic operators (add, subtract, multiply and divide), and square root are also implemented. Further relational operators need to be implemented for more arithmetic computations. The guess function, when applied to a unum after an arithmetic operation, will produce a rounded result much like a floating-point calculation.

The project report will contain:

- An introduction to the topic
- A background section which describes the unum format in LLNL and mathematical inductions
- An evaluation of logarithm, exponential and power functions in accuracy, precision and efficiency comparing with IEEE
- A description of the proposed algorithm for decoding
- A description of the implementation
- Experimental chapter which describes: the hardware used for experiments, the experiments done, and the results graphed and analyzed
- Conclusion, discussion, limitations and future work chapter

Appendix B Study Contract

Australian National University

INDEPENDENT STUDY CONTRACT PROJECTS

Note: Enrolment is subject to approval by the course convenor

SECTION A (Students and Supervisors)

UniID: _____U6063820_____

SURNAME: _____FENG_____ FIRST NAMES: _____SIDONG_____

PROJECT SUPERVISOR (*may be external*): _____Josh Milthorpe_____

FORMAL SUPERVISOR (*if different, must be an RSSCS academic*): _____ Josh Milthorpe _____

COURSE CODE, TITLE AND UNITS: __COMP3740, Project work in computing, 6 Units_____

SEMESTER ☑ S1 ☐ S2 YEAR: _____2018_____ Two-semester project (12u courses only): ☐

PROJECT TITLE:

Improvement of Unum computing

LEARNING OBJECTIVES:

- **Show understanding of performance and error characteristics of numerical software**
- **Show understanding of computing with Unum number format**
- **Develop both software and hardware to improve the efficiency of unum computing**
- **Effectively present research, methods and outcomes in oral, written, graphical forms**

PROJECT DESCRIPTION:

Develop the unum computing on both software and hardware level.

On the software level, improve the Unum Library built by LLNL. To find a way to make the computation more efficient.

On the hardware level, the hardware component needs to convert between Unum type and Binary expression, this costs a lot and slows down the computing. To make it more efficient, we can design our own hardware structure to do the computation without "wasting" time on converting expression. FPGA, doesn't have a fixed hardware structure, on the contrary it is programmable, which can be used to develop the unum computation.

Research School of Computer Science *Form updated Nov 2017*

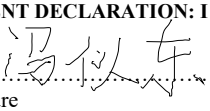
Australian National University

ASSESSMENT (as per the project course's rules web page, with any differences noted below).

Assessed project components:	**% of mark**	**Due date**	**Evaluated by:**
Report: name style: _____research report and software description_____ (e.g. research report, software description...,)	(min 45%) 45		(examiner) Eric McCreath
Artefact: name kind: _____user interface_____ (e.g. software, user interface, robot...,)	(max 45%) 45		(supervisor) Josh Milthorpe
Presentation : Improvement of Unum	(10%) 10		(course convenor) Peter Strazdins

MEETING DATES (IF KNOWN):

STUDENT DECLARATION: I agree to fulfil the above defined contract:



…….. ……05 / 03 / 2018……..
Signature Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project. I nominate the following examiner, and have obtained their consent to review the report (via signature below or attached email)

……..……..…….. ……..
Signature Date

Examiner:
Name: …… Signature ……
(Nominated examiners may be subject to change on request by the supervisor or course convenor)

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course convenor approval)

…….. ……..
Signature Date

Research School of Computer Science *Form updated Nov 2017*

Appendix C Description of artefacts

C.1 Submitted code files

Part of the following files are my own work:

- gbnd.h gbnd.c
- glayer.h glayer.c
- ubnd.c
- unum.c
- tbasic.c

My own work on java for finding root

- java/test.java

C.2 Experiment details

Analysis data is in the following files, and all of these tests are my own work:

- logarithm.xlsx
- exponential.xlsx
- power.xlsx

We evaluated our implementation on functions in 2.5GHz Intel core i5 Mac 2012.

Appendix D Readme file

D.1 Required libraries

Unum in LLNL version
    https://github.com/LLNL/unum
GMP library
    https://gmplib.org/
A supported C/C++ host compiler (gcc 5.4.0 was used)

D.2 Compilation

cd unum
./ configure
make check

D.3 Running

tbasic file in tests provides few correctness and time testing.

./tests/tbasic

Reference

1. Gustafson. John L. “The end of numerical error”. 06 June 2016
http://arith22.gforge.inria.fr/slides/06-gustafson.pdf

2. Kulisch, Ulrich W. “Up-to_date Interval Arithmetic from closed intervalsto connected sets of real numbers”. Institut tur Angewandte und Numberische Mathematik – Karlsruhe Institutue of Technology (KIT), Germay. Feb 2015
http://www.math.kit.edu/iwrmm/seite/preprints/media/preprint%20nr.%2015-02.pdf

3. The GNU Multiple Precision Arithmetic Library. April 2018
https://gmplib.org/

4. R.W.Hamming. Numerical Methods for scientists and engineers, Second edition, Dover Publications. 1973
https://www.amazon.com/Numerical-Methods-Scientists-Engineers-Mathematics/dp/0486652416

5. Apache Software Foundation. Apache Commons Math. 2018
https://github.com/apache/commons-math

6. Oracle and/or its affiliates. IEEE math library. May 2012
https://github.com/openjdk-mirror/jdk7u-jdk/tree/f4d80957e89a19a29bb9f9807d2a28351ed7f7df

7. Michael A. Gottlieb and Rudolf Pfeiffer. Algebra. California Institute of Technology. 2013
http://www.feynmanlectures.caltech.edu/I_22.html#Ch22-T2

8. Nicol N. Schraudolph. A Fast, Compact Approximation of the Exponential Function. March 13, 1998
https://pdfs.semanticscholar.org/35d3/2b272879a2018a2d33d982639d4be489f789.pdf

9. Gourdon, X. and Sebah, P. "Newton's Iteration."
http://numbers.computation.free.fr/Constants/Algorithms/newton.html

10. Doug Glasshof. Taylor Polynomials. University of Nebraska – Lincoln. July 2006
https://digitalcommons.unl.edu/cgi/viewcontent.cgi?referer=https://www.google.com.au/&httpsredir=1&article=1014&context=mathmidexppap

11. J.B.Young. Mathematical Methods, Fast Course. 2009
https://www-diva.eng.cam.ac.uk/lecture-notes/part-ia-lecture-notes/mathematics/paper-4-professor-davidson/1a-maths-functions.pdf

12. William Y.X. Zou. A New Method for Approximating Logarithms with kth Order. Pierre Elliott Trudeau High School. April 30, 2014
https://www.siam.org/students/siuro/vol7/S01325.pdf

13. A simple method for finding the roots of numbers. Hudson Mohawk Valley Area Mathematics Conference,Albany, New York. March 2013
http://planetmath.org/asimplemethodforfindingtherootsofnumbers

14. Rick Regan. Ratio of bits to decimal digits. Jan 2013
http://www.exploringbinary.com/ratio-of-bits-to-decimal-digits/

15. Guillaume Da Graçca, David Defour. Implementation of float-float operators on graphics hardware. 29 Mar 2006
https://hal.archives-ouvertes.fr/file/index/docid/63356/filename/float-float.pdf

16. Andrew Thall, Alma College. Extended-Precision Floating-Point Numbers for GPU Computation. March 15, 2007
http://andrewthall.org/papers/df64_qf128.pdf

17. Wikipedia. Natural Logarithm. May 2018
https://en.wikipedia.org/wiki/Natural_logarithm

18. Wikipedia. e (mathematical constant). May 2018
https://en.wikipedia.org/wiki/E_(mathematical_constant)

19. Wikipedia. Shifting nth root algorithm. April 2016
https://en.wikipedia.org/wiki/Shifting_nth_root_algorithm