数理解析研究所講究録
第1793巻 2012年 30-37

近似代数の汎用ライブラリに向けて*

長坂耕作
KOSAKU NAGASAKA†
神戸大学人間発達環境学研究科
GRADUATE SCHOOL OF HUMAN DEVELOPMENT AND ENVIRONMENT, KOBE UNIVERSITY

Abstract

近似GCDや近似因数分解などの近似代数演算（または数値数式融合計算，Symbolic-Numeric Algorithms for Polynomials）を，MathematicaやMapleなどの数式処理システム(CAS)上のパッケージとしてではなく，下位言語による汎用ライブラリとして実装する試みについて取り上げます。特に，そのような取り組みに至った経緯と，実際の実装を始めた直後である現在の状況について速報として報告します。

1 Introduction

科学研究費補助金の研究課題「多変数多項式の近似代数演算の実用化とその検証」において，2004年度から2006年度にかけ，Mathematica用のアドオンパッケージ「SNAP(Symbolic Numeric Algebra for Polynomials)」を開発しています[1)]。このパッケージの目的は，係数に誤差を含むような多項式（パッケージ中では，SNAP構造体と表記）を対象とした演算を包括的に行えうる関数を提供することでした。参考までに，主な特徴を以下に述べておきます。

- 次のような様々な近似代数演算（係数体は$\mathbb{R},\mathbb{C}$）の統合
  - 誤差を含む多項式同士の基本的な四則演算の提供（単変数，多変数）
  - 誤差を含む多項式の近似GCDの計算（単変数，多変数）
  - 誤差を含む多項式の近似因数分解（単変数，多変数）
  - 誤差を含む代数方程式の解の存在範囲の計算（0次元のみ，包括的Gröbner基底に基づく）
- 誤差情報を多項式に付与することで，計算結果の精度保証を実現

しかしながら，数式処理システムのユーザー言語を使用して実現したパッケージのため，パフォーマンスの点で改善すべきところが多々ありました。特に，Mathematicaに組み込まれている精度保証は，2次以上の誤差項を無視するため，実用性は高いものの数学的な厳密性には欠けます。開発したパッケージでは，これを厳密に行うために，精度保証をユーザー言語で上書きしており，著しくパフォーマンスに悪影響を与えています。これを改善するためには，ユーザー言語での実装でなく，下位言語によるライブラリといった形式での実装が必要になってくると考えられます。加えて，当然ではありますが，特定の数式処理システム用のパッケージである限り，そのシステムが動作には必須であり，利用可能な環境が制限されてしまい

*本研究の一部は科研費（22700011）の支援で行われている。
†nagasaka@main.h.kobe-u.ac.jp
1) 当時公開したバージョンは，技術的な理由からメンテナンスしておりませんが，ダウンロードは現在でも可能です。

ます。このような経緯から，2010 年度から取り組んでいる科学研究費補助金の研究課題「実践的な問題に即した近似代数計算の確立と実用化」においては，科学技術計算を行おうとする研究者や技術者に対して，実践的な近似代数の機能を提供するライブラリを開発しようとしています。本報告は，その途中経過（実際の作業を始めた直後の経過）の速報となります。

1.1 近似代数の汎用ライブラリ

本研究で取り組んでいる近似代数の汎用ライブラリとは，概ね次のようなものとなります。

- 基本的な近似代数演算（係数体は $\mathbb{R}, \mathbb{C}$）としての，近似 GCD と近似因数分解
- 近似 GCD と近似因数分解の有理整数環 ($\mathbb{Z}$) 上の多項式への拡張の組み込み
- 近似 Gröbner 基底（行列を用いた Structured Gröbner 基底としての実現）
- C/C++で実装され，C/C++から容易に利用可能なライブラリとしての提供

素朴な疑問として，そもそもこのような近似代数の汎用ライブラリは本当に必要なのか，というのが挙げられると思います。一般的に考えますと，現在のところ代数演算を行うユーザは，Mathematica や Maple などの CAS を使用していると思われますので，汎用的な計算代数や近似代数のライブラリは必要ないかもしれません。また，記号的な処理が必要となるような近似代数演算を，C/C++で組んでいるプログラムからライブラリ経由で呼び出す必要性は，広範囲の近似代数演算を行うことがない限り，必ずしも高くない可能性もあります。しかしながら，現実的な問題を代数的に取り組む場合には，誤差を含む計算は不可避となりますので，今後の利用価値は非常に高いと考えられます。

加えて，筆者が興味を持って取り組んでいる研究課題について言及しますと，計算コストが高いものが多く，CAS 上のユーザ言語での実装には計算速度面で力不足が否めません。例えば，整数係数多項式の整数上の近似 GCD においては，格子算法（LLL アルゴリズムないしはその亜種）を何度も呼ぶ必要性がありますし，構造化 Gröbner 基底においては，比較的大きなサイズの行列に対する特異値分解を何度も計算する必要性があります。これらの研究成果を実用的なレベルに向上する 1 つの方法としては，ユーザ言語ではなく汎用ライブラリとして実装することもあり得ると考えています。特に，筆者が頻繁に用いている Mathematica には，浮動小数点数を用いた LLL アルゴリズムの亜種は実装されていないようですし，巨大な行列を構成する際に必要となる多項式の単項式操作は必ずしも高速には実現されていません。このような背景から，近似代数の汎用ライブラリには，一定の需要があると考えられます。

そこで，本発表においては，CAS 上のユーザ言語での実装を日常的に行っているものとして，汎用ライブラリとして実現することによるメリット（既に言及）と，その手法について確認するために短期的に実施した試験実装について報告を行います。実際には，以下の論文等で発表しているアルゴリズムの簡易的な実装に取り組んだ結果として，判明した CAS 上への実装との違いについて記していきます。なお，これらの発表で提案したアルゴリズムの Mathematica 上への実装は，ウェブサイトで公開しておりますので，その URL も併記しておきます。

- K. Nagasaka. Approximate polynomial gcd over integers. ACM Communications in Computer Algebra (ISSAC 2008 Poster Session), 42(3):124–126, 2008.
  (URL: http://wwwmain.h.kobe-u.ac.jp/~nagasaka/research/snap/issac08.nb)
- K. Nagasaka. Approximate polynomial gcd over integers. J. Symbolic Comput. (in press).
  (URL: http://wwwmain.h.kobe-u.ac.jp/~nagasaka/research/snap/issac08plus.nb)

- K. Nagasaka. An improvement in the lattice construction process of approximate polynomial gcd over integers. In: Proceedings of Symbolic-Numeric Computation (SNC2011). 63–64, 2011. (extended abstract).
  (URL: http://wwwmain.h.kobe-u.ac.jp/~nagasaka/research/snap/snc2011.nb)

2 Survey

筆者は繰り返しになるが，基本的に CAS 上へのアルゴリズムの実装を行っており，残念ながら，研究開発レベルでの C/C++へのスクラッチからの実装については余り行ってきていません。日常的に使用するのは Mathematica であり，プログラム言語と開発環境としては Microsoft Visual C#だけです[2)]。そのため，これまでのパッケージ開発では考慮する必要のなかった，入出力やデータ表現の仕様などの初歩から検討する必要性があります。特に，近似代数演算とは言え，整数係数多項式の整数上の近似 GCD の計算においては，多倍長整数演算が必要であり，格子算法においては，浮動小数点数の高速な線形演算が必要になることから，これらを自ら実装するのか，別の汎用ライブラリを用いるのか，用いるとすればどのライブラリを用いるのか，といった点から検討する必要がありました。そこで，数式処理関係の国際学会などで比較的使用されていると考えられる次のような既存のライブラリやプログラムを参考にすることにしました。

NTL 数論に関するアルゴリズムの高速計算ライブラリ

- 著者: Victor Shoup
- ウェブサイト: http://www.shoup.net
- ライセンス: GPL2 / 多倍長演算: 独自 xor GMP

MPSolve 多倍長浮動小数点数を用いた単変数多項式の求根プログラム

- 著者: Dario Andrea Bini and Giuseppe Fiorentino
- ウェブサイト: http://www.dm.unipi.it/cluster-pages/mpsolve/
- ライセンス: 独自 / 多倍長演算: GMP

Linbox 厳密な線形演算を行う高速ライブラリ

- 著者: たくさん (E. Kaltofen, M. Giesbrecht, D. Saunders など)
- ウェブサイト: http://linalg.org
- ライセンス: LGPL / 多倍長演算: GMP (BLAS も)

CoCoALib 可換代数用のライブラリ（CoCoA のバックエンド）

- 著者: John Abbott など
- ウェブサイト: http://cocoa.dima.unige.it
- ライセンス: GPL3 / 多倍長演算: GMP

これらのうち，NTL で利用可能な主なデータ型は次の表に記載されているものです。

[2)]実際には，システム管理上の必要性から C や PHP などの，オープンソースで開発されているシステムで幅広く利用されている言語や，大学で担当している授業での必要性から，Sage などのオープンソースの CAS についても扱っていますが，書籍を出すレベルの Mathematica に比べると，その実装に関するレベルは決して高いとは言えません。

ZZ	big integers	ZZ_p	big integers modulo p
zz_p	integers mod "single precision" p	GF2	integers mod 2
ZZX	univariate polynomials over ZZ	ZZ_pX	univariate polynomials over ZZ_p
zz_pX	univariate polynomials over zz_p	GF2X	univariate polynomials over GF2
ZZ_pE	ring/field extension over ZZ_p	zz_pE	ring/field extension over zz_p
GF2E	ring/field extension over GF2	ZZ_pEX	univariate polynomials over ZZ_pE
zz_pEX	univariate polynomials over zz_pE	GF2EX	univariate polynomials over GF2E

表 1: NTL の主なデータ型

このうち，整数係数多項式の整数上の近似 GCD に関連するデータ型として，ZZX: $\mathbb{Z}[x]$ について，NTL/ZZX.h より抜粋したものが右の Class 定義です。参考にしようかと思いましたが，NTL は非常に作り込まれており，NTL/vector.h や NTL/ZZ.h などを順次調べないと詳しい構造は理解できず，今回の発表には間に合いませんでした。

```
class ZZX {
  public:
    vec_ZZ rep;
    (データ部はこれだけなので，以下省略)
```

Linbox や CoCoALib もライブラリとしての規模が大きく，参考にするには複雑なため，多項式の求根という限られた機能に特化している MPSolve について詳しく述べておきます。MPSolve で許容される入力形式は次のようになっています。

多項式の表現形式	
s	sparse polynomial
d	dense polynomial
u	polynomial provided by the user

多項式の係数種別	
r	real coefficients
c	complex coefficients

多項式の係数表現	
i	integer real/imaginary parts
q	rational real/imaginary parts
b	bigfloat real/imaginary parts
f	float real/imaginary parts

表 2: MPSolve の入力形式

多項式の表現は，mps_poly.c で次のように定義される構造体__mpspoly_struct で行われています。

```
typedef struct {
  int deg;              /* starting polynomial degree */
  char data_type[3];    /* polynomial data type */
  long int prec_in;     /* number of digits of input precision */
  int n;                /* degree */
  boolean *spar;        /* sparsity structure of the polynomial */
  double *fpr;          /* standard real coefficients */
  cplx_t *fpc;          /* standard complex coefficients */
  rdpe_t *dpr;          /* dpe real coefficients */
  cdpe_t *dpc;          /* dpe complex coefficients */
  mpz_t *mip_r;         /* real part of integer input coefs */
```

```
  mpz_t *mip_i;        /* imaginary part of integer input coefs */
  mpq_t *mqp_r;        /* real part of rational input coeff. */
  mpq_t *mqp_i;        /* imaginary part of rational input coefs */
  mpf_t *mfpr;         /* multiprecision real coefficients */
  mpc_t *mfpc;         /* multiprecision complex coefficients */
} __mpspoly_struct;
```

非常に単純明快な表現方法であり，本稿でも同じように係数を配列として保持することにしました。なお，MPSolve では疎な多項式も表現可能になっていますが，本稿で扱う整数制約のある近似 GCD においては，摂動後の多項式は密になる可能性が非常に高いため，当座，密な多項式のみを対象としました。

3 Specification

前述のシステムなどを参考に，最終的に本稿ではライブラリの内部表現を次のように定めました。

多倍長整数	GMP の mpz_t
多倍長有理数	GMP の mpq_t
多倍長整数を係数とする単変数多項式	STL の vector で vector<mpz_t*>
多倍長整数を要素とするベクトル	GMP の mpz_t の 1 次元配列
多倍長整数を要素とする行列	GMP の mpz_t の 1 次元配列
多倍長有理数を要素とするベクトル	GMP の mpq_t の 1 次元配列
多倍長有理数を要素とする行列	GMP の mpq_t の 1 次元配列

表 3: 本稿におけるデータ型

なお，メモリ管理を行って配列で表現することも考えましたが，時間的制約から実装が容易な面を考慮して，C++の STL の vector を使っています。ただし，最新の手元の実装では，配列の利用に変更していることを申し添えておきます。

3.1 Implementation

今回は，Microsoft Visual C++に，Intel MKL(Math Kernel Library) を加えた環境において，次のような多項式関連の関数などを実装しました。本来は，Linux 上で GCC の C コンパイラと GMP のみを使用する予定でしたが，実装作業が可能な期間において利用可能な開発環境に制限があり，このような形での実装となりました。なお，Intel MKL を使わずに，VC++と GMP のみを使用することも可能でしたが，次のような理由から今回は見送りました。理由は 3 つあり，(1) GMP, BLAS, LAPACK などが組み込まれており，近似代数演算に必要となる基礎的な演算に関する環境を，容易に整えることが可能なこと，(2) 主要なプラットフォーム（Windows, Linux, MacOSX）をカバーしていること，(3) Intel Compiler を使った方が速度面で有利であると考えられること，です。

```
typedef std::vector<mpz_t*> upz_t;
void upz_init(upz_t&);
void upz_init(upz_t&, int);
void upz_init_allzero(upz_t&, int);
void upz_clear(upz_t&);
```

```
void upz_set(upz_t&, upz_t&);
void upz_set(upz_t&, mpz_t*, int);
void upz_set(upz_t&, mpz_t*, int, int);
int upz_set_str(upz_t&, char*, char);
void upz_setc(upz_t&, mpz_t, int);
```

```
void upz_getc(upz_t&, mpz_t, int);    void upz_mul(upz_t&, upz_t&, upz_t&);
void upz_setc_si(upz_t&, signed long int, int);
signed long int upz_getc_si(upz_t&, int);
void upz_canonicalize(upz_t&);        void upz_self_mul(upz_t&, upz_t&);
void upz_1norm(upz_t&, mpz_t);        void upz_mul_si(upz_t&, upz_t&, signed long int);
void upz_c2norm(upz_t&, mpz_t);       void upz_self_mul_si(upz_t&,signed long int);
void upz_f2norm(upz_t&, mpz_t);       void upz_add(upz_t&, upz_t&, upz_t&);
void upz_inorm(upz_t&, mpz_t);        void upz_sub(upz_t&, upz_t&, upz_t&);
```

これらの基本的なデータ型に対応する関数を用いて，以下のような格子算法と整数係数多項式の整数上の近似 GCD を計算する関数を準備しました。今回は，C++で記述したため，それぞれを public メソッドを 1 つずつ持つクラスとして実装しています。

```
class LLL {
  private:
    static int mpq_cmp2x(mpq_t, mpq_t);
    static void mpq_sub_z(mpq_t, mpq_t, mpz_t);
    static void mpq_get_z(mpz_t, mpq_t);
    static void mpq_mul_z(mpq_t, mpq_t, mpz_t);
    static void mpq_inner(mpq_t, mpq_t *, int, int, int);
    static void mpq_inner_z(mpq_t, mpq_t *, int, mpz_t *, int, int);
    static void gramschmidt(mpz_t *, mpq_t *, mpq_t *, int, int, int, int);
  public:
    static int main(mpz_t *, int, int);
};
```

```
class KN201XJSC {
  private:
    static void makeH(mpz_t*, int&, int&, upz_t&, int&, upz_t&, int&, upz_t&, int&,\
                                                          upz_t&, int&, int&, mpz_t);
    static void makeL(mpz_t*, int&, int&, upz_t&, int&, upz_t&, int&, int&, mpz_t);
    static void knuth_bound(mpz_t, upz_t&, upz_t&);
    static void candidate_cofactors(std::vector<int>&, mpz_t*, int&, int&, upz_t&, \
                                                    int&, upz_t&, int&, int&, mpz_t);
    static void candidate_gcds(std::vector<int>&, mpz_t*, int&, int&);
  public:
    static int main(upz_t&, upz_t&, upz_t&, upz_t&, upz_t&, mpz_t&);
};
```

3.2 実装上の要改善点

実装を行った格子算法を念のため説明しておきます。$\vec{b}_1, \ldots, \vec{b}_n \in \mathbb{R}^m$ の格子とは，$\left\{ \sum x_i \vec{b}_i \,\middle|\, x_i \in \mathbb{Z} \right\}$ のことです。このような格子が与えられたときに，格子に含まれる最短ベクトルを探す問題を「SVP」，別途

与えられたベクトルに最も近いベクトルを格子から探す問題を「CVP」と言います。これらの問題に対して，近似的な解を与える有名なアルゴリズムが，LLL アルゴリズム（ないしは L^3 アルゴリズム）です。

今回実装したクラスのメソッド main は，1 次元配列で格子基底を受け取り，LLL 簡約基底を計算します（整数係数多項式の係数ベクトルから構成される基底ベクトルは，整数のみを含みますので，整数の配列となります)。一見すると整数演算のみのように思えますが，LLL アルゴリズムは内部で Gram-Schmidt の直交化を行い直交化係数を求める必要性があります。従って，この過程において多倍長有理数 mpq_t に関連した演算や比較関数などが必要になります。しかしながら，今回の発表に向けての開発は前述の通り，Microsoft Visual C++に Intel MKL を加えた環境で行ったわけですが，GMP の多倍長有理数の型である mpq_t は Intel MKL に入っていません。これは想定外の出来事で，意図せず自分で mpq_t を実装するはめになりました。結果として，以下のような関数を実装するという余計な作業が必要となってしまいました。

```
typedef struct {
  __mpz_struct num;
  __mpz_struct den;
} mpq_struct;
typedef mpq_struct mpq_t[1];

void mpq_init(mpq_t);
void mpq_clear(mpq_t);
void mpq_set(mpq_t, mpz_t, mpz_t);
void mpq_set_si(mpq_t, signed long int, unsigned long int);
void mpq_canonicalize(mpq_t);
```

```
void mpq_add(mpq_t, mpq_t, mpq_t);
void mpq_sub(mpq_t, mpq_t, mpq_t);
void mpq_mul(mpq_t, mpq_t, mpq_t);
void mpq_div(mpq_t, mpq_t, mpq_t);
void mpq_neg(mpq_t, mpq_t);
void mpq_abs(mpq_t, mpq_t);
void mpq_inv(mpq_t, mpq_t);
double mpq_get_d(mpq_t);
int mpq_cmp(mpq_t, mpq_t);
void mpq_swap(mpq_t, mpq_t);
```

その他，次のようなことに戸惑いましたので報告しておきます。多倍長有理数 mpq_t 関連の実装においては，GMP の mpq_t に関するソースは参考にせずに，しかしプロトタイプ宣言は一致するように組みました。しかし，車輪の再発明はするものではなく，浮動小数点数への変換方法や負数の商の符号を間違えるなど，原因に気がつかずデバッグに半日以上費やすこともありました。単変数多項式を表現した upz_t 関連の関数の実装においては，使い慣れない STL の vector 関係で原因が突き止められないエラーに遭遇しました。1 つは，Intel MKL に含まれる GMP の mpz_t に対して，std::vector<mpz_t*> は問題なく利用できるのですが，自前実装の mpq_t に対して，std::vector<mpq_t*> を行うとエラーになってしまうことで，もう 1 つは，なぜか std::vector<mpz_t*> を配列で渡すとメモリエラーとなってしまうことです。やはり，C++ではなく，旧来の C90(C89) やせめて C99 で記述した方が筆者には合っているようです。

3.3 整数制約のある近似 GCD

整数制約のある近似 GCD は，次のような互いに素な多項式が与えられた場合に，係数を整数上で摂動させることで，自明でない GCD とそれを持つ多項式を求めることを言います。

$$f(x) = 968x^3 + 2622x^2 + 357x - 1576,\ g(x) = 595x^4 + 1326x^3 - 225x^2 + 65x + 1277$$

例えば，この場合は，次のような形の近似 GCD を持っています。

$$f(x) = (23x + 51)(42x^2 + 21x - 31) + 2x^3 - 3x^2 - x + 5,$$
$$g(x) = (23x + 51)(26x^3 - 10x + 25) - 3x^4 + 5x^2 + 2$$

本稿では，この近似 GCD に関して次の定義を用いています。

定義 1 (Approximate GCD Over Integers)
*Let $f(\vec{x})$ and $g(\vec{x})$ be polynomials in variables $\vec{x} = x_1, \ldots, x_\ell$ over $\mathbb{Z}$, and let ε be a small positive integer. If they satisfy $f(\vec{x}) = t(\vec{x})h(\vec{x}) + \Delta_f(\vec{x})$, $g(\vec{x}) = s(\vec{x})h(\vec{x}) + \Delta_g(\vec{x})$ and $\varepsilon = \max\{\|\Delta_f\|, \|\Delta_g\|\}$ for some polynomials $\Delta_f, \Delta_g \in \mathbb{Z}[\vec{x}]$, then we say that the above polynomial $h(\vec{x})$ is an **approximate GCD over integers**. We also say that $t(\vec{x})$ and $s(\vec{x})$ are **approximate cofactors over integers**, and we say that their **tolerance** is ε. ($\|p\|$ denotes a suitable norm of $p(\vec{x})$.)* ◁

今回の実装では，多項式をいくつか受け取って近似GCDを計算する本体の他，ある種の特殊な行列を生成する関数`makeH`や`makeL`，近似GCDなどに対応する短いベクトル候補を選別する関数`candidate_cofactors`や`candidate_gcds`，そして多項式の2ノルムでの因子係数上限を求める関数`knuth_bound`を作成しています。実装とは話が離れますが，本稿で採用している以下の因子係数上限について補足をしておきます。

定理 2 (Knuth's bound)
多項式 $f, g, h \in \mathbb{C}[x_1, \ldots, x_t]$ は，$f = gh$ を満たしていて，$m_i = \deg_{x_i} g$, $k_i = \deg_{x_i} h$ とする。このとき，

$$\|g\|_2\|h\|_2 \leq (\phi(m_1, k_1) \cdots \phi(m_t, k_t))^{1/2} \|f\|_2, \quad \phi(m, k) = {}_{2m}C_m \times {}_{2k}C_k$$

が成り立つ。また，この系として，$\|g\|_2\|h\|_2 \leq 2^{\deg_{x_1} f + \cdots + \deg_{x_t} f} \|f\|_2$ なる因子係数上限も成立する。 ◁

当初の筆者の近似GCDに関する論文では，この上限をGelfond's boundとして紹介/引用しておりましたが，正しくは，Knuth's boundと呼称すべきものでした。詳しくは，次の書籍のExercise 21, Section 4.6.2 (Factorization of Polynomials) をご覧ください。ただし，最新版の第三版では別の内容に変更されていますので，証明などを参照する場合は，必ず第二版をご参照ください。筆者はこれに気がつかず，類似の上限をいくつか示しているGelfondを出典だと誤解しておりました。

- Donald E. Knuth. The Art of Computer Programming. Volume 2 / Seminumerical Algorithms. SECOND EDITION.

4 Remarks

数理解析研究所での発表に際しては，ライブラリの使用例として，入力した多項式の近似GCDを実際に計算を行う表示するプログラムにおいて，実装したライブラリの関数を呼び出すデモンストレーションを行いましたが，(少なくとも発表当時のバージョンに関しては) これらは現在のところ公開の予定はありません。なお，デモンストレーションで示したように，既に公開しているMathematica向けの実装に比べて，計算速度の向上は見られませんでした。これは，簡易的な実装が今回の目的でしたので，特にLLLアルゴリズムの高速算法を実装していないことが原因と思われます。そのため，今後の実装では，LLLアルゴリズムに関して知られている比較的新しい高速なアルゴリズムを使用する予定です。本研究の計画では，今年度はライブラリ構築の下調べの年であり，来年度に向けて色々と実験を行っている段階となります。その結果に基づいて，来年度以降に本格的な実装作業を進める予定となっておりますので，公開はそれ以後となる計画です。