有理数線形代数計算における有理数BLASの提案

寒川　光[a)]

概要：数値計算は浮動小数点演算によって実装されてきた．そのためアルゴリズムは，丸め誤差を制御する方法を取り入れつつ発達した．一方，有理数計算は，計算機科学の黎明期から研究されてきたが，数値計算には，計算機の性能が不十分であったため，実用的な製品としての実装は少ない．初代スパコン CRAY-1 と現在最速のスパコンを比較すると 1 億倍以上の性能差がある．並列システムの計算能力の発展が続くなら，浮動小数点演算による数値計算は，徐々に有理数演算に置き換えられてゆく可能性がある．有理数計算は正確な計算結果を提供するので，現在の数値計算のアルゴリズムの内，丸め誤差の影響を制御する部分は不要になる．浮動小数点計算では主流である「直交化を基礎とする解法」は，有理数計算では桁数が膨大になるため，直接的な解法に劣る．このため有理数計算による数値計算アルゴリズムのメニューは，浮動小数点演算用のものと異なる．本論文では，線形代数計算を有理数計算で行うためのプログラミング環境を，多桁整数演算を実現する階層，有理数演算を実現する階層の上に，BLAS に対応する「有理数 BLAS」階層を構築することで，既存の浮動小数点計算用のプログラムを，有理数計算環境に移行する方法を提案する．

キーワード：有理数計算，有理数 BLAS，共通因子の抽出，数値線形代数，共役勾配法

A Proposal of Rational BLAS for Numerical Linear Algebra with Ratinal Number Arithmetic

Hikaru Samukawa[a)]

Abstract: Numerical methods have been developed based on floating-point arithmetic. As a result, the algorithms have grown with adopting capabilities to control rounding errors. In contrast, though rational number arithmetic has been a theme of computer science from early period of computer, there are a few implimentations for numerical methods as products. The reason is it requires huge computational resouce. The firstest computer today bocomes more than one hundred million times faster than the first supercomputer CRAY-1. If the advancement of the computer power continues for decades, numerical methods with floating-point arithmetic may gradually be replaced with rational number arithmetic. Since rational number arithmetic provides exact computation, a portion to control rounding errors becomes no use. The algorithms based on orthogonalization should handle large number of digits in rational number arithmetic, it is inferior compared to direct solving algorithms. The menu of numerical algorithms with rational number arithmetic becomes quite different to that of floating-point arithmetic. In this paper, a programming environment for numerical linear algebra by rational number arithmetic is reported. We propose a rational BLAS layer heaped up on the multi-digit integer layer and the rational number arithmetic layer, which attains easier conversion from floating-point arithmetic to rational number arithmetic.

Keywords: rational number arithmetic, rational BLAS, common factor extraction, numerical linear algebra, conjugate gradient method

1. はじめに

数値計算の多くは浮動小数点演算によって行われており，計算の効率は高く，融通がきくが，信頼性は低い．数値計算と対照的な信頼性の高い方法に数式処理がある．数式処理は数値計算の前処理として利用されることも多い[*1]．両者を比較すると

計算方式	効率	信頼性	融通性
数値計算	高	保証が必要	高
数式処理	低	高	低

のようにまとめられる．数値計算と数式処理の融合を目指す「数値数式融合計算」の試みもあり，数式処理に数値計算を組み込むことで，数式処理の計算効率を高める方向で研究されている [1].

本論文で扱う，有理数計算は**信頼性の高い数値計算**である．数値計算の世界に誤差のない計算を導入することで，次のような目的での使用を想定している [2].

- 丸め誤差の理論を学ぶ前の学生を対象とした数学教育[*2]
- 浮動小数点演算による数値計算で発生した精度に関係する問題を分析するツール
- 2 つのアルゴリズムの計算結果の一致を確かめることによる計算機援用証明（Computer Assisted Proof）
- 数値計算ライブラリ開発時に使用する高精度計算を，数値計算の延長として行う[*3]
- ベンチマークの検収など，計算機システムの稼働確認[*4]

「数学教育」だけが目的なら十進 BASIC でよいが，2 番目の目的では，有理数と浮動小数点数を混在させることが必要となる．BASIC の言語規格では「数値の表現が 1 種類」に限定される．我々は，「有理数と浮動小数点数を混在して自由にプログラミングできる計算環境」を C++ で開発している[*5].

本論文では，第 2 章でこの計算環境の概要を説明する．有理数計算の最大の障害は計算時間にあるので，高速化の実現が必須ある（HPC の知識が重要である）．高速化は，アルゴリズムの選択，有理数を構成する分数の分母と分子の整数の桁数削減，並列化の 3 者が鍵である．第 3 章では，桁数がどのように推移するかを述べ，数値線形代数で扱うベクトル演算で，ベクトルから共通因子を括り出すことで達成される桁数の削減とその効果を述べる．第 4 章でまとめる．

2. 有理数計算プログラミング環境

有理数 rational number は，2 つの整数の比 ratio で表される数である．一般のプログラミング言語で扱える整数の範囲は限定されたものなので，有理数演算は，分母，分子を多桁の整数で保持する．r_1, r_2 を有理数，a, b, c, d を多桁整数とすると，四則演算を，

$$r_1 \pm r_2 = \frac{b}{a} \pm \frac{d}{c} = \frac{bc \pm ad}{ac} \quad (1)$$

$$r_1 \times r_2 = \frac{b}{a} \times \frac{d}{c} = \frac{bd}{ac} \quad (2)$$

$$r_1 \div r_2 = \frac{b}{a} \div \frac{d}{c} = \frac{bc}{ad} \quad (3)$$

によって行う．n 桁と n 桁の整数の積は $2n$ 桁になるので，これらの式の ac などの整数の桁数は演算のたびに a や c よりも増えるが，分母，分子の最大公約数（GCD）で割り，既約分数にする．開発中のプログラミング環境は，上記の a, b, c, d などの多桁の整数を **longint** 型で扱う多桁整数の階層を持つ．その上に $r = \frac{b}{a}$ などの有理数を **rational** 型で扱う有理数の階層を置く．

計算機システムの高性能化に伴い，実用可能な問題の規模が徐々に広がりつつあることを考慮し，応用分野として数値線形代数（Numerical Linear Algebra）を対象とする[*6]．有理数 BLAS（Rational Basic Linear Algebra，以下 rblas）階層を構築すると，有理数計算の高速化を行い易くなる．これらの 3 階層の上

[1] 芝浦工業大学
Sibaura Institute of Technology, Minuma, Saitama 337–8570, Japan
[a)] samukawa@sic.shibaura-it.ac.jp

[*1] 数式出力を Fortran 形式にして，Fortran プログラムのソースコードとして埋め込む．
[*2] 高校の数学教科書「数学 B」では，十進 BASIC を使用したものもある [4]．十進 BASIC では「有理数モード」を選択することで，有理数計算ができる [3].
[*3] 定積分公式の係数や補間多項式係数の正確な計算などの，数値計算ライブラリの開発に必要な高精度計算．
[*4] 現在よく行われる，擬似乱数で作成された係数行列に対する LU 分解では，行列サイズが 1000 万に達し，直接解法で得られた解の精度は上位数桁と言われている．これを 1 回だけ反復改良して，残差ベクトルのノルムが許容範囲に入るかどうか調べている．このような浮動小数点演算による方法では，下位の桁にわずかな計算機のエラーが現れても分からない．つまり，計算機システムに対する検収としては，エラーの検出能力が高くない．有理数計算による方法ではエラーの検出能力は高いので，次世代ベンチマークとして有力である [2].
[*5] 力学を学ばない数学科の学生が，HPC プログラミングを体験する教材にも使用可能と考えている．
[*6] 無理数の使用が比較的少ない分野でもある．

で行う数値計算プログラミングは，C++ で扱える変数型に加えて，多桁整数と有理数型の変数，有理数型ベクトルと行列を定義してプログラミングできる．

本章では，有理数計算プログラミング環境の階層構造を説明し，倍精度浮動小数点演算から有理数への型変換，rblas の既存 BLAS との相違点，並列化について述べる．

2.1 有理数計算プログラミング環境の階層構造

有理数の四則演算は $+, -, *, /$ の記号でプログラミングできる（オペレータオーバーロード）．浮動小数点計算で生じた問題の分析を，正確な解と比較することで問題解決の手掛かりとすることが目的ひとつである．したがって，倍精度浮動小数点数を有理数に変換するような処理が簡単にプログラミングできなくてはならない．このような機能は，longint クラス，rational クラス，rblas クラスを階層的に構成することで実現できる．

2.1.1 多桁整数（longint）クラス

多桁の正の整数を，基数を 2^{32} として 32 ビット符号なし整数型の配列に格納し，桁数と符号を別途保持する．配列長は 64 から始め，不足すると動的に倍，倍と拡張する可変長とした．最大長は 32,768 としている（10 進数で 31 万桁）[*7]．

四則演算は筆算で行うのと同様のアルゴリズムで計算するが，乗算は桁数が増えると，高速フーリエ変換（FFT）を用いることで高速化した．FFT は基数 2 のみを使用し，32 ビット整数を，基数が 2^{24} の単精度浮動小数点数に変換し，多倍長演算を，各桁では倍精度で計算する．通常の乗算と FFT 乗算の境界は，2^{24} 進数で $2^8 = 256$ 桁（10 進数で約 1850 桁）としている．

2.1.2 有理数（rational）クラス

longint 型の 2 つの数を分母 a と分子 b とする有理数 $r = \dfrac{b}{a}$ は `RRset(b,a)` によって rational 型の有理数にできる．ここで分母, 分子の GCD を, 四則演算後と同様に, 2 進 GCD アルゴリズムで GCD を求め, 既約分数とする．なお, GCD アルゴリズムは 2 進 GCD の, longint 型を返す関数 `lgcd2(longint,longint)` を longint クラスにもつ．この演算は教科書 [5] の多倍長演算と有理数演算にほぼ忠実に行った．

反対に，有理数の分母，分子を longint 型で取り出す関数は `denominator`，`numerator` とした．絶対値 `abs`，最大 `max`，最小 `min`，フロア `floor` などの関数もある．

浮動小数点数は有限のビット数で表される数なので，数学的には有理数である．IEEE 標準の浮動小数点数から有理数への変換の例を示す．倍精度浮動小数点数 a を，e を指数，d_i を 0 または 1 として 2 進数で次のように表す．

$$a = \left(\pm \sum_{i=0}^{52} d_i 2^{-i} \right) \cdot 2^e = A \cdot 2^e \qquad (4)$$

$B = A \cdot 2^{52}$ は 2^{54} よりも小さい整数なので，10 進数では 17 桁以下で表すことができる．B を用いると式 (4) の数は，$a = B \cdot 2^{e-52}$ である．$C = 52 - e$ とおくと $a = B \div 2^C$ である．B の最下位ビットが 1 であれば，分子は B，分母は 2^C と有理数表現される．B の下位 k ビットが 0 であれば，分子は $B \cdot 2^{-k}$，分母は 2^{C-k} と有理数表現される．

倍精度浮動小数点数 a は `Rdset(a)` によって rational 型の有理数にできる．

2.1.3 基本線形代数演算（rblas）クラス

このクラスで，有理数で構成されるベクトルと行列を，**vector** と **matrix** のデータ構造でサポートする．これにより行列演算が容易に記述できる．rblas は，浮動小数点計算と有理数計算の性質の違い，高速化のアプローチの違い，開発プログラミング言語の違いから，重要な点で既存の BLAS とは異なる．

rblas が対象とするメニューは，浮動小数点計算用の BLAS-1（dot，axpy など）と BLAS-2（gemv など）が対象である．浮動小数点計算でキャッシュブロック化を目的とした BLAS-3 は，有理数計算では 1 つの有理数要素が長い配列に格納されるため，同様の効果は期待できないので対象外とした．それに代わる高速化の重要なメニューが，**ベクトルから共通因子を抽出してベクトルをスケーリング**するルーチンであり，これを加えた（次章でこの点を述べる）．以下，rblas の特長を述べる．

行列の形状は，すべて長方形行列とした（疎行列や三角行列は零要素も含めて扱う）．有理数計算では，行列の 1 つの要素が占めるメモリ容量が可変長である．したがって零要素をデータ構造として扱うことでメモリ，計算時間を節約する利得は小さい．`matrix` はすべて $m \times n$ の形式に統一した．

既存の BLAS はレベル 1，2，3 に分かれて策定されている [6],[7],[8]．策定された時代は，ベクトル型のスーパーコンピュータの全盛時代である．この時代の Fortran の Sequence Association（SA）を前

[*7] $\log_{10} 4294967296 = 9.633\cdots$

提にインターフェースが定義された．したがって，Fortran のアドレス渡し（call by address）と，配列要素の内部での順序付けが言語規格のレベルで定義され，これを前提としている．これを前提に BLAS では，配列の途中の要素，先導次元，ストライドを渡すインターフェースを採用した．

rblas では参照渡し（call by reference）を使用するので，配列の途中の要素から始まる短いベクトルを渡すことや，行列の特定の列をベクトルとして渡す（2 次元配列を 1 次元配列で受ける）ことはできない．

BLAS-1 のベクトルをスケール倍するサブルーチン dscal を示す．

```
subroutine dscal(n,a,x,ix)
double precision x(n)
do i=1,n
  x(1+(i-1)*ix)=a*x(1+(i-1)*ix)
enddo
return
```

x には先頭要素，ix にはストライドを渡す．ガウス消去法でこれを呼び出す部分を示す．

```
subroutine dgauss (a,lda,n,ipvt)
double presision a(lda,n)
略
do k=1,n-1
  call dscal(n-k,1.d0/a(k,k),a(k+1,k),1)
  略
```

Fortran ではこの例のように，2 次元配列の要素を渡し，渡された側はこれを 1 次元配列で受けることができる．これは，アドレス渡しと SA が言語規格として定められているから可能になる．上の例で lda が先導次元（Leading Dimension of Array）であり，Fortran ではコンパイル時に lda の中身が確定していなくても，上記のコードは稼働する．C ではコンパイル時にこれが確定している必要があるので，このインターフェースを踏襲することは，やや複雑になる．

有理数計算プログラミング環境の開発では，十進 BASIC の有理数モードとの計算結果の比較を使用しているので，十進 BASIC とのループ構成や数式の一致が重要である．このため，途中の要素から始まるベクトルを使用しているものや，2 次元配列を渡し，1 次元配列で受ける使用法は，プログラムを変更して調整している．

内積 dot を例として，rblas のプログラム例を示す．

```
    #include "rblas.h"
    rational rblas::rdot(int n, const
    rational_vector& x, int ix, const
    rational_vector& y, int iy){
       rational s;
       s = rational::ZERO;
       for (int i=0; i < n; ++i) {
           s = s + x[i*ix]*y[i*iy];
       }
       return s;
    }
```

このルーチンそのものは，BLAS-1 の dot と機能的に同じであるが，BLAS-1 では，行列の行ベクトルと列ベクトルの内積を計算できる．rblas では，2 次元配列を 1 次元配列では受けられないので，中間ルーチンを介するなどの調整が必要である（例を後述する）．

2.2 スレッド並列化

GNU/Linux には，複数のスレッド間で同期をとる操作として mutex（mutual exclusion），セマフォ（semaphore），読み取り・書き込みロック（read-write lock），バリア（barrier），条件変数，スピンロックなどが用意されている．有理数計算プログラミング環境では，セマフォを用いた並列化を実装している．

3. 有理数計算の桁数削減による高速化

共役勾配（Conjugate Gradient: CG）法は，疎行列を係数行列とする連立 1 次方程式の解法として，最も一般的に使用される．線形最小 2 乗法 $A\boldsymbol{x} \cong \boldsymbol{b}$ の解法では，正規方程式 $A^TA\boldsymbol{x} = A^T\boldsymbol{b}$ を解く方法は，浮動小数点演算の丸め誤差の影響を敏感に受けるので「数値計算の常識」として使ってはならないと教えられる [10]．精度的に安定な長方形行列 A を直交分解 $A = QR$ する方法が一般的に用いられる．これらのアルゴリズムが効果的であるのは，永年の浮動小数点演算の丸め誤差に対する研究成果によるところが大きい[*8]．一方，有理数演算では，丸め誤差が現れないので，計算精度と計算時間に関する振舞いが根本的に異なる．したがって，浮動小数点計算で培われたアルゴリズム選択の常識が通用しない．本章では，正確な計算を実現するための桁数の増加の推移を，三角分解と直交化を基礎とする CG 法，シュミットの直交化のアルゴリズムについて調べた結果を報告する．

[*8] したがって丸め誤差の理論を学ぶ前の学生には荷が重く，数値計算が人気のない科目となっている．

3.1 プログラムと桁数の推移の例

正則な正方行列 A は $A = LU$ と，単位下三角行列 L と上三角行列 U の積に分解される（ガウスの消去法）．行列 A が対称の場合，上三角行列の要素 u_{ij} と，下三角行列の要素 l_{ij} の間では，$l_{ji} = u_{ij}/u_{ii}$ の関係がある．この関係を行列によって表すと，

$$A = LDL^T \tag{5}$$

となる（D は対角行列で，$U = DL^T$）．この定式化を修正コレスキー分解（modified Cholesky factorization，LDL 分解）という．ここでは下三角要素を転置して $t_{ij} = l_{ji}$ によって L^T の要素を表す．

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} t_{ki}u_{kj}, \qquad (i \le j) \tag{6}$$

$$t_{ij} = \frac{u_{ij}}{u_{ii}} \qquad (i < j) \tag{7}$$

プログラミング例を示す．$n \times n$ の対称行列 A と n を入力に渡すと，対角行列 D の対角項と上三角行列 L^T の対角項以外の項を，行列 A の要素の渡された領域に上書きして返す関数 LDL を示す．

```
void LDL(rational_matrix& a, int n){
    rational s,t;
    int i,j,k;
    for (j=1; j < n; ++j) {
        for (i=1; i < j; ++i) {
            s=0;                              //
            for (k=0; k < i; ++k) {           //
                s = s + a[k][i] * a[k][j]; //
            }                                 //
            a[i][j] = a[i][j] - s;            //
        }
        s=0;
        for (k=0; k <= j-1; ++k) {
            t = a[k][j]/a[k][k];
            s = s + t * a[k][j];
            a[k][j] = t;
        }
        a[j][j] = a[j][j] - s;
    }
}
```

このプログラムは，通常の数値計算の教科書の LDL 分解と同じループ構成である [9]．浮動小数点計算の BLAS-1 を用いた Fortran プログラムでは，“//” で印を付けた 5 行は “a(i,j)=a(i,j)-ddot(i-1,a(1,i),1,a(1,j),1)” によって記述できる．前章で述べたように，SA を前提としないので，この例のような調整（dot 関数のインライン展開）が必要になる．

表 1 係数行列の比較（$n = 100$）

係数行列	時間（秒）
フランク行列	0.7
ヒルベルト行列	3.8
浮動小数点ヒルベルト	92.6
$\lvert a_{ij}\rvert \le 0.5$ の乱数	989.3

データ型が倍精度浮動小数点数でなく **rational** 型であるので，計算機が行う計算は大きく異なる．引数 a の matrix A と，2 つの変数 s と t が rational 型である．“a[i][j]=a[i][j]-s” などの有理数演算は，動的にメモリを拡張しつつ，正確な計算を実行する．

桁数の増加は，データ依存性が強い．係数行列を，フランク行列 $a_{ij} = n - \max(i,j) + 1$，ヒルベルト行列 $a_{ij} = \dfrac{1}{i+j-1}$，浮動小数点数に丸めたヒルベルト行列，$|a_{ij}| \le 0.5$ の倍精度浮動小数点乱数の 4 つの場合の，次数 $n = 100$ での比較を表 1 に示す．計算機は Intel の Core i5（2.67GHz）を搭載したノートパソコンで，1 スレッドのみを使用した．このように，計算時間は $O(n^3)$ から $O(n^5)$ に近い状態に分布する．この理由は論文に詳しい [2]．

三角行列の要素は，式 (6) の右辺にある内積で生成される．次数 10 の乱数による対称行列を LDL 分解したときの，対角項 D と上三角行列 L^T の要素の，**分母と分子の 2^{32} 進数での桁数（以下「桁数」と記す）の和**を示す．

$$\begin{pmatrix}
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
 & & 12 & 12 & 12 & 11 & 12 & 12 & 11 & 11 \\
 & & & 15 & 15 & 14 & 14 & 14 & 14 & 15 \\
 & & & & 18 & 18 & 18 & 18 & 18 & 18 \\
 & & & & & 22 & 22 & 22 & 22 & 22 \\
 & & & & & & 26 & 26 & 26 & 26 \\
 & & & & & & & 28 & 28 & 29 \\
 & & & & & & & & 32 & 32 \\
 & & & & & & & & & 36
\end{pmatrix}$$

1 行目は a_{1j} を d_1 で割って u_{1j} が得られるので，桁数の和は 4 のままである．2 行目は l_{21} と u_{1j} の積（4 桁同士）があるので，桁数の和は増加して 8 になる．3 行目は，4 桁同士の積と 8 桁同士の積の和で，約 12 桁になる．このように，l_{ij} の桁数は，行位置 i にほぼ線形に増加する[*9]．この理由は，桁数が行位

[*9] フランク行列の場合，桁数はすべて 2 であり，ヒルベルト行列の場合は 10 行目が 3 桁になるほかはすべて 2 であ

置に対して線形に増加する 2 本のベクトルの内積によって作られる有理数の桁数が，ベクトル長にほぼ比例して増加するからである．

3.2 直交化における桁数削減

数値線形代数では，直交化を用いるアルゴリズムが使用されることが多いが，有理数計算では直交化を用いると桁数は非常に多くなる．CG 法は，有理数計算では，桁数の増大が膨大になる．このため，ベクトル要素から共通因子を抽出して，これによってベクトルをスケーリングし，桁数の少ないベクトル要素からなるベクトル演算に切り替えることが，高速化に効果がある．この操作は，**有理数の四則演算のたびに，分母・分子の GCD を求めて既約分数にする操作のベクトル版**と言える．rblas にはこのルーチンを用意する．本節では，CG 法とシュミットの直交化アルゴリズムにおける，スケーリングの効果を紹介する．

3.2.1 CG 法

浮動小数点演算を基本とする数値計算の分野では，CG 法は「反復解法」に分類される[*10]．しかし，筆算や有理数計算では内積は正確に計算することができるので，正確な直交化を計算できる．

> 直交するベクトルは 2 次元空間には 2 本，n 次元空間には n 本しか存在しない．CG 法は，反復のたびに解ベクトルの存在する空間の次元を少なくしてゆくので，n 次元の問題は，n 回以内の反復で解ける．

この単純な原理に基づく解法が，浮動小数点計算では内積を正確に計算できないために実現できなかった．有理数計算では，正確な直交化ができるので，CG 法は「直接解法」である．

$\boldsymbol{p}_i^T A\boldsymbol{p}_j = 0$ のとき「$\boldsymbol{p}_i$ と $\boldsymbol{p}_j$ は A 直交である」あるいは「（互いに）共役である」という．CG 法が用いる直交性は $\boldsymbol{p}_i^T A\boldsymbol{p}_{i-1} = 0$（あるいは同値の $\boldsymbol{r}_i^T \boldsymbol{r}_{i-1} = 0$）である．2 組の 2 項漸化式を用いる（探索方向ベクトル $\boldsymbol{p}$ を用いて残差ベクトル $\boldsymbol{r}$ を更新，$\boldsymbol{r}$ を用いて $\boldsymbol{p}$ を更新する）ことで，$\boldsymbol{p}_i$ と $A\boldsymbol{p}_{i-1}$ が直交するように組み立てられる．CG 法のアルゴリズムを示す [9]． 有理数（正確な）計算では，収束判定条件は $\boldsymbol{r}_k \neq 0$ で記す．

$k = 0;\ \boldsymbol{x}_0 = 0;\ \boldsymbol{r}_0 = \boldsymbol{b}$
while($\boldsymbol{r}_k \neq 0$)
　$k = k + 1$
　if($k = 1$)
　　$\boldsymbol{p}_1 = \boldsymbol{r}_0$
　else
　　$\beta_k = \dfrac{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}{\boldsymbol{r}_{k-2}^T \boldsymbol{r}_{k-2}}$
　　$\boldsymbol{p}_k = \boldsymbol{r}_{k-1} + \beta_k \boldsymbol{p}_{k-1}$
　end
　$\alpha_k = \dfrac{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}{\boldsymbol{p}_k^T A \boldsymbol{p}_k}$
　$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$
　$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k A\boldsymbol{p}_k$
end
$\boldsymbol{x} = \boldsymbol{x}_k$

この計算を筆算で，すべての数値を丸めずに分数で行えば，n 回以下の反復で残差が零となって収束することが確認できる．**有理数計算は分数を用いた筆算の延長**なので，筆算同様に収束する．一方，浮動小数点計算では，有限桁の有効数字に丸めながら計算するので正確な計算はできない．そのため，前処理に関する数学知識と，判定の閾値を調整するなどの計算機に対する知識を駆使して使用している[*11]．

CG 法を LDL 分解と比較すると，この規模の問題でも，CG 法で扱う有理数の分母・分子に現れる整数の桁数は非常に大きくなるため，LDL 分解のほうが桁違いに速い．ガウスの消去法の k 回目の基本操作は，係数行列の第 k 列の，対角項よりも下の要素を消去する変換を行っているが，この変換では行列式の値が保存される（シンプレクティックな変換）．行列式は座標軸方向の単位ベクトルを使って求められる．桁数が初期状態（a_{1j} の桁数）から線形に増加するベクトル同士の内積によって，三角行列の各項が求められた．

CG 法では，与えられた右辺ベクトルが最初の探索方向ベクトル $\boldsymbol{p}$ になり， $A\boldsymbol{p}$ 方向と直交する空間を計算する．これにより，解の存在する空間の次元数が減る．反復を LDL 分解の基本操作と比較すると，各基本操作で問題の次元数が減るところは同じであるが，その空間の基底ベクトルの方向は，単位ベクトル方向ではない．この方向を計算し，その空間で

る．ヒルベルト行列の要素を浮動小数点数に丸めると，対角項の桁数の和は 2,4,10,13,16,19,21,23,25,25 と推移する．

[*10] CG 法は 1952 年に Hestenes と Stiefel によって発表されたが，計算誤差の問題から実用的な方法としては用いられなかった．これらの手法が再認識されたのは 1970 年代後半で，前処理技術の発展によって本格的な実用期を迎え，その後数値線形代数の重要な研究テーマとなって現在も新しい手法が開発されている．主たる研究テーマは，非対称行列用の前処理技術と並列化技術にある．

[*11] 数値計算の現場では，大規模疎行列を係数行列とする連立 1 次方程式の解法には，標準的に CG 法や BiCG 法が使用されている．これらの方法の使用にあたっては，前処理技術を理解しなければならないので，その使用には，固有値の知識が必要である．

直交性を求めるので，桁数はLDL分解よりもはるかに多くなる．このため，反復ベクトル（残差ベクトルと探索方向ベクトル）は，反復のたびに桁数の多い α_k や β_k を掛けられて桁数を増加し続け，これらのベクトルの内積によって作られる残差ノルム，α_k，β_k の桁数は増加してゆく[*12]．

そこで，次の式で示すように，$\boldsymbol{r}_k$ と $\boldsymbol{p}_k$ が求まった段階で，ベクトルの要素の分母の最大公約数，分子の最大公約数を求めて，共通因数 s_p を外に出す必要がある[*13]．

$$\boldsymbol{p} = \begin{pmatrix} \frac{b_1}{a_1} \\ \frac{b_2}{a_2} \\ \vdots \\ \frac{b_n}{a_n} \end{pmatrix} = s_p \begin{pmatrix} \frac{d_1}{c_1} \\ \frac{d_2}{c_2} \\ \vdots \\ \frac{d_n}{c_n} \end{pmatrix} = s_p \boldsymbol{p}_s$$

$$s_p = \frac{\mathrm{GCD}(b_1, b_2, \cdots, b_n)}{\mathrm{GCD}(a_1, a_2, \cdots, a_n)} \tag{8}$$

これによって，$A\boldsymbol{p} = s_p A\boldsymbol{p}_s$ のように，行列ベクトル積の計算で，桁数の少ないベクトルによって計算することができる．

```
void ScalVec(rational_vector& a,int n,rational& s
           , rational_vector& b,int update) {
  int i;
  longint sd, sn, gd, gn;
  gd = a[0].denominator();
  gn = a[0].numerator();
  for (i=1; i < n; ++i) {
    sd = a[i].denominator();
    sn = a[i].numerator();
    if(gd!=longint::ZERO)gd=longint::lgcd2(gd,sd);
    if(gn!=longint::ZERO)gn=longint::lgcd2(gn,sn);
  }
  s = RRset(gn, gd);
  if(s != rational::ZERO){
    if(update == 1){
      for (i=0; i < n; ++i) { b[i] = a[i] / s; }
    }
  }
}
```

引数 update は，共通因子でベクトルをスケールするかしないかの指定をする．

十進BASICの有理数モードでは，有理数の分母を組込み関数 denom，分子を numer で取りだすことができる．整数は分母が1の有理数なので，これらと組込み関数 gcd によってScalVecをプログラミングできる[*14]．

```
EXTERNAL SUB scalvec(a(),n,s,b(),update)
LET gcdd=denom(a(1))
LET gcdn=numer(a(1))
FOR i=2 TO n
   LET sd = denom(a(i))
   LET sn = numer(a(i))
   LET gcdd=gcd(gcdd,sd)
   LET gcdn=gcd(gcdn,sn)
NEXT i
LET s=gcdn/gcdd
IF s <> 0 and update =1 THEN
   FOR i=1 TO n
      LET b(i)=a(i)/s
   NEXT i
END IF
END SUB
```

探索方向ベクトル $\boldsymbol{p}_k$ と残差ベクトル $\boldsymbol{r}_k$ を生成したら，それらの共通因子をScalVecルーチンによって抽出し，式(8)の $\boldsymbol{p}_s$ によって後続のベクトル計算を置き換えることで，

- 行列ベクトル積 $\boldsymbol{q} = A\boldsymbol{p}_k$，
- 内積 $\boldsymbol{q}^T\boldsymbol{p}_k$，
- 2つのaxpy計算 $\boldsymbol{x}_{k-1} + \alpha_k\boldsymbol{p}_k$ と $\boldsymbol{r}_{k-1} - \alpha_k\boldsymbol{q}$，
- 内積 $\boldsymbol{r}_k^T\boldsymbol{r}_k$

の計算の桁数を削減することができる．これによって計算時間は桁違いに削減される．$n = 20$ や 30 の乗算合同法による擬似乱数による係数行列で解くと，スケール化した反復ベクトルを使用することで，計算速度はほぼ10倍速くなった．

この改良によっても，CG法の内積を正確に計算することは膨大な演算量を必要とするので，有理数計算で連立1次方程式を解く場合は，CG法は直接解法よりも桁違いに計算時間がかかるアルゴリズムとなり，CG法は連立1次方程式の解法としては意味がない[2]．しかし，対称行列の正確な3重対角化を実現するにはCG法は必要である[11]．

3.2.2 シュミットの直交化

シュミットの直交化アルゴリズムを**浮動小数点演算**によってプログラミングする場合，古典的な計算順序による Classical Gram-Schmidt（CGS）法，計算誤差を少なく抑える目的から計算順序を変更した

[*12] LDL分解が初期行列の桁数から線形に増加する桁数のベクトルの内積で計算できたのに対し，CG法で求める内積は，反復ごとに桁数を増加させるベクトルを扱うので，桁数が $O(n^2)$ あるいはそれ以上になる．$n = 10$ の乱数による行列では，LDL分解が10進で330桁で解けるのに対し，CG法では6000桁を必要とする．

[*13] s_p は $\boldsymbol{p}$ ベクトルのスケール因子の意．また $\boldsymbol{p}_s$ は $\boldsymbol{p}$ ベクトルをスケール因子 s_p で割ったベクトルの意．

[*14] 有理数計算を試すことのできる計算環境が少ないので，確認の必要な方のために十進BASICのコードを添えた．

Modified Gram-Schmidt（MGS）法，計算速度の観点から CGS を 2 回実行するダブル CGS 法 の 3 つが存在する [9]．有理数演算では丸め誤差が現れないので，CGS と MGS は結果に差異はなく，ダブル CGS の意味はない．

CGS では元の行列 A の列ベクトルに対して内積 $\boldsymbol{p}_k^T\boldsymbol{a}_j$ を計算するのに対し，MGS では `axpy` 更新された後のベクトルに対して内積 `dot` を計算する．この計算の違いが，内積をとる項の**桁数の差に表れるので，MGS は CGS に比べて 1.5 倍近い計算時間を要する**．浮動小数点計算で誤差を小さく抑える工夫は，誤差なし計算である有理数計算では，不要な処理であるだけでなく，桁数を増加させる逆効果になる．したがって有理数計算ではシュミットの直交化に MGS を使ってはならない．

シュミットの直交化では，直交行列の列ベクトルごとに共通因子を ScalVec 関数によって括り出すことで，内積計算の桁数を少なくすることができる．このチューニングにより，`ScalVec` での GCD 計算の少なからぬ時間を相殺して，乗算の桁数削減効果で，約 55% の計算時間に改善された．十進 BASIC で乱数を RND とするとメルセンヌツイスターを使用するが，これによって生成した一様乱数から 0.5 を引いて作成した 100×50 の行列で線形最小 2 乗解を求めた．次表の数値は，表 1 の計測と同じパソコンによる計算時間である．

algorithm	スケール CGS	CGS	MGS
計算時間（秒）	581	1068	1991

シュミットの直交化による方法では，共通因子を括りだしてチューニングした版でも，正規方程式を素直に解く方法より 6 倍以上の計算時間を必要とする．浮動小数点演算による数値計算法の常識は，丸め誤差を制御するための工夫から生まれたものが多いので，丸め誤差なしの有理数計算では通用しないものがある．

4. おわりに

数値計算は永年，浮動小数点計算を前提に解法を改良してきた．この改良の多くは，精度と計算速度に焦点があてられてきた．したがって，浮動小数点計算では，精度と計算時間の両方を意識したプログラミングをしなければならないが，有理数計算は正確な計算が実現されるので，精度の問題は考えないでよい．計算時間については，計算機そのものに対する知識だけでなく，桁数を考慮することも重要である．これが有理数計算による数値計算の大きな特徴である．本報告では，数値線形代数の基本的なテーマ，連立 1 次方程式の解法に関する「直接解法」を，「直交化を用いる解法」と比較することで，浮動小数点計算とはアルゴリズムの選択が異なることを，直交化を基礎とする解法を有理数計算で実行する場合の桁数の推移を調べることから述べた．

有理数計算プログラミング環境は，桁数の削減で 1 桁，スレッド並列化で（16 ウェイの PC で）1 桁の高速化が達成されたケースもある．今後，分散メモリ型並列も実装することで，大規模な並列計算機上では，1 スレッドのパソコンの数 100 倍の高速化も期待される．初代スパコンと称される CRAY-1 と比べると，現在の最高速のスパコンは 1 億倍以上速い．HPC 技術のこのようなスケールは，現在は数学教育用に使用されている十進 BASIC の有理数モードの計算を，第 1 章に述べた目的での使用に拡大できる日が近づきつつあることを示唆している．

参考文献

[1] 関川 浩, 数式処理と数値計算の融合, **情報処理**, 2009, April.
[2] 寒川 光, 有理数計算による実対称行列の正確な 3 重対角化による固有値の高精度計算, *HPCS2013* **論文集**, 2013.
[3] *http://hp.vector.co.jp/authors/VA008683/*.
[4] 飯高 茂， 松本幸夫（他）, **高等学校–数学** *B*, 東京書籍, 2008.
[5] D. Knuth., *The Art of Computer Programming, Volume 2, Third Edition*, Addison-Wesley, 1998, 有澤　誠，和田英一監訳：The Art of Computer Programming, Third Edition , 株式会社アスキー, 2004.
[6] J.J. Dongarra, F. G. Gustavson, and A. Karp, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Review*, Vol. 26, No. 1, pp. 36–50, 1986.
[7] J.J. Dongarra, J. Du Croz, S. Hammarling, and J. Hanson, An extended set of fortran basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, pp. 1–17, 1988.
[8] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 1–17, 1990.
[9] 寒川 光, 藤野清次, 長嶋利夫, 高橋大介, *HPC* **プログラミング** *—ITText* **シリーズ**, オーム社, 2009.
[10] G. H. Golub and C. F. Van Loan, *Matrix Computations — third edition*, Johns Hopkins University Press, 1996.
[11] Hikaru Samukawa, Rational number arithmetic including square root handling, **日本シミュレーション学会論文誌**, 4(4):pp. 181–189, 2013.