GAL における近似代数演算用の諸機能[1]

佐々木建昭（筑波大学 数学系）　加古富志雄（奈良女子大学 理学部）

0. はじめに

現在、多項式の近似演算が世界的に流行の気配を見せているが、この方面では日本が最先端である。我々は８年も前に近似代数の概念を提唱し [1]、近似 GCD、近似因数分解などの算法を考案するかたわら、システム面で近似代数演算に必要な機能を追求してきた。この過程で、新しい数値型として『有効浮動小数』なるものを提案し、NSL（Nara Standard Lisp, 加古が開発した Lisp システム）に実装した。また、関連する数式処理用の諸機能を GAL（General Algebraic Language/Laboratory, 佐々木が開発した数式処理システム）に付加してきた。本稿では、これまでに我々が NSL-GAL システムに付加した近似代数用の諸機能について、簡単に述べる。

1. NSL における数の体系

現在、NSL には数として、整数、有理数、浮動小数、浮動複素数、有効浮動小数、有効浮動複素数、および区間数（有理数以外は固定精度と任意精度）が装備されている。（有効数については次節で説明する）。これらは次のように体系化されている。

- 数（`numberp`）…
  - 整数（`integerp`）
  - 有理数（`rationalp`）
  - 浮動小数（`floatp`）……
    - 浮動小数
    - 有効浮動小数
  - 複素数（`complexp`）…
    - 浮動複素数
    - 有効浮動複素数
  - 区間数（`intervalp`）

上記で `numberp` などは個々の数値型を識別する NSL の質問関数である。複数の数値型にまたがる有効数と任意精度数はそれぞれ `effectivep`, `bignump` で識別される。

2. 有効浮動小数と有効浮動複素数

浮動小数などの近似数を扱う場合、丸め誤差（数を２進 K 桁で表すとき、2^{-K} 未満の端数を切り上げるか切り捨てるかで生じる誤差）と桁落ちによる誤差（上位 k 桁が等しい数を引くと有効桁数が k だけ減少する）に注意しなければならない。

上記の誤差へ対処する方法として、1960 年代なかばに区間数なる数値表現が考案された [2]。誤差対処法というと区間数が持ち出されるところをみると、それ以上のアイデアはまだ提出されていないのであろう。区間数とは、実数 a の誤差が ε 以下であるとき、a を $[a-\varepsilon, a+\varepsilon]$ なる区間で表すものである。もちろん、演算結果が常に区間内に厳密に収まるように区間幅を決定する。したがって、数学的には申し分のない表現のはずである。しかし、実際に区間数を使用してみると、特殊な場合（たとえば、根の近傍の初期値から出発してニュートン法で根を計算する場合など）を除き、区間幅が計算毎に急速に拡大して、答が意味をなさなくなる場合がほとんどである。すなわち、区間数は実際には「絵に描いた餅」と言ってよかろう。

ところで、上述の２種類の誤差は全然異なる性質のものである。丸め誤差の集積を理論的に正確に予測するのは非常に難しいが、実際にその大きさを知るのは時間さえかければ容易である：異

[1] 本研究は部分的に文部省科研費（課題番号 06558037：近似代数計算システムの開発）、および日本学術振興会・日中科学協力事業 (課題：近似的代数算法の研究と応用) の援助を受けて行われた

なる精度で同じ計算をして結果を引き算してみればよい。しかも、非常に多数回の計算をするのでないかぎり、丸め誤差はそんなに大きくない。一方、桁落ちの方は、たった一回の計算で精度が5桁くらい落ちることはザラである。しかも、代数的計算の場合、桁落ちは頻繁に起きるのである：たとえば、多項式剰余列の計算がそうである。

そこで筆者らは、桁落ちをほぼ正確に検出できる数値表現として、「有効浮動小数」なるものを考案した [3]。アイデアは実に簡単である。実数を

$$[\text{Value-part},\ \text{Error-part}]$$

と二つの数の組で表す。Value-part は通常の浮動小数そのものであり（演算も通常の浮動小数として実行する）、Error-part は初期値を $\varepsilon_M \times |\text{Value-part}|$ と定める。ここで、ε_M はマシンイプシロンである（浮動小数の仮数部を K ビットで表すとき、$\varepsilon_M = 2^{-K}$ となる）。そして、二つの有効浮動小数 $[v_1, e_1]$ と $[v_2, e_2]$ の和と差、積、商はそれぞれ

$$\begin{aligned}
[v_1, e_1] \pm [v_2, e_2] &= [v_1 \pm v_2,\ \max\{e_1, e_2\}], \\
[v_1, e_1] \times [v_2, e_2] &= [v_1 \times v_2,\ \max\{|v_1 e_2|, |v_2 e_1|\}], \\
[v_1, e_1] \div [v_2, e_2] &= [v_1 \div v_2,\ \max\{|e_1/v_2|, |v_1 e_2/v_2^2|\}],
\end{aligned}$$

と計算する。すなわち、Value-part の数字のうち Error-part の大きさまでの数値が Value-part の有効数字であるように Error-part を定めるのである。同様に、複素数は

$$[\text{RealValue-part},\ \text{ImagValue-part},\ \text{Error-part}]$$

と三つの数の組で表現する（詳しくは [3] を参照）。

有効浮動小数は、上述のように丸め誤差へは別途対処するものとし、代数的計算において真に重要な桁落ちに実際的に対処する数値表現である。上記だけでは区間数と有効数の優劣は分からないが、実際に計算してみるとその差は歴然としてくる。

3. 近似係数多項式の扱い

NSL には上記のように多数の型の数が装備されているが、これらの数値はいずれも GAL の多項式・有理式あるいは一般数式の係数として使える。そして、これらの係数はコマンド一つで任意の指定された型に変換できる。たとえば、P を多項式として、

`coef2int`(P);　　（“2” は “from 〜 to 〜” の “to” に対応する）

とすると、答として、整数あるいはガウス整数（実数部と虚数部がともに整数の複素数）を係数とする多項式が返される。コマンド `coef2int` における変換規則は以下のとおりである。

$$\begin{aligned}
\text{整数} &\Longrightarrow \text{そのまま、} \\
\text{有理数} &\Longrightarrow \text{商の整数部、} \\
\text{浮動小数} &\Longrightarrow \text{小数部を切捨て、} \\
\text{浮動複素数} &\Longrightarrow \text{実数部と虚数部を整数化、} \\
\text{有効浮動小数} &\Longrightarrow \text{浮動小数にして整数化、} \\
\text{有効浮動複素数} &\Longrightarrow \text{浮動複素数にして整数化、} \\
\text{区間数} &\Longrightarrow \text{区間の中間値にして整数化。}
\end{aligned}$$

なお、有効数においては、|Value-part| が Error-part 以下のときは 0 に変換され、区間数が 0 を含む区間のときも 0 に変換される。

有効数を係数とする多項式に対しては、係数部全体の有効精度を検出する次の三つのコマンドが用意されている。

`precmin`(P)　：多項式 P の全ての係数の有効精度の最小値を返す、
`precmax`(P)　：多項式 P の全ての係数の有効精度の最大値を返す、
`precminmax`(P)　：[有効精度の最小値, 有効精度の最大値] を返す。

ここで、有効精度とは次のように Error-part に相対的に Value-part の大きさで定義される。

$$\text{有効浮動小数} \quad : \quad |\text{Value-part}| \ / \ \text{Error-part}$$
$$\text{有効浮動複素数} \quad : \quad \max\{|\text{RealValue-part}|, |\text{ImagValue-part}|\} \ / \ \text{Error-part}.$$

上記の関数を用いて、計算の節目節目で結果式の有効精度をチェックし、有効精度が大幅に減少したら数値精度を上げて再計算をするのである。

NSL の数には固定精度と任意精度の二つがあることを述べたが、任意精度数の精度指定は、Lisp では関数 `precision` で行い、GAL では declare 文で次のように行う。

`declare precision: 100 ;`　（`decl prec: 100 ;` と略記可）

近似係数の多項式の四則演算については、加減算と乗算は正確な多項式に対するプロシジャを用いて行うが、除算は特別なプロシジャを用いて実行している。通常の除算法では、係数が近似数の場合、主係数が小さい多項式で割ると桁落ちが生じて、商と剰余の有効精度が低くなるからである。有効精度を可能な限り下げないで除算を実行するために、1 変数多項式に対しては [4] に記されている算法がインプリメントされた。多変数多項式に対しては、Hensel 構成的に商と剰余を構成する Yun の算法 [5] がインプリメントされた。

4. DKA 法と因子分離法

DKA 法とは Durand-Kerner-Aberth 法の略称で、1 変数代数方程式の全根を同時に計算する非常に実用的な数値計算法である。因子分離法とは、1 変数多項式 $F(x)$ と精度 ε_0 でのその近似因子 $G_0(x), H_0(x)$ が与えられたとき、すなわち

$$F(x) = G_0(x)H_0(x) + \delta F_0(x), \quad \|\delta F_0\|/\|F\| = \varepsilon_0 \ll 1,$$

のとき、より高精度な近似因子 $G_1(x), H_1(x)$ を計算することである：

$$F(x) = G_1(x)H_1(x) + \delta F_1(x), \quad \|\delta F_1\|/\|F\| = \varepsilon_1 \ll \varepsilon_0.$$

上記二つの演算は、種々の近似的代数演算の一部分として使われる基礎的な演算である（後者については研究が始まったばかりで、応用法は今後の課題である）。

DKA 法はどんな数値計算ライブラリにも装備されているが、任意精度の演算は通常のライブラリには装備されていない。代数的算法では任意精度で根を求めることが必要であるので、NSL-GAL システムでは独自に DKA 法を実装した（Lisp でプログラムされている）。さらに、与式 $F(x)$ が実係数の場合には、スツルムの定理で実根の個数を確定してから実根と複素共役根を計算する、いわゆる実多項式用 DKA 法（[6] を参照）が実装されている。演算に対する要求精度を ε とするとき、DKA 法と実多項式用 DKA 法はそれぞれ次のように起動される。

$$\texttt{dka}(F(x), [\text{初期値の組}], \varepsilon),$$
$$\texttt{dkareal}(F(x), [\text{実根初期値の組}], [\text{複素共役根初期値の組}], \varepsilon)$$

ただし、[初期値の組] や [実根初期値の組] などは [nil] でもよい。単根の場合、根は精度 ε まで計算されるが、m 重根は精度 $\sqrt[m]{\varepsilon}$ までしか計算できない。各根に対する Smith の誤差上界値が GAL 変数 `byproduct` の値として取り出せる。

因子分離法は、GAL コマンド `sepfactor` により次のように起動される。

$$\texttt{sepfactor}(F(x), G_0(x), H_0(x), \varepsilon)$$

ここで、$G_0(x), H_0(x)$ は与多項式 $F(x)$ の粗い精度での近似因子で、ε は分離の精度である。因子分離法では、G_0 と H_0 が共通近接根を持たぬことが必要であり、この条件が満たされない場合、

答が求まらなかったり、あるいは収束が遅くなる。因子分離法の詳細と一つの応用例については [7] を参照されたい。因子分離法は非常に簡単だが、それゆえ非常に強力で、今後、多くの応用がなされると思う。

5. 近似 GCD と近似因数分解

多項式の GCD と因数分解はもっともポピュラーかつ基本的な代数演算である。これらの演算に対する GAL のコマンドは以下である。

$$\texttt{gcd}(P_1, P_2, \varepsilon); \quad \texttt{factorize}(P, \varepsilon); \qquad (\varepsilon\text{は省略可})$$

ここで、ε は近似演算に対する精度であり、この引数が入力されると、与式 P_1, P_2 あるいは P が正確な多項式であっても、近似演算が起動される。引数 ε が省略された場合は、近似演算が起動されるか否かは与多項式の係数が正確か否かによる。

近似 GCD について：1 変数多項式に対しては Sasaki-Noda により提案された多項式剰余列に基づく算法 [8] がインプリメントされた。多変数の場合、[9] に記されている算法は中間式膨張を起こして効率が悪いので、概略以下のような Hensel 構成に基づく算法がインプリメントされた。

多変数近似 GCD 算法の概略

入力： $P_1, P_2 \in \mathbf{C}[x, y, \ldots, z]$, $\varepsilon =$ 近似の精度；

出力： $G = \text{approx-GCD}(P_1, P_2, \varepsilon)$；

Step 1： 法 $S = (y - b, \ldots, z - c)$ の選択, $b, \ldots, c \in \mathbf{C}$；

Step 2： 1 変数多項式 $P_1(x, b, \ldots, c), P_2(x, b, \ldots, c)$ の近似 GCD：
$G_0(x) = \text{approx-GCD}(P_1(x, b, \ldots, c), P_2(x, b, \ldots, c), \varepsilon)$；

Step 3： $G_0(x), H_0(x) = P_1(x, b, \ldots, c)/G_0$ による一般化 Hensel 構成：
$P_1(x, y, \ldots, z) \equiv G_k(x, y, \ldots, z) H_k(x, y, \ldots, z) \pmod{S^{k+1}}$；

Step 4： 精度 ε で G_k が P_1 を割り切れば、G_k を approx-GCD として返す。

近似因数分解について：1 変数多項式の場合は与式の根を指定された精度で数値的に計算すればよく、多変数多項式の場合には [10] で提案された算法のうち、概略以下の原始的バージョンがインプリメントされた。

多変数近似因数分解の概略（モニックな場合）

入力： $P(x, y, \ldots, z) \in \mathbf{C}[x, y, \ldots, z]$, $\varepsilon =$ 近似の精度；

出力： 精度 ε での近似因子の組 $[Q_1, \ldots, Q_r]$；

Step 1： 法 $S = (y - b, \ldots, z - c)$ の選択、P は法 S で無平方とする；

Step 2： 1 変数多項式 $P(x, b, \ldots, c)$ の根を精度 ε で決定する：
$P(x, b, \ldots, c) = (x - u_1) \cdots (x - u_n) \quad (\text{acc } \varepsilon)$；

Step 3： S を法として各因子を並列 Hensel 構成する：
$P \equiv (x - U_1(y, \ldots, z)) \cdots (x - U_n(y, \ldots, z)) \pmod{S^{k+1}}$；

Step 4： $\{1, x - U_1, \ldots, x - U_n\}$ から 2 個以上の要素の積を作り、
精度 ε で P を割り切るものを近似因子とする。

上記の算法では P はモニックと仮定されている。モニックでない場合、[10] では

$$\tilde{P}(x, y, \ldots, z) = c^{n-1} P(x/c, y, \ldots, z), \quad c = \text{lc}(P)$$

なる有名な変換をすると述べられているが、これを行うと数式膨張が起きて効率を大きく損なう。我々はこの主係数問題を、1 変数多項式の場合に対する方法を拡張して、以下のように解決した。c は定数項を含むとする。$y, \ldots, z$ をべき級数変数とみなし、次の変換をする。

$$\tilde{P}(x, y, \ldots, z) = c^{-1} P(x, y, \ldots, z) \pmod{S^{k+1}}$$

すると、$\tilde{P}$ はモニックな多項式となる。並列 Hensel 構成を行ったあと、因子候補 Q_i を

$$Q_i \equiv c \times (x - U_{i1}) \cdots (x - U_{im}) \pmod{S^{k+1}}$$

と構成する。この Q_i で $c \times P(x, y, \ldots, z)$ を試し割りするのである。

6. おわりに

近似代数は、その提唱から８年を経たにすぎず、研究は始まったばかりである。素人目には、近似代数は単に多項式の係数を浮動小数にしただけと見えるかもしれない。しかし、従来の数式処理が整数・有理数の離散性に基づいていた（それが計算の厳密性を保証する）のに対し、近似代数は実数の連続性に基づく近似概念を取り込んだのである。しかも、近似の精度はマシンイプシロン程度である必要は毛頭なく、10^{-5} でも 10^{-2} でもよいのである。近似代数の扱いは計算機代数の扱いとはかなり異なるものとなるのである。

近似代数の研究では、種々の代数的演算に対する近似演算アルゴリズムの開発とともに、摂動項（誤差項）の持つ意味を明らかにし、その処理法を考案することが重要である。研究を進めてみると、近似代数の持つ豊富な内容に驚かされる。今後、世界の数式処理の流れが近似代数に向かうことは間違いないと思う。

我々は、昨年度までは専ら机上でアルゴリズムを考案していたが、本年度からは NSL-GAL にインプリメントを開始した。実際にアルゴリズムを稼働させてみると、ちょっとした工夫で計算が高速になったりする半面、思いがけない局面で計算が不安定になったこともあった。近似演算を用いるアルゴリズムはその安定化が重要な課題である。今後は、理論的な誤差解析も行いながら、実用に耐えるアルゴリズムを開発していこうと考えている。

参考文献

[1] 佐々木建昭、“近似的代数計算”、数理解析研究所講究録 676 号、pp. 307-319 (1988).

[2] G. Alefeld and J. Herzberger, *Introduction to Interval Computation* (trans. J. Rokne), Academic Press, New York, 1983.

[3] F. Kako and T. Sasaki, *Proposal of “Effective Floating-point Number” for Approximate Algebraic Computation*, preprint (in preparation).

[4] T. Sasaki and M. Sasaki, *Study of Approximate Polynomials I – Representation and Arithmetic –*, Japan J. Indus. Appl. Math. **12** (1993), pp. 137-161.

[5] D. Y. Y. Yun, *A p-adic Division with Remainder Algorithm*, ACM SIGSAM Bulletin **8** (1974), pp. 27-32.

[6] A. Terui and T. Sasaki, *Drawing Implicit Algebraic Functions on the Real Plane*, preprint (in preparation).

[7] Y. Ozaki and T. Sasaki, *Factor Separation of Univariate Polynomial and its Application to Multiple/Close Root Problem*, preprint (in preparation).

[8] T. Sasaki and M-T. Noda, *Approximate Square-free Decomposition and Root-finding of Ill-conditioned Algebraic Equations*, J. Inform. Process. **12** (1989), pp. 159-168.

[9] M. Ochi, M-T. Noda and T. Sasaki, *Approximate Greatest Common Divisor of Multivariate Polynomials and its Application to Ill-conditioned System of Algebraic Equations*, J. Inform. Process. **14** (1991), pp. 292-300.

[10] T. Sasaki, M. Suzuki, M. Kolar and M. Sasaki, *Approximate Factorization of Multivariate Polynomials and Absolute Irreducibility Testing*, Japan J. Indus. Appl. Math. **8** (1991), pp. 357-375.

[11] T. Sasaki, T. Saito and T. Hilano, *Analysis of Approximate Factorization Algorithm I*, Japan J. Indus. Appl. Math. **9** (1992), pp. 351-368.