About mathematical function's accuracy (1.2.0)

My application's all calculation process is by homemade.
Addition/subtraction/multiplication/division is done in decimal or fractional format.
No calculation error occurs in the limit of 16-digits.
And it has mode that it automatically expands the calculation length to 34-digits when the result exceeds 16-digits.
However, mathematical function like sin, log returns approximate calculation result.
It can't return true answer.
All other mathematical function process in the world, as far as I know, can't return a perfect right answer, too. You who try to read this document, surely know it.
This document is for persons who take an interest in calculation accuracy of this scientific calculator. (I am included, too.)

In 16 significant digits numeric of my process, the difference exists only in last 16th-digit and don't invade 15th-digit. (In my test)
I did a comparison with the calculation processes which Intel CPU and ARM CPU provided with. The result is below.
My process needs 64bits to hold value. It is same as 'double'. So I compared with it.
It is no doubt that my process loses the comparison with 'long double', bit amount is double of 'double'.
Both CPU double sometimes arises the difference of 15th digit.
In conclusion, my process wins to 'double' in calculation accuracy.
Electronic equipment scientific calculator is generally lower accuracy.
There may be higher accuracy equipment. But I don't equip the financial power and labor to confirm all.

Table downward is test result and my assessment. Below lines are annotation.
- Trigonometric function input angle is degree. In practical usage, input unit is almost degree. So I think degree is better as a comparison from user side.
  My trigonometric function lets degree as an input angle unit and it calculates in degree value as well as possible.
  In 'double', function input angle unit is radian only, so before calculation input value is converted from degree to radian.
- It is natural that 'long double' calculation result is more accurate, so it is used as reference accurate value in error detection.
  However, long double', conversion to degree for trigonometric function input already remains an error.
  In the doubtful case, I conformed the result using external more accuracy calculator (Windows XP calculator).
- Fact data is too large. So I make different summary files.
- Somehow arm64/armv7 sometimes returns subtly different value. So both results are shown.

< sin >
***test1 :** Calculation in the interval -360 ~ 360 degree, the each value shifted by 1 degree.
- Intel double is less accurate at 180, -180, -360 neighborhood degree. (6points)
- arm64/armv7 double is less accurate at -180, -360 neighborhood degree. (3points)
- For example, in -359 case, sin of 360 - 359 = 1degree is necessary to calculate. 'double' do this substraction in radian and error occurs.
  Just the error remains in result, I think.
  Not clear things still remains, difference between 359 and -359. (Exept for negative sign, all must be the same.)
  But I do not inquire it because the 'double' accuracy validation is not my aim.

***test2 :** Calculation in the interval 0 ~ 90 degree, the each value shifted by i * 90/10000 (i = 0 ~ 10000) degree.
- Neither has big error.

< cos >
***test1 :** Calculation in the interval -360 ~ 360 degree, the each value shifted by 1 degree.
- intel double is less accurate at 270, 90, -90, -270 neighborhood degree. (6points)
- arm64/armv7 double is less accurate at 270, -270 neighborhood degree. (2points)
- The reason seems to be the same as shown ever.

***test2 :** Calculation in the interval 0 ~ 90 degree, the each value shifted by i * 90/10000 (i = 0 ~ 10000) degree.
- intel double is less accurate at 90 neighborhood degree. (51points)
- arm64/armv7 double is less accurate at 90 neighborhood degree. (56points)
- 'sin' calculates 0 neighborhood result from 0 neighborhood value. 'cos' calculates 0 neighborhood result from not 0 neighborhood value.
  Error is caused in substraction, I think. (See explanation downward)

< tan >
***test1 :** Calculation in the interval -360 ~ 360 degree, the each value shifted by 1 degree.
- intel double is less accurate at 270, 180, 90, -180, -270, -360 neighborhood degree. (11points)
- arm64/armv7 double is less accurate at 270, 180, -270, -360 neighborhood degree. (5points)
- The reason seems to be the same as shown ever.

***test2 :** Calculation in the interval 0 ~ 90 degree, the each value shifted by i * 90/10000 (i = 0 ~ 10000) degree.
- intel double is less accurate at 90 neighborhood degree. (47points)
- arm64/armv7 double is less accurate at 90 neighborhood degree. (54points)
- The reason seems to be the same as shown ever.

< sin^{-1} >
***test1 :** Calculation in the interval -1 ~ 1, the each value shifted by i/10000 (i = -10000 ~ 10000) step. (Radian Result)
- Neither has big error.

< cos^{-1} >
***test1 :** Calculation in the interval -1 ~ 1, the each value shifted by i/10000 (i = -10000 ~ 10000) step. (Radian Result)
- intel double is less accurate at 1 neighborhood and smaller. (14points)
- arm64/armv7 double is less accurate at 1 neighborhood and smaller. (14points)
- 'sin^{-1}' calculates 0 neighborhood result from 0 neighborhood value. 'cos^{-1}' calculates 0 neighborhood result from not 0 neighborhood value.
I think that it is the reason error remains.

< tan^{-1} >
***test1 :** Calculation in the interval -1 ~ 1, the each value shifted by i/10000 (i = -10000 ~ 10000) step. (Radian Result)
- Neither has big error.

***test2 :** Calculation in the interval ~ -1 1 ~, the each value shifted by 10000/i (i = -10000 ~ 10000) step. (Radian Result)
- Neither has big error.

< log10 >
***test1** : Calculation in the interval 0 ~ 10, the each value shifted by i/1000 (i = 1 ~ 10000) step.
- intel double is less accurate at 1 neighborhood and larger. (5points)
- arm64/armv7 double is less accurate at 1 neighborhood and larger. (5points)
- 1 neighborhood can be converged to the level error goes out of the range by Maclaurin's expansion. I think it may be the soft point of pursuit accuracy.

***test2 :** Calculation in the interval 0 ~ 10000, the each value shifted by i (i = 1 ~ 10000) step.
- Neither has big error.

< loge >
***test1 :** Calculation in the interval 0 ~ 100, the each value shifted by i/100 (i = 1 ~ 10000) step.
- Neither has big error.

< log2 >
***test1 :** Calculation in the interval 0 ~ 100, the each value shifted by i/100 (i = 1 ~ 10000) step.
- Neither has big error.

< 10^ >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- intel double is less accurate at the interval -99.99 ~ -64.01, 64.01 ~ 99.99 thoroughly. (2880points)
- arm64/armv7 double is less accurate at the interval -99.99 ~ -64.01, 64.01 ~ 99.99 thoroughly. (2658/2659points)
- In decimal calculation, input integer part is equal to the output exponential part, and only the fraction decimal part calculation is necessary.
  So such kind error doesn't occur even if the input value becomes larger.
  In binary calculation like double, exponential part is powered by 2. 10 -> 2 conversion error must be amplified by power calculation.

< e^ >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- intel double has no big error.
- arm64/armv7 double is less accurate at the interval -99.82 ~ -64.57, 98.32 thoroughly. (135points)
 - I can't decide the different reason well. But I guess same kind amplification of error as above.

< 2^ >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- Neither has big error.

< sinh >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- Neither has big error.

< cosh >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- Neither has big error.

< tanh >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- Neither has big error.

< $\sinh^{-1}$ >
***test1 :** Calculation in the interval -100 ~ 100, the each value shifted by i/100 (i = -10000 ~ 10000) step.
- Neither has big error.

< $\cosh^{-1}$ >
***test1 :** Calculation in the interval 1 ~ 100, the each value shifted by i/100 (i = 100 ~ 10000) step.
- Neither has big error.

< $\tanh^{-1}$ >
***test1 :** Calculation in the interval -1 ~ 1, the each value shifted by i/10000 (i = -10000 ~ 10000) step.
- intel double is less accurate at 0.99xx neighborhood. (5points)
- arm64/armv7 double is less accurate at 0.99xx neighborhood. (5points)
- I can't guess the difference reason well.

My process had ever included more error than now. I investigated and improved error cause points by every function and achieve present accuracy.
The reason of the error is always same. Significant digits are lost in subtraction of two similar values.
For example, thinking two similar values of 5-digits, subtraction between 1.0000 and 0.99965 is 0.00035.
If counting upper 000, it is 5-digits. But significant digits becomes 2-digits. In 5-digit result 3.5000x10^-1, bottom 3-digits includes error. Accuracy is lost in a flash.
The pristine solution of it is extension of calculation registor. But it instantly increases calculation complexity and necessary process time.
Without prolonging calculation bit amount, by changing the content of calculation or detailed countermeasure, I upgraded my process accuracy as well as possible.
The primary cause of 'double power calculation error' is presumed in conversion from input decimal to calculation binary, and to output decimal.
So when input and output forms are binary, the accuracy surely becomes better.
But thinking the usage of user input and display, the calculation numeric is decimal. I think using decimal in testing is not pork barrel to my process.

I must mention calculation speed. My software process cannot match 'double' hardware process. It is my complete defeat.
There is no possibility to win by a little try of speed up. It is the reason I must stick to accuracy.
But I think, when my process is implemented by hardware, the speed could be almost same level as double.

I comment necessary complexity, too. It is sure that Intel double uses very large size tables, from program list on Intel site.
ARM process inside is not quite certain. But I guess it uses the same level large table, because error pattern is very resemble.
My process is modification of algorithm used in scientific calculator. The program is very compact and the table inside is very small.
If my guess is right, the comparison of (calculation accuracy / necessary complexity) is my overwhelming victory.

If you need is only 16-digits accuracy, it is better to calculate in 'long double' and show the cut result until 16th-digits.
Necessary bit size for calculation becomes double, but the speed "long double" is still faster than my process, by its hardware calculation assist.
And there is already well known decimal calculation process 'Big Decimal'.
Against these, I add merit to my decimal calculation process with following items.
- Sign to show whether accurate result or not. Hybrid system of fraction, square root number calculation.
I think meaning of my process's existence is already sufficient in present version.