古典的誤差評価法に基づく4次以下の実係数代数方程式向け任意精度ソルバの開発

Development of Arbitrary Precision Solver based on Classical Error Estimation for Real Algebraic Equations of Degree less than or equal to 4

幸谷智紀*

Tomonori KOUYA*

Abstract: Most numerical algorithms are based on the discretization of original continous problems, and these algorithms are executed by using finite precision floating-point arithmetic operations. Therefore, the numerical results obtained throught such floating-point operations have two types of errors; theoritical errors derived from the discretization, and round-off errors derived from the finite precision floating-point arithmetic operations. A well-known a posteriori and *empirical* error estimation method is used to estimate errors separately by comparing several numerical results obtained under different conditions. We call this method "Classical Error Estimation (CEE)." We have proved experimentally that CEE-based algorithms employed for eigenvalue problems for symmetric matrices, nonlinear equations, and initial value problems for ordinary differential equations, provide efficient results. However, a large number of problems remain unsolved; we target the solving of such problems by the application of the CEE method. Real algebraic equations are one of the problems and they often appear in various fields. Equations of degree less than or equal to 4 can be solved through finit algebraic manipulations; a CEE-based arbitrary precision solver requires only the estimation of round-off errors. Therefore, they are suitable for verifying the CEE-based estimation of round-off errors. In this paper, we demonstrate that the CEE method can be applied to various real algebraic equations of degree less than or equal to 4, and this method is highly efficient in obtaining user-required precision numerical results.

1. 初めに

数値計算アルゴリズムの多くは，連続問題を離散化し，有限桁の浮動小数点数を用いることを前提としたものである。従って，得られた数値解には離散化に伴って発生した理論誤差と有限桁の浮動小数点演算に伴って発生した丸め誤差が含まれている。

古くから，多くの問題やアルゴリズムにおいては，条件を変えて複数の数値解を比較することにより，この二つを区別してそれぞれ事後的に評価することができるということが経験的に知られている [3]。我々はこのような経験的な誤差評価法を「古典的誤差評価法」と呼び，実対称行列の固有値問題 [7]，非線型方程式 [7]，常微分方程式の初期値問題に対してこれを適用し，多くの問題に対して的確な誤差評価が可能であることを示してきた。

しかしまだ対象とする問題は数多く残っている。特に実係数の代数方程式は応用範囲が広い。4次以下のものは有限回の代数演算で解が得られるため，丸め誤差のみ評価することで任意精度計算が可能になり，丸め誤差評価が適切に働いているかを確認するためには相応しい問題となる。そして，これを土台とすることにより，高次の代数方程式の任意精度計算も可能になる。

本稿において，我々はこの4次以下の実係数代数方程式に対して古典的誤差評価法を適用し，多くの問題に対して有効に働くことを示す。また，実装したアルゴリズムの数値的安定性の比較実験にも有用なものとなることも併せて示す。

2008年3月4日受理

*理工学部情報システム学科

2. 4次以下の代数方程式の解法

ここでは実装した実係数代数方程式の解法について，その概要のみを簡潔に述べる。3次，4次代数方程式の解公式は既存のもの [10] に基づいて実装した。後述する数値実験は全てこのアルゴリズムに基づいて実装したプログラムによって得られたものである。なお，本稿を通じて，虚数単位は $i = \sqrt{-1}$ で表わすことにする。

2.1 1次，2次方程式

1次方程式

$$a_1 x + a_0 = 0 \quad (a_1 \neq 0) \tag{1}$$

は

$$x = -a_0/a_1$$

とすればよい。2次方程式

$$a_2 x^2 + a_1 x + a_0 = 0$$

は多くの数値計算の教科書で述べられている通り，解公式

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}}{2a_2}$$

を桁落ちを避けるようにして計算する [5]。

2.2 Cardano 法 — 3次代数方程式の解公式

3次方程式

$$a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0 \ (a_3 \neq 0) \tag{2}$$

に対する代表的な解公式が Cardano 法である。この解公式を導く手順を大雑把に示す。

まず

$$x = y - \frac{a_2}{3a_3}$$

とし，式 (2) に代入して

$$y^3 + 3py + q = 0 \tag{3}$$

と式変形する。この時

$$\begin{aligned} p &= -\frac{a_2^2}{9a_3^2} + \frac{a_1}{3a_3} \\ q &= \frac{2a_2^3}{27a_3^3} - \frac{a_1 a_2}{3a_3^2} + \frac{a_0}{a_3} \end{aligned}$$

である。更に $y = u + v$ とし，かつ u と v は $uv = -p$ を満足するものとすると，式 (3) の y に代入して

$$u^3 + v^3 = -q$$

を得る。また $u^3 v^3 = -p^3$ なので，u^3, v^3 は 2 次方程式

$$z^2 + qz - p^3 = 0$$

の解である。従って

$$\begin{aligned} u^3 &= \frac{-q + \sqrt{q^2 + 4p^3}}{2} \\ v^3 &= \frac{-q - \sqrt{q^2 + 4p^3}}{2} \end{aligned}$$

である。これを満足するもののうち一つを $\hat{u}, \hat{v}$ とし，1 の複素数 3 乗根のうちどちらか一つを ω とすれば，$uv = -p$ を満足するものは

$$\begin{aligned} \beta_1 &= \hat{u} + \hat{v} \\ \beta_2 &= \omega\hat{u} + \omega^2\hat{v} \\ \beta_3 &= \omega^2\hat{u} + \omega\hat{v} \end{aligned}$$

である。従って式 (3) の解 $\beta_1, \beta_2, \beta_3$ を具体的に書くと

$$\begin{aligned} \beta_1 &= \hat{u} + \hat{v} \\ \beta_2 &= -\frac{1}{2}(\hat{u} + \hat{v}) - \frac{\sqrt{3}}{2}(\hat{u} - \hat{v})\,\mathrm{i} \\ \beta_3 &= -\frac{1}{2}(\hat{u} + \hat{v}) + \frac{\sqrt{3}}{2}(\hat{u} - \hat{v})\,\mathrm{i} \end{aligned} \tag{4}$$

となる。よって，これらを用いて元の 3 次代数方程式 (2) の解 α_i $(i = 1, 2, 3)$ は

$$\alpha_i = \beta_i - \frac{a_2}{3a_3} \quad (i = 1, 2, 3)$$

となる。

なお，3 次方程式の解のうち絶対値最小の解を精度良く求めるため，元の方程式の解の逆数を解として持つ代数方程式を再度解き直す実装も存在する [10] が，後述するように今回は精度不足の際のリトライを自動的に行うようにしているので，この方法は採用しなかった。それによる不具合も今の所発見されていない。

2.3 Ferrari 法 — 4 次代数方程式の解公式

4 次方程式

$$a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0 \ \ (a_4 \neq 0) \tag{5}$$

に対して歴史的に最初に提唱されたものとされているが Ferrari 法である。これも Cardano 法と同様に

$$x = y - \frac{a_3}{4a_4}$$

とおいて式 (5) に代入することにより

$$y^4 + py^2 + qy + r = 0 \tag{6}$$

となる。ここで

$$\begin{aligned} p &= -\frac{3a_3^2}{8a_4^2} + \frac{a_2}{a_4} \\ q &= \frac{a_3^3}{8a_4^3} - \frac{a_2 a_3}{2a_4^2} + \frac{a_1}{a_4} \\ r &= -\frac{3a_3^4}{256a_4^4} + \frac{a_2 a_3^2}{16a_4^3} - \frac{a_1 a_3}{4a_4^2} + \frac{a_0}{a_4} \end{aligned}$$

である。

$q = 0$ の時は直ちに因数分解でき

$$\left(y^2 - \frac{-p - \sqrt{p^2 - 4r}}{2}\right)\left(y^2 - \frac{-p + \sqrt{p^2 - 4r}}{2}\right) = 0$$

を y について解けばよい。

$q \neq 0$ の時は，式 (6) を

$$y^4 = -py^2 - qy - r$$

とし，この両辺に $y^2 z + z^2/4$ を加えて

$$\left(y^2 + \frac{z}{2}\right)^2 = (z-p)\left(y - \frac{q}{2(z-p)}\right)^2 + \frac{1}{4(z-p)}(z^3 - pz^2 - 4rz + 4pr - q^2) \tag{7}$$

と式変形する。さすれば右辺の 3 次式が $z^3 - pz^2 - 4rz + 4pr - q^2 = 0$ となる z を一つ見つければ，$q = 0$ の時と同様，直ちに因数分解でき

$$\left(y^2 + \frac{z}{2} - \sqrt{z-p}\left(y - \frac{q}{2(z-p)}\right)\right) \times \left(y^2 + \frac{z}{2} + \sqrt{z-p}\left(y - \frac{q}{2(z-p)}\right)\right) = 0 \tag{8}$$

を y について解けばよいことになる。

Ferrari 法の概要は以上であるが，実際にこれをそのまま実装すると，特に近接解の時には数値的不安定さを惹起することがある。それを回避する工夫が必要になると思われるが，今回は十分な解析が行えていないので，まだそこまでの改良は行っていない。また，後述する我々の任意精度計算アルゴリズムは，必要な精度に達するまでリトライを繰り返すため，多少の不安定性があってもそれをカバーすることが可能であり，最終結果への精度面での影響は今の所見つかっていない。

3. 古典的誤差評価法について

伊理 [3] は，第 4 章「たった一回だけの計算なんて」(pp.20-25) において，「系統的に計算方法を変えて 2 回以上計算すると非常に有用な情報が得られる」と述べ，理論誤差と丸め誤差の大きさを評価する方法を具体例をもとに述べている。ことに後者は有限桁の浮動小数点演算を用いる限り逃れ得ないものであるが，厳密な丸め誤差限界を得ようとすると過大評価になりがちになる。しかし，実用的には同じアルゴリズムを計

算桁数を変えて実行して得た近似値の差を取ることで，短い計算桁数で計算した近似値に含まれる丸め誤差を評価することができる。

本稿ではこの考え方に基づき，10 進 S 桁の近似値 $x^S \in \mathbb{R}$ に含まれる理論 (打ち切り) 誤差 $T(x^S)$ と丸め誤差 $R(x^S)$ を下記のように評価する方法を「古典的誤差評価法」と呼ぶことにする。

理論誤差の評価 理論誤差が丸め誤差より優越している地点の誤差を理論誤差の評価値 $T(x^S)$ として使用

丸め誤差の評価 同じアルゴリズムを実行し，長い桁数 L ($> S$) で計算した結果 x^L を真値として用い，それより短い桁数 S による結果 x^S に含まれる丸め誤差の評価値 $R(x^S)$ を

$$R(x^S) = |x^L - x^S| \tag{9}$$

とする

本稿ではこれに基づき，x^S に含まれる絶対誤差の評価値 $E(x^S)$ を

$$E(x^S) = \max\left(T(x^S), R(x^S)\right) \tag{10}$$

とする。実際に適用する時には，$T(x^S) \approx R(x^S)$ となるような，必要最小限の桁数で計算することを仮定している。

上記の評価法のうち，丸め誤差の評価は問題やアルゴリズムによらず共通である。同じアルゴリズムを桁数を変えて実行するだけであるから，x^L の計算はあまり計算時間が変わらない程度に計算桁を増やし，x の計算と並列実行することが可能である。

これに対して理論誤差の評価法は，アルゴリズムが依拠する近似理論に応じたものが必要となるが，今回は有限回の代数演算のみで終了する処理のみなので，詳細については参考文献 [7] に譲り，解説は割愛する。

4. 要求精度を持つ近似値を得るための自動計算アルゴリズム

以上の古典的誤差評価法を用いて，ユーザが指定した精度桁数を持つ近似値を得るためには，計算桁数も当然それ以上必要となる。しかし計算桁数が多すぎると計算時間が長くなるため，必要最小限の計算桁数で実行することが望ましい。そこで，今回我々は，古典的誤差評価法によって得られる誤差の評価値に基づいて，下記のように文字通りの Trial & Error を行うことで計算桁数を必要なだけ自動的に確保するようにした。この際の計算桁数の増減方法は，

1. 収束するが要求精度より近似値の精度が低い場合→計算桁数の増量分を単調増加
2. 収束しない場合→計算桁数の増量分を 2 倍に増加

とする。計算桁数さえ十分確保できれば収束する見通しがあれば，(2) の場合は計算桁数を早く増加させることで，失敗試行を減らすことができる。Ziv の戦略 [9] と似ているが，これより計算桁数の増量は抑えた方が，多くの問題において少ない桁数で要求精度を満足することが実験的に確認できているため，このように設定した。

以下，我々のアルゴリズムの詳細を述べる。

ユーザが指定した精度桁数 (10 進) を $U \in \mathbb{N}$ とする。あらかじめ指定しておいた精度桁数の増分の最小値を $D \in \mathbb{N}$ を用い，桁数の増分量 C を

$$C = \max(D, U/10) \tag{11}$$

とする。つまり 10%増量する。これを元に，実際に計算する桁数 S を

$$S = U + C \tag{12}$$

とし，丸め誤差を評価するために必要な精度桁数 L を

$$L = S + C \tag{13}$$

として，S より C 桁だけ余分に桁数を取るようにする。このようにして決めた精度桁数に基づいて $R(x^S)$ を求める。

更に，収束判定に必要な相対許容度 ε_r と絶対許容度 ε_a を

$$\varepsilon_r = 10^{-U}, \quad \varepsilon_a = 10^{-2U} \tag{14}$$

とし，基本的には

$$|x_k^S - x_{k-1}^S| \le \varepsilon_r |x_k^S| + \varepsilon_a$$

を満足した時点で収束したものとする。この時の反復回数を k_s とすれば，この時点における理論誤差の評価値 $T(x_{k_s}^S)$ を得るために，更にもう一度だけ反復を行い，$x_{k_s}^S$ の理論誤差の評価値 $T(x_{k_s}^S)$ を確定する。同時に丸め誤差の評価値 $R(x_{k_s}^S)$ も確定するので，(10) 式に基づいて絶対誤差の評価値 $E(x_{k_s}^S)$ が決定される。

もしこの時点で $E(x_{k_s}^S) < \varepsilon_r |x_{k_s}^S|$ を満足していれば，これをユーザ指定の精度を持つ近似値として受け入れ，U 桁に丸めてユーザに返す。そうでなければさらに C 桁追加してもう一度計算を行う。

これとは別に，もし指定された最大反復回数に達しても収束しない場合は，C を 2 倍して再計算を行うようにするが，前述したように今回この処理は行う必要はない。

今回実装した代数方程式ソルバは，以上の計算桁数増加アルゴリズムに基づき，ユーザ指定精度を満足するために必要な桁数で自動的に計算が行えるようになっており，極端に過大すぎる桁数での計算を行わない。この実現のためには，次の 3 つの機能を持った多倍長浮動小数点ライブラリが不可欠である。

- 変数ごとに精度を可変に設定できる
- 異なる精度を持つ変数どうしの演算 (混合精度演算) が可能
- プログラム実行中に動的に精度を変更する機能

我々の BNCpack[4] が土台としている GMP[1] や MPFR[11, 8] はこれらの機能を持っており，提案した自動計算アルゴリズムの実行が可能となるものである。

5. IEEE754 倍精度計算による前処理

我々の目指す，ユーザの要求精度を満足する近似値を自動的に得るためのアルゴリズムには，以上のような 3 条件を満たす混合精度演算が可能な多倍長計算環境が欠かせない。しかし，現状ではソフトウェアによって実装された多倍長計算は，CPU 内でハードウェア処理が行われる IEEE754 倍精度計算に比べ 10^3〜10^6 倍遅くなることが確認されている*。また，多くの良条件な問題では倍精度計算でも十分な精度が得られることを考えると，処理の重い多倍長計算の前処理として，まずは高速な倍精度計算を行うことは無駄ではないと思われる。

この前処理がうまく働くのであれば次の 2 点が期待できる。

* cf. http://na-inet.jp/research/akita2006.pdf

1. 理論誤差，丸め誤差評価によって，損失桁 P の見積もりが行えるので，多倍長計算で必要な桁数に上乗せが可能となる。
2. ユーザの要求精度 ($U < 15$ の場合に限定) を満足することが明らかとなれば，この時点で計算を終了させることができる。

なお，丸め誤差評価は IEEE754 倍精度計算のみで実行するため，4 つの丸めモード (RN, RZ, RP, RM) で計算した値を用いて，デフォルトの RN モード値との差の最大値を取って簡易評価を行う [6]。こうすることで，前処理の高速性を最大限維持することができる。

しかし，悪条件な問題に対しては初期誤差による影響によって P の見積もりや要求精度の判定が騙されることもあり得る。実際，後述する数値実験の結果では，特に近接解を持つ場合にそのような現象が現れる問題も見つかっている。従って，現状ではこの前処理を行うか否かはユーザの判断に任せるようにしている。もしこの前処理を行う時には，計算桁数の増分量 (11) を

$$C = \max(D, U/10) + P$$

として考慮するようにした。しかし，十分余裕のある精度桁数を取って計算する我々のアルゴリズムでは，P をどのように繰り入れようとあまり影響はなさそうである。

6. 数値実験

以上述べてきた代数方程式の解法，及び計算精度桁数を増加させるアルゴリズムに基づき数値実験を行った結果をここで示す。数値実験に使用したハードウェア及びソフトウェアは下記の通りである。

H/W AMD Athlon64 X2 3800+ (2GHz), 4GB RAM

S/W Fedora Core 4 x86_64, gcc 4.1.1, CRlibm[2] 1.0beta1, MPFR 2.2.0, GMP 4.2.1

Cardano 法，Ferrari 法内部では cos 関数を使用しているが，x86_64 の環境では gcc 標準の libm ライブラリに重大なバグ*が出現するため，今回のように IEEE754 倍精度計算において丸めモードを変更した後の値の正確さが全く保証されない。それを回避するために CRlibm に実装されている cos_rn,rz,ru,rd 関数を用いている。

6.1 例題

使用した例題は数値計算ハンドブック [10] に掲載されている下記の 14 個の代数方程式である。本稿では式番号で方程式を区別することにする。

2 次方程式

例 1 $x^2 + 53.99999999x - 0.00000054$
$= (x - (-0.00000001))(x - 54) = 0$ (15)

例 2 $x^2 + x + 1$
$= \left(x - \left(-\frac{1}{2} + \frac{\sqrt{3}}{2}\mathrm{i}\right)\right)\left(x - \left(-\frac{1}{2} - \frac{\sqrt{3}}{2}\mathrm{i}\right)\right) = 0$ (16)

例 3 $x^2 - 2.40001x + 1.440012$
$= (x - 1.2)(x - 1.20001) = 0$ (17)

*cf. http://sourceware.org/bugzilla/show_bug.cgi?id=3976

3 次方程式

例 1 $x^3 - 6x^2 + 11x - 6 = (x-1)(x-2)(x-3) = 0$ (18)

例 2 $x^3 - 3x^2 - 9x - 5 = (x-1)^2(x-5) = 0$ (19)

例 3 $x^3 + 2x^2 + 3x + 2$
$= (x + (-1))\left(x - \left(-\frac{1}{2} + \frac{\sqrt{7}}{2}\mathrm{i}\right)\right)\left(x - \left(-\frac{1}{2} - \frac{\sqrt{7}}{2}\mathrm{i}\right)\right) = 0$ (20)

例 4 $x^3 + 3x^2 + 3x + 1$
$= (x - (-1))^3 = 0$ (21)

例 5 $x^3 + 3x^2 - 2$
$= (x - (-1))\left(x - \left(-1 + \sqrt{3}\right)\right)\left(x - \left(-1 - \sqrt{3}\right)\right) = 0$ (22)

4 次方程式

例 1 $x^4 + 2x^3 + 6x^2 + 26x + 45$
$= (x - (-2 - \mathrm{i}))(x - (-2 + \mathrm{i}))$
$\times \left(x - \left(-1 + 2\sqrt{2}\,\mathrm{i}\right)\right)\left(x - \left(-1 - 2\sqrt{2}\,\mathrm{i}\right)\right) = 0$ (23)

例 2 $x^4 + 10004.01x^3 + 40107.04x^2 + 70400.07x + 700$
$= (x - (-10000))(x - (-0.01))$
$\times \left(x - \left(-2 - \sqrt{3}\,\mathrm{i}\right)\right)\left(x - \left(-2 + \sqrt{3}\,\mathrm{i}\right)\right) = 0$ (24)

例 3 $x^4 + 2x^3 + 2x^2 + 16x + 24$
$= (x - (-2))(x - (-2))$
$\times \left(x - \left(1 - \sqrt{5}\,\mathrm{i}\right)\right)\left(x - \left(1 + \sqrt{5}\,\mathrm{i}\right)\right) = 0$ (25)

例 4 $x^4 - 8x^3 + 24x^2 - 32x + 16$
$= (x-2)^4 = 0$ (26)

例 5 $x^4 + x^2 + 1$
$= \left(x - \left(-\frac{1}{2} - \frac{\sqrt{3}}{2}\mathrm{i}\right)\right)\left(x - \left(-\frac{1}{2} + \frac{\sqrt{3}}{2}\mathrm{i}\right)\right)$
$\times \left(x - \left(\frac{1}{2} + \frac{\sqrt{3}}{2}\mathrm{i}\right)\right)\left(x - \left(\frac{1}{2} - \frac{\sqrt{3}}{2}\mathrm{i}\right)\right) = 0$ (27)

例 6 $x^4 - 4x^3 + 5.9999999995x^2 - 3.999999999x$
$+ 0.99999999950000000004$
$= (x - 0.99998)(x - 0.99999)$
$\times (x - 1.00002)(x - 1.00001) = 0$ (28)

6.2 要求精度と真の精度との比較

まず，ユーザ要求精度 (10 進桁数)U を 20, 50, 100, 1000 桁とした場合の計算結果を Table 1 に示す。真の解を指定精度の 10 倍の桁数で計算し，これを真の値として実数部，虚数部それぞれの相対誤差を導出したものである。数値が 0 になっているのは，全ての桁が一致しているということを示している。

この結果から，全ての指定精度及び全ての例題に対して要求精度を満足する解が得られていることが分かる。要求精度より 1, 2 桁多めの桁数が出ているのは MPFR の多倍長浮動小数点数が 2 進であるため，10 進変換の際の誤差を考慮して天井関数 (ceil) によって 2 進 bit 数が多めに与えられていることによるものと思われる。今回は全て 10^{-U} 以下の相対誤差に収まっているが，数値によっては丸めた結果，若干これを上回る相対誤差になることがある [7]。

Table 1: 数値計算ハンドブックの例題: 要求精度 U と相対誤差

2nd degree					
Eq.No.	Exact root	Relative Errors: Re, Im			
		$U = 20$	$U = 50$	$U = 100$	$U = 1000$
(15)	-54	0, 0	0, 0	0, 0	0, 0
	0.00000001	7.1e − 22, 0	2.5e − 51, 0	1.4e − 101, 0	1.3e − 1001, 0
(16)	$-\frac{1}{2}+\frac{3}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	$-\frac{1}{2}-\frac{3}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
(17)	1.20001	1.5e − 21, 0	2.7e − 51, 0	3.4e − 101, 0	4.3e − 1001, 0
	1.2	2.3e − 21, 0	1.8e − 51, 0	1.9e − 101, 0	6.3e − 1001, 0
3rd degree					
Eq.No.	Exact root	Relative Errors: Re, Im			
		$U = 20$	$U = 50$	$U = 100$	$U = 1000$
(18)	3	0, 0	0, 0	0, 0	0, 0
	2	0, 0	0, 0	0, 0	0, 0
	1	0, 0	0, 0	0, 0	0, 0
(19)	5	0, 0	0, 0	0, 0	0, 0
	-1	0, 0	0, 0	0, 0	0, 0
	-1	0, 0	0, 0	0, 0	0, 0
(20)	-1	0, 0	0, 0	0, 0	0, 0
	$-\frac{1}{2}-\frac{\sqrt{7}}{2}$i	0, 3.9e − 21	0, 2.6e − 52	0, 1.0e − 102	0, 3.8e − 1001
	$-\frac{1}{2}+\frac{\sqrt{7}}{2}$i	0, 3.9e − 21	0, 2.6e − 52	0, 1.0e − 102	0, 3.8e − 1001
(21)	-1	0, 0	0, 0	0, 0	0, 0
	-1	0, 0	0, 0	0, 0	0, 0
	-1	0, 0	0, 0	0, 0	0, 0
(22)	$-1-\sqrt{3}$	2.9e − 21, 0	3.8e − 51, 0	3.4e − 101, 0	6.8e − 1001, 0
	$-1+\sqrt{3}$	1.6e − 21, 0	5.4e − 52, 0	3.0e − 101, 0	5.9e − 1002, 0
	-1	0, 0	0, 0	0, 0	0, 0
4th degree					
Eq.No.	Exact root	Relative Errors: Re, Im			
		$U = 20$	$U = 50$	$U = 100$	$U = 1000$
(23)	$1+2\sqrt{2}$ i	0, 1.9e − 21	0, 2.9e − 51	0, 2.0e − 101	0, 3.4e − 1001
	$1-2\sqrt{2}$ i	0, 1.9e − 21	0, 2.9e − 51	0, 2.0e − 101	0, 3.4e − 1001
	$-2+$ i	0, 0	0, 0	0, 0	0, 0
	$-2-$ i	0, 0	0, 0	0, 0	0, 0
(24)	$-2+\sqrt{3}$ i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	$-2-\sqrt{3}$ i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	-10000	0, 0	0, 0	0, 0	0, 0
	0.01	8.5e − 22, 0	6.7e − 52, 0	1.1e − 101, 0	6.5e − 1001, 0
(25)	$1+\sqrt{5}$ i	0, 2.3e − 21	0, 8.7e − 52	0, 4.5e − 101	0, 4.3e − 1001
	$1-\sqrt{5}$ i	0, 2.3e − 21	0, 8.7e − 52	0, 4.5e − 101	0, 4.3e − 1001
	-2	0, 0	0, 0	0, 0	0, 0
	-2	0, 0	0, 0	0, 0	0, 0
(26)	2	0, 0	0, 0	0, 0	0, 0
	2	0, 0	0, 0	0, 0	0, 0
	2	0, 0	0, 0	0, 0	0, 0
	2	0, 0	0, 0	0, 0	0, 0
(27)	$\frac{1}{2}+\frac{\sqrt{3}}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	$-\frac{1}{2}+\frac{\sqrt{3}}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	$\frac{1}{2}-\frac{\sqrt{3}}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
	$-\frac{1}{2}-\frac{\sqrt{3}}{2}$i	0, 3.2e − 21	0, 2.3e − 52	0, 1.3e − 101	0, 2.5e − 1002
(28)	1.00001	8.8e − 22, 0	5.3e − 51, 0	1.7e − 101, 0	3.2e − 1001, 0
	0.99999	8.8e − 22, 0	5.0e − 53, 0	1.7e − 101, 0	6.3e − 1001, 0
	0.99998	1.8e − 21, 0	9.9e − 53, 0	2.2e − 101, 0	3.1e − 1001, 0
	1.00002	1.8e − 21, 0	9.9e − 53, 0	3.5e − 101, 0	6.4e − 1001, 0

6.3 前処理における損失桁評価値

次に，IEEE754 倍精度計算による損失桁評価値 P の値を Table 2 に示す。

Table 2: IEEE754 倍精度計算における損失桁評価値 P

Ex.no	P	Ex.no	P
(15)	4	(22)	2
(16)	0	(23)	2
(17)	6	(24)	11
(18)	1	(25)	9
(19)	1	(26)	1
(20)	1	(27)	0
(21)	0	(28)	1

前述したように，この評価を行うと多くの場合に丸め誤差の上限を捕らえることができるが，反面，評価としては少し大きめに出ることが多くなる。例えば (15) では 4 桁の損失があると報告されているが，RN モードでの近似値は

$$-5.40000000000000000e+01$$
$$1.00000000000000002e-08$$

であり，また，(25) でも

$$1.00000000000000000 + 2.23606797749978981\,i$$
$$1.00000000000000000 - 2.23606797749978981\,i$$
$$-2.00000000000000000 + 0.00000000000000000\,i$$
$$-2.00000000000000000 + 0.00000000000000000\,i$$

ほぼ 16 桁一致している。後者のケースは，後述するように実装した Ferrari 法の不安定さに起因するものかもしれない。

逆にほぼ正確に捉えているのは (17) で，実際，

$$1.20001000000020674$$
$$1.19999999999979323$$

という近似値になる。また，(24) でも

$$-2.00000008949473340 + 1.73205091026944813\,i$$
$$-2.00000008949473340 - 1.73205091026944813\,i$$
$$-1.00000000000000000e+04$$
$$-9.99982101075147511e-03$$

となり，11 桁の損失桁という報告はほぼ適正と言える (小数部を 18 桁出力していることに注意)。

過小評価になっているのは (28) だが，これは定数項が倍精度では収まり切れず，結果として別の方程式に化けてしまっていることが原因である。実際，定数項が 0.99999999950000000003 の場合，解は

$$0.99997925\cdots,\ 0.99999165\cdots$$
$$1.00000834\cdots,\ 1.00002074\cdots$$

となり，定数項が 0.99999999950000000005 の場合，解は

$$0.99998097\cdots,\ 0.99998824\cdots$$
$$1.00001175\cdots,\ 1.00001902\cdots$$

と激しく変化する。よって，問題自体が変化している以上，変化した問題に対してこの評価結果が正しいかどうかを考える必要が出てくる。最終的には初期誤差の範囲を特定した上で，区間として係数を与え，多くの精度保証数値計算において行われているように，それによって解がどのような変動を示すかを深く考える必要があろう。なお前述したように，このような不正確な評価値が与えられても我々の任意精度計算アルゴリズムの最終結果には影響していない。

参考文献 [10] ではこの辺りの事情を深く吟味せずによしとしているが，このような初期誤差に対する不注意さが，結果としてそこで著者が実装した Ferrari 法の数値的不安定性にも気が付かなかった要因と思われる。他山の石としたい事例である。

6.4 計算桁数の変化

最後に，提案した計算桁数の変動アルゴリズムに則って自動的に変化していく計算桁数について考察する。Table 3 に 2 次方程式の例題を，Table 4 に 3 次方程式の例題を，Table 5 に 4 次方程式の例題の結果をそれぞれ示す。

Table 3: 要求精度 U と計算桁数の変化 (1/3)

2nd degree		
Eq.No.	U	Changes of $S(L)$
(15)	20	34(48)
	50	64(78)
	100	114(128)
	1000	1104(1208)
(16)	20	30(40)
	50	60(70)
	100	110(120)
	1000	1100(1200)
(17)	20	36(52)
	50	66(82)
	100	116(132)
	1000	1106(1212)

2，3 次方程式の場合はリトライを一度も実行することなく要求精度の解が得られていることが分かる。それに対して，4 次方程式の場合は条件の悪い問題 (25)，あるいは初期誤差による影響の大きな問題 (28) において，最大 4 回のリトライを行っていることが分かる。

特に (28) の場合では，計算桁数が比較的小さい時には Ferrari 法内部の実数の平方根計算においてゼロに近いマイナス値になることがあり，それをリカバーするためにリトライを繰り返している。IEEE754 倍精度計算結果が参考文献 [10] のものとほぼ同じ精度であること，最終的には要求精度の近似値が得られていることから，解法に潜む数値的不安定性による現象と思われるが，現時点では深く解析できていない。

しかし，不安定なアルゴリズムの実装であることがこのリトライ回数によって明確に現れることは，我々の任意精度計算アルゴリズムの存在価値になり得る。今後，更に安定性を増した実装を行った際の比較実験では有効に働くと思われる。

7. まとめと今後の課題

以上述べてきたように，我々の 4 次以下の任意精度代数方程式ソルバは，全ての例題に対してユーザの要求精度を満足する近似値が得られることが判明した。しかし，Ferrari 法の

Table 4: 要求精度 U と計算桁数の変化 (2/3)

3rd degree		
Eq.No.	U	Changes of $S(L)$
(18)	20	31(42)
	50	61(72)
	100	111(122)
	1000	1101(1202)
(19)	20	31(42)
	50	61(72)
	100	111(122)
	1000	1101(1202)
(20)	20	31(42)
	50	61(72)
	100	111(122)
	1000	1101(1202)
(21)	20	30(40)
	50	60(70)
	100	110(120)
	1000	1100(1200)
(22)	20	32(44)
	50	62(74)
	100	112(124)
	1000	1102(1204)

Table 5: 要求精度 U と計算桁数の変化 (3/3)

4th degree		
Eq.No.	U	Changes of $S(L)$
(23)	20	32(44)
	50	62(74)
	100	112(124)
	1000	1102(1204)
(24)	20	41(62)
	50	71(92)
	100	121(142)
	1000	1111(1222)
(25)	20	39(58) → 42(64)
	50	69(88) → 78(106) → 106(162)
	100	119(138)
	1000	1109(1218) → 1218(1436) → 1436(1872)
(26)	20	31(42)
	50	61(72)
	100	111(122)
	1000	1101(1202)
(27)	20	30(40)
	50	60(70)
	100	110(120)
	1000	1100(1200)
(28)	20	31(42) → 42(64) → 64(108)
	50	61(72) → 72(94)
	100	111(122) → 122(144)
	1000	1101(1202) → 1202(1404) → 1404(1808)

アルゴリズムに関してはまだ改良の余地が残っている。

今後の課題としては，次のことが挙げられる。

7.1 4 次方程式向けの計算アルゴリズムの改良

現在実用に供されている数値計算ライブラリでも，GSL[12] のように 4 次方程式向けのソルバが用意されていないものがある。これは前述したように安定したアルゴリズムの実装が難しいこと，反復法で十分高精度の解が得られることが要因として上げられよう。今後，現状の Ferrari 法より安定した計算アルゴリズムを実装していきたい。その際には，別の方法 (Brown 法など) との比較検討も行っていきたい。

7.1.1 高次代数方程式に対する任意精度ソルバの開発

我々は既に 1024 次の実係数代数方程式に対して実数解の任意精度計算が可能であることを示した [7]。現在では，特に高次の代数方程式は，行列の固有値問題用のアルゴリズムを用いることで，近似値の損失桁を少なくすることができることが知られている。実際，近接度が増した高次の代数方程式に対しても，QR 法を用いることで損失桁は殆ど発生しないことが我々の数値実験で確認できている。しかし，適切な初期値が設定でき，多倍長計算を用いることができる場合は，あくまで代数方程式のまま解くことで，計算時間と必要記憶領域が節約できることも同時に判明している。従って，前処理には IEEE754 倍精度計算を用いた固有値計算アルゴリズムを用いて近似値を得，これを多倍長計算による代数方程式向けの反復法の初期値として用いる方法が，今のところ多くの問題では有効に働くと考えている。

次の課題は複素数解を含む高次代数方程式の全ての解の任意精度計算を可能にするソルバを実装することだが，この場合でも，IEEE754 倍精度計算による前処理と，多倍長計算による計算アルゴリズムとの組み合わせを行うことが良い効果を生むかどうかを確認する必要がある。その知見を考慮した上で，より有効な任意精度代数方程式ソルバを実装してきたい。

7.1.2 より多くの問題に対する検証

我々の提案する古典的誤差評価法は，あくまで経験的な知見を元にしているため，更に多くの例題に対して我々のソルバを適用してその有効性を確認することが欠かせない。そこで，本稿で実装した 4 次以下の実係数代数方程式ソルバを広く世界中のユーザに使ってもらうため，Fig.1 のような Web インターフェースを実装し，広く公開を行っている*。

セキュリティ維持のため，ユーザの要求精度は 1000 桁に抑えてあるが，問題によってはそれよりも大きな計算桁数を必要とするケースもあり，本当にセキュアなものになっているかどうかは定かではない。常に監視を行いつつ，多くの問題に対してこの Web インターフェースを適用してもらい，より多くの例題の収集と手法の有効性の確認を行っていきたい。

謝辞

Web インターフェースを公開後，Vincent Lérfeve, Paul Zemmermann の両氏から適切なアドバイスを頂いた。また，x86_64 Linux 環境における glibc の初等関数のバグについては，NA Digest を通じて様々な方から資料や知見を寄せて頂いた。今回行った計算が可能になったのはこれらの方々の助力なくしては考えられない。ここで御礼申し上げたい。

最後に，古典的誤差評価法に基づく任意精度計算の実装は，長年に渡る元・日本大学　永坂秀子先生とのディスカッションによって培われたものである。博士前期課程以来十数年以

*http://ex-cs.sist.ac.jp/~tkouya/try_cee_algebraic_eq.html

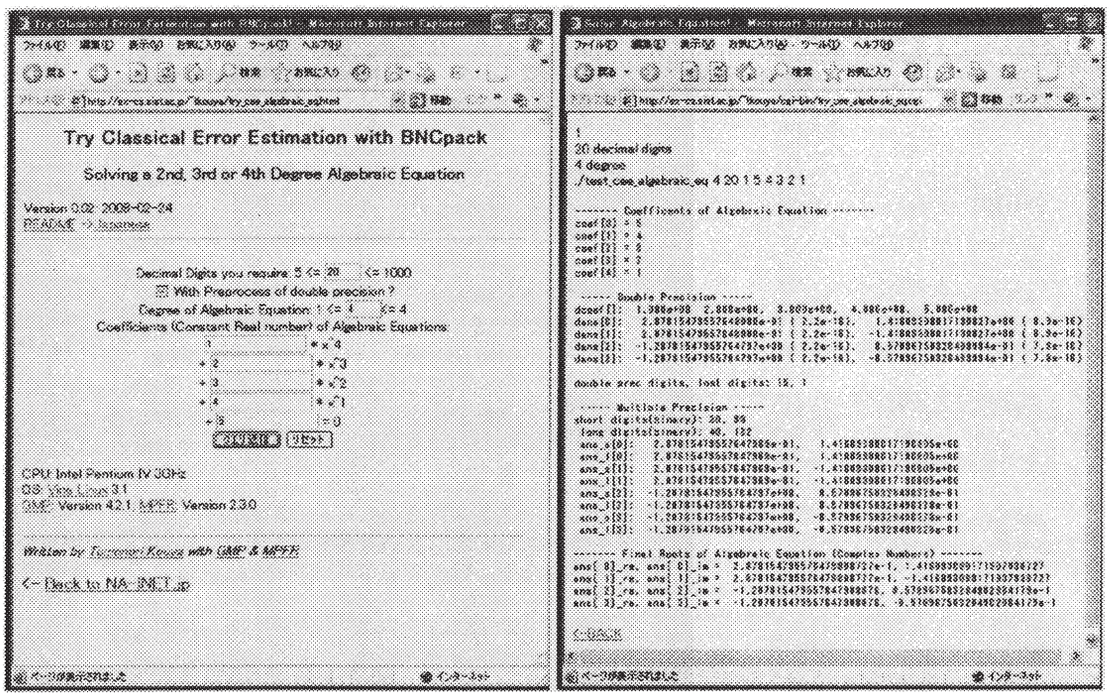

Fig. 1: デモ用 Web インターフェース (左) と計算結果表示例 (右)

上もの間，ご指導頂いていることに対して，厚く御礼申し上げる。

参考文献

1) Swox AB. GNU MP. http://swox.com/gmp/.
2) CRlibm. Correctly rounded mathematical library. http://lipforge.ens-lyon.fr/www/crlibm/.
3) 伊理正夫, 藤野和建. 数値計算の常識. 共立出版, 1985.
4) Tomonori Kouya. BNCpack. http://na-inet.jp/na/bnc/.
5) 幸谷智紀. ソフトウェアとしての数値計算. http://na-inet.jp/nasoft/.
6) 幸谷智紀, 永坂秀子. IEEE754 規格を利用した丸め誤差の測定法について. 日本応用数理学会論文誌, Vol. 7, No. 1, pp. 79 – 89, 1997.
7) 幸谷智紀. 実用的な古典的誤差評価法の提案と gauss 型積分公式の分点計算への応用について. 情報処理学会論文誌, Vol. 48, No. SIG18(ACS20), pp. 1 – 11, 2007.
8) L.Fousse, G.Hanrot, V.Lefèvre, P.Pélissier, and P.Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, Vol. 33, No. 2, p. 13, 2007.
9) J.-M. Muller. *Elementary Functions. Algorithm and Implementation*. Birkhäuser, first edition, 1997.
10) 大野豊, 磯田和男監修. 新版 数値計算ハンドブック. オーム社, 1990.
11) MPFR Project. The MPFR library. http://www.mpfr.org/.
12) GSL Team. Gnu scientific library. http://www.gnu.org/software/gsl/.