# 算術演算関数の精度に関して (1.2.0)

　私のアプリの演算関数は全て内製です。
加減乗除は常に10進数あるいは分数の形で行なっており、数値の保持桁数16桁を超えない限り誤差が生じない様になっており、また16桁を越えた事を検出すると保持桁数34桁まで拡張するモードも備えます。
しかし sin, log 等の算術関数は常に16桁で近似計算をしており完全に正しい値は返しません。世間に存在する他の算術関数処理も完全に全桁正しい値を返す物は存在しません(私の知る限りですが)。
この事は、今この書類を読もうとしている様な方ならご理解されているとは思います。
よって、この関数電卓の算術関数演算精度はいかほどなのか興味を持たれる方 (私も含まれます) の為にこの資料を作っています。

　結論から述べると数値の保持桁数16桁のうち16桁目に誤差が乗るレベルであり、15桁目より上へ生じる誤差は検出されない状態になっています。(私の行ったテストによる)
下記に intel CPU 及び arm CPU に備える演算処理との比較を示します。
私の処理の値の保持に必要な bit数は 64bits で double (倍精度浮動小数点数) と同じであり、これと比較します。保持サイズが倍になる long double には精度で負ける事は明白です。
double はどちらの CPU にしろ色々15桁目の誤差が検出されており、私の処理は double より演算精度が上であると結論できます。
また電子機器の関数電卓は一般的にはもっと精度が下です。上のものもあるかもとは思いますが、全て買い集めて確認する財力と労力は私には存在しません。

　下記に行ったテスト内容とその評価をまとめます。注釈をここでまとめておきます。
・三角関数の入力は度で計算しています。実際の用途では入力はほとんど度単位であり、ユーザー側から見た比較としては度が良いと考えます。
　私の処理では角度単位は度も許し、度のまま計算できる所は度で計算します。double では角度単位は radian のみであり、度から radian に変換して計算しています。
・long double で計算した値がより正確である事は当たり前であるので、これを正確な値とし誤差検出に使用しています。
　ただし、三角関数の度においては変換誤差により入力に誤差が残るので long double にも誤差を含む事もあり、この場合外部の精度のいい電卓 (Windows XP の電卓) を使用して確認しています。
・実際のデータは長くなるので別ファイルにまとめます。
・arm64/armv7 はなぜか微妙に値がずれている事があるので両方結果を示します。

**< sin >**
***test1** : -360 ～ 360 度まで1度づつ変えた値の計算
・intel double は 180, -180, -360度付近 少し精度が悪くなっている (6points)
・arm64/armv7 double は -180, -360度付近 少し精度が悪くなっている (3points)
・理由は例えば -359 なら 360 - 359 = 1 度を計算する時に double は radian で計算するので誤差が残り、そのまま結果の誤差として残ってしまうと思われる。
ただ 359 と -359 の違い等(普通符号が違うだけで全く同じ値になると思われる)不明な事が多いが、double の精度は単に比較対象であり、この確認が目的ではないので詮索しない。

***test2** : 0 ～ i * 90/10000度づつ (i = 0 ～ 10000) 90度まで変えた値の計算
・何も大きな誤差なし

**< cos >**
***test1** : -360 ～ 360 度まで1度づつ変えた値の計算
・intel double は 270, 90, -90, -270度付近 少し精度が悪くなっている (6points)
・arm64/armv7 double は 270, -270度付近 少し精度が悪くなっている (2points)
・理由は上に同じと思われる。

***test2** : 0 ～ 90度まで i * 90/10000度づつ (i = 0 ～ 10000) 変えた値の計算
・intel double は 90度付近 少し精度が悪くなっている (51points)
・arm64/armv7 double は 90度付近 少し精度が悪くなっている (56points)
・sin は 0 近傍の入力から 0 近傍の解を計算するが、cos は 0 でない値から 0 近傍の解を計算するので引き算の誤差(下方に説明あり)が残ると思われる。

**< tan >**
***test1** : -360 ～ 360 度まで1度づつ変えた値の計算
・intel double は 270, 180, 90, -180, -270, -360度付近 少し精度が悪くなっている (11points)
・arm64/armv7 double は 270, 180, -270, -360度付近 少し精度が悪くなっている (5points)
・理由は上に同じと思われる。

***test2** : 0 ～ 90度まで i * 90/10000度づつ (i = 0 ～ 10000) 変えた値の計算
・intel double は 90度付近 少し精度が悪くなっている (47points)
・arm64/armv7 double は 90度付近 少し精度が悪くなっている (54points)
・理由は上に同じと思われる。

**< $sin^{-1}$ >**
***test1*** : -1 ～ 1 まで i/10000度づつ (i = -10000 ～ 10000) 変えた値の計算 (結果は Radian)
・何も大きな誤差なし

**< $cos^{-1}$ >**
***test1*** : -1 ～ 1 まで i/10000度づつ (i = -10000 ～ 10000) 変えた値の計算 (結果は Radian)
・intel double は 1 付近 少し精度が悪くなっている (14points)
・arm64/armv7 double は 1 付近 少し精度が悪くなっている (14points)
・$sin^{-1}$ は 0 近傍の入力から 0 近傍の解を計算するが、$cos^{-1}$ は 0 でない値から 0 近傍の解を計算するので誤差が残ると思われる。

**< $tan^{-1}$ >**
***test1*** : -1 ～ 1 まで i/10000度づつ (i = -10000 ～ 10000) 変えた値の計算 (結果は Radian)
・何も大きな誤差なし

***test2*** : ～ -1, 1 ～ まで 10000/i度づつ (i = -10000 ～ 10000) 変えた値の計算 (結果は Radian)
・何も大きな誤差なし

**< log10 >**
***test1*** : 0 ～ 10 まで i/1000 づつ (i = 1 ～ 10000) 変えた値の計算
・intel double は 1 より少し大きいあたり 少し精度が悪くなっている (5points)
・arm64/armv7 double は 1 より少し大きいあたり 少し精度が悪くなっている (5points)
・1 近傍はマクローリン展開で計算すれば大きな誤差を含まないレベルまで早く収束するはずであり、単に精度の追求が甘いのでは考えられる。

***test2*** : 0 ～ 10000 まで i づつ (i = 1 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< loge >**
***test1*** : 0 ～ 100 まで i/100 づつ (i = 1 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< log2 >**
***test1*** : 0 ～ 100 まで i/100 づつ (i = 1 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< 10^ >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・intel double は -99.99 ~ -64.01, 64.01 ~ 99.99  あたりまんべんなく精度が悪くなっている (2880points)
・arm64/armv7 double は -99.99 ~ -64.01, 64.01 ~ 99.99  あたりまんべんなく精度が悪くなっている (2658/2659points)
・10進数で計算すると整数部を指数部に入れ、小数部のみを計算すれば良いので入力の数がいくら大きくなっても演算精度に違いは生じない。
　double は 2進数で計算するので指数部は2進数のべき乗であり、10 -> 2進数変換誤差がべき乗で増幅されてしまう模様である。

**< e^ >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・intel double は大きな誤差なし
・arm64/armv7 double は -99.82 ~ -64.57, 98.32  あたりまんべんなく精度が悪くなっている (135points)
・違いの生ずる理由はよくわからない。

**< 2^ >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< sinh >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< cosh >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< tanh >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< $\sinh^{-1}$ >**
***test1** : -100 ～ 100 まで i/100 づつ (i = -10000 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< $\cosh^{-1}$ >**
***test1** : 1 ～ 100 まで i/100 づつ (i = 100 ～ 10000) 変えた値の計算
・何も大きな誤差なし

**< $\tanh^{-1}$ >**
***test1** : -1 ～ 1 まで i/10000 づつ (i = -10000 ～ 10000) 変えた値の計算
・intel double は 0.99xx 付近精度が悪くなっている (5points)
・arm64/armv7 double は 0.99xx 付近精度が悪くなっている (5points)
・違いの理由はよくわからない。

私の処理も当初はもっと誤差を含んでいましたが、各関数毎に誤差の原因を追求・改善する事により現在の精度を達成しています。
誤差が大きくなる場合の基本的な原因について述べておくと、同程度の値の引き算で 0 近傍の値になる時有効桁が失われる為です。
例えば有効桁5桁の2つの同じぐらいの数 1.0000 と 0.99965 を引き算すると結果は 0.00035 となります。
これは上位の 000 を含むと5桁ですが、有効桁は2桁です。結果を5桁表示すると 3.5000x10^-1 の内下3桁は誤差が含まれた数値となり、一気に精度が失われます。
演算に使用するレジスタの長さを増やすのが正当な解決法です。しかしそうすると演算の複雑度と演算時間が一気に増えてしまいます。
私の処理では長さを延ばす事なく演算の内容を変えたり色々細かく対処する事により、演算精度を可能な限り保たせる様にしています。
　また double のべき乗での誤差の原因は、入力数値及び表示数値が10進数で計算は2進数である事による変換誤差と推測されます。
入力と結果が2進数のままであると精度は改善するであろう事は明白ですが、ユーザーが入力した数値を用いて計算しユーザーに表示する用途であると、
数値は10進数であり現在のテストは決して私の有利な方に振った比較ではないと考えます。

　演算速度についても述べておくと、ハードで処理する 各CPU double にソフトで敵うはずもなく私の完敗であり、少しぐらい高速化を計っても勝つ要素はありません。
精度にはこだわらねばならなかった所以です。
ただ私の処理もハード化したら double と速度はさして変わらない気は個人的にはしています。

計算に要する処理の複雑さについても述べておくと Intel double は Intel のサイトにあるプログラムと同様であると考えると非常に大きなサイズのテーブルを使っていると思われます。

ARM はいまいち処理内容が不明ですが、同様に大きなテーブルを使っているだろうと推測できます。誤差の出方が非常に似通っている為です。
私のアプリは従来関数電卓で使用されてるアルゴリズムの改良版で非常にコンパクトなものであり、テーブルサイズは非常に小さなものです。
そうだとすると (演算精度 / 要する複雑度) の比較では圧勝であると言えます。

ただ16桁の有効桁で精度があればいいのであれば long double で計算し、そのうち 16桁までに切って表示すればいいだけとも思えます。
値のサイズが倍になるだけで、やはり動作速度はハードでほとんど処理する為速いです。
また 10進数計算なら既に Big Decimal と言うものがあります。
これに対して、正確な値が保持されているか否かわかる様にする事により 10進数で計算するメリットを与え、また分数、平方根の付いた数等を交えた計算をできる様にした
ハイブリッドなシステムによりそれらに対する優位性を与えています。よって現状で私の処理の存在意義は十分あると考えています。