Vol.2017-HPC-158 No.31
2017/3/10

有理数計算プログラミング環境での有理数BLASの並列化について

寒川　光[1,a)]

概要：「有理数計算プログラミング環境」は，分母・分子を可変長の多桁数で保持する有理数を，C++の演算子多重定義を用いて実装することで，四則演算を +，−，∗，/ の記号で書くことができる．これにより，有理算術演算による数値計算を，浮動小数点演算のようにプログラミングできるが，計算の高速化が課題である．数値線形代数（Nummerical Linear Algebra）のサブルーチンを，Fortran 90 の部分配列に似た表現で，行列やベクトルの一部を外部関数に渡して，BLAS ライブラリ的な並列化で設計した．本稿ではその報告と有理算術演算の特長的な数値線形代数の演算である，完全軸選択で行列の階数を特定できる LU 分解のスレッド並列化を計測し，その結果を紹介する．

キーワード：有理数計算，スレッド並列化，BLAS，部分行列，完全軸選択，階数，連立 1 次方程式，同次方程式，非自明解

A Parallelizaion of Rational Arithmetic Programming Environment

Hikaru Samukawa[1,a)]

Abstract: We are developing a rational arithmetic programming environment. The numerator and the denominator of the rational number are kept in variable length array, and the rational arithmetic is written by using symbols +, −, ∗, / by using C++ operator overload. Though the users of our programming environment can program numerical methods code in the rational arithmetic similar to the floating-point arithmetic, performance issue is still annoying. We have developed a parallel version of rational BLAS by using submatrix expression resembles to Fortran 90 submatrix in BLAS style. In this paper, we report its implementation and thread parallel performance of numerical linear programs, the LU decomposition with complete pivoting for the purpose to reveal rank of the matrix.

Keywords: rational arithmetic, thread parallelization, BLAS, submatrices, linear equation, complete pivoting, rank, homogeneous equation, non-trivial solution

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

[1] 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University, Shinjuku-ku, Tokyo 169–8555, Japan
[a)] samukawa@sic.shibaura-it.ac.jp

低い．数値計算と対照的に，有理算術演算やモジュラー算術演算を用いる数式処理（計算機代数）は，信頼性は高いが柔軟性に欠け，計算効率も低い．数値計算に有理算術演算を含めると，両者の利点を備えたプログラミング環境ができそうに思えるが，計算効率の問題は残る．

汎用プログラミング言語で有理算術演算をサポートする身近な言語に十進 BASIC がある．十進 BASIC は，文教大学の白石教授が開発したプログラミング言語である [1]．数学教育を目的に開発され，数学者の中には「日本の宝」と高く評価される先生もおられる [2], p. 24．十進 BASIC は 5 つの計算モードを持つ．十進 15 桁，十進 1000 桁，2 進（Intel x86 倍精度浮動小数点），2 進による複素数，有理数である．「有理数モード」で計算可能な有理数の分母・分子の桁数は 2000 桁程度と推定される．BASIC の言語規格により，数値は単一の表現に統一されるので，浮動小数点数と有理数を同時に使用することはできない．

我々は，浮動小数点演算を用いる数値計算のプログラミング環境に，有理数型の変数を加えて，有理算術演算と浮動小数点演算の両者を使用できる「有理数計算プログラミング環境」を開発し，継続して機能を拡張している．有理数の分母・分子を約 30 万桁まで拡張可能な多桁の自然数（以下，多桁数）で保持する `rational` 型の変数を C++ のプログラミング環境に追加することで，有理数型の変数を浮動小数点型の変数と同じように記述できる．

この環境は次の使用を目的としている [3]．

- 丸め誤差の理論を学ぶ前の学生を対象とした，プログラミングを含めた数学教育
- 数値シミュレーションで用いられる，浮動小数点演算による数値計算プログラムで発生した精度に関係する問題の分析ツール
- ベンチマークの検収など，計算機システムの稼働確認

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）．高速化には，有理算術演算に適したアルゴリズムの選択，有理数を構成する分母と分子の桁数や基数などの構成，並列化の 3 者が鍵である．本稿では，数値線形代数プログラムのスレッド並列化について述べる．次章でプログラミング環境の概略を紹介する．3 章で部分行列表現，4 章で並列化の実装について述べる．5 章で数値実験の結果を示す．6 章でまとめと今後の課題について述べる．

なお，有理算術演算による数値計算は，浮動小数点演算による数値計算とは，（計算の進行とともに桁数が増加するため）計算時間の振舞いが大きく異なる．有理算術演算による数値計算の経験のない読者のために，随所に浮動小数点演算による数値計算との違いを解説した．並列化とは直接関係しない解説もあるが，誤解を避ける目的で入れた．

2. 有理数計算とそのプログラミング環境

有理算術演算は計算機科学の黎明期からの研究テーマである [4]．開発中のプログラミング環境では，有理数を，分母と分子を多桁数で保持し，これに符号を付加することで表現する．四則演算は次のように行う．

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

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

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

ここに r_1, r_2 は有理数，a, b, c, d は多桁数である．

2.1 「プログラミング環境」の概要

n 桁と n 桁の数の積は $2n$ 桁になるので，式 (1) の ac，bc，ad などの数の桁数は演算のたびに倍になる．そこで演算結果は，分母・分子の最大公約数（GCD）で割り既約な分数とする[*1]．「有理数計算プログラミング環境」は，上記の a, b, c, d などの多桁の数を **longint クラス**で，変数型 longint で扱う．多桁数は，基数を $r = 2^{32}$ として 32 ビット符号なし整数の配列に格納し，桁数 l を整数型で保持する．

$$z = \sum_{i=0}^{l-1} d_i r^i \tag{4}$$

ここに d_i は各桁の数で $0 \le d_i < 2^{31}$ である．零は桁数が零（$l = 0$）とする．**配列長は 64 から始めて，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 進数で約 31 万桁）[*2]．`longint` クラスは，式 (1)(2)(3) の左辺の演算の本体である四則演算の関数や，GCD 関数をもつ．

有理数を扱う **rational クラス**をその上に構築し

[*1] 有理数の逆数を求める場合や，正規連分数の計算など，既約な有理数になることが保証されるアルゴリズムではこの操作を省略したいので，ユーザプログラムから GCD 計算の省略を選択可能とした [5]．

[*2] $\log_{10} 4294967296 = 9.633\cdots$ である．計算可能な最大の数は，再コンパイルで変更可能である．

た．有理数 r を多桁数 n と d と符号 s で保持する．

$$r = s \times \frac{n}{d} \tag{5}$$

$s = 0$ の場合 $r = 0$ である．rational クラスでは，式 (1)(2)(3) の有理算術演算を記号 $+$, $-$, $*$, / でプログラミングするための演算子多重定義のほか，`min`，`max`，`floor`，`ceil` などの関数や，浮動小数点数を有理数に変換する関数などをもつ．

区間算術演算（interval arithmetic）を扱う **interval クラス**をその上に構築した．有理算術演算では無理数を正確に計算することはできない[*3]．非線形方程式の求解のように，求解の時点では解が有理数か無理数かが分からないことは多い．このような問題のために区間算術演算を用意した．2 つの有理数で区間の下限と上限を抑える `interval` 型の変数を設けた．`interval` 型の変数同士，また `interval` 型と `rational` 型の変数の算術演算は，記号 $+$, $-$, $*$ でプログラミングできる．区間 x からその下限を rational 型の変数に取り出すには `lower`，上限を取り出すには `upper` 関数を使用する．区間 $x_k = (a_k, b_k)$ の符号を反転すると $(-b_k, -a_k)$ になるが，これには `neg` 関数を用いる．区間 $x = (a, b)$ の逆数は `inv` 関数を用いる．`interval` 型の変数同士の除算は，逆数を掛けることで得られる．

数値線形代数（NLA: numerical linear algebra）を扱う **rblas クラス**をその上に構築した [7]．rblas は有理数（rational）BLAS（Basic Linear Algebra Subprograms）の意で，並列版を使用する場合には **prblas**（parallel rblas）クラスを使用する．

2.2 有理算術演算の用途

浮動小数点数は，有限のビット数で表現される数なので，数学的には有理数である[*4]．数値計算アルゴリズムで，四則演算だけで定式化されるものは，有理算術演算では正確に計算できる（丸め誤差が発生しない）．したがって，浮動小数点数を有理数に変換した数を係数行列と右辺にもつ連立 1 次方程式を，LU 分解のような四則演算だけで定式化されるアルゴリズムで解く場合は，正確な解が得られる．行列が特異なら階数も判定でき，非自明解も正確に得られる．

有理数係数の非線形方程式の解は，一般的には無理数なので，区間算術演算を用いて，解の存在する区間を得て，必要な精度が得られるまで区間を縮小反復する．有理数と浮動小数点数を混在させられるプログラミング環境の利点の例として，浮動小数点演算で得られた対称行列を有理数に変換した行列の全固有値を求める問題を紹介する [8][9]．有理算術演算では，基本変換を相似変換の形で適用して，正確な特性多項式を求めることができる．有理数係数の特性方程式の解（固有値）は，浮動小数点演算で求めた近似固有値を含む区間に対し，区間算術演算によって縮小反復すれば，必要な精度まで求められる．高精度の近似固有値を「根と係数の関係式」に代入すれば，特性多項式の因数分解もできる[*5]．浮動小数点演算で得られた近似解を，誤差を含む測定データと見れば，これは一種の逆問題である．

本稿では，LU 分解を NLA の例として計測し，並列化の効果を報告する．

3. 部分行列の表現と使用法

「有理数計算プログラミング環境」では，行列の形状はすべて長方形とした（疎行列や三角行列も零要素も含めて扱う）[*6]．長方形行列から，部分行列，行ベクトル，列ベクトルなどを定義することで，NLA の多くのアルゴリズムの記述が簡潔になる．本章では NLA で定着している BLAS インターフェースを踏襲するための，部分行列の定義について述べる．

3.1 BLAS-1 と-2 のインターフェース

ガウス消去法のアルゴリズムは，軸選択を省略して素朴に Fortran で表すと次のように書ける [10], p. 14.

```
dimension a(n,n)
do k=1,n-1
```

[*3] 例外的に平方根は，後続の計算でその 2 乗が使用される場合に，平方根の 2 乗を保持する「陰的定式化」によって正確な計算を実現できる場合がある [3][6].

[*4] 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．

$$a = \left(\pm \sum_{i=0}^{52} d_i 2^{-i} \right) \cdot 2^e = A \cdot 2^e \tag{6}$$

$B = A \cdot 2^{52}$ 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 (6) 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．分母は 2 のべき乗である．

[*5] 十進 BASIC の有理数モードで正確な特性多項式を求めることはできるが，近似固有値を精度改良するのは困難である．ここに「有理数計算プログラミング環境」の有意点がある．

[*6] 行列の 1 つの要素が占めるメモリ容量は可変長なので，零要素をデータ構造によって排除しても，メモリや計算時間を節約する利得は小さい．

```
      do i=k+1,n
        a(i,k)=a(i,k)/a(k,k)
      enddo
      do j=k+1,n
        do i=k+1,n
          a(i,j)=a(i,j)-a(i,k)*a(k,j)
        enddo
      enddo
    enddo
```

BLAS は Fortran の列優先（column major）の順序を規定する Sequence Association（SA）を前提に作られた．引数のアドレス渡し（call by address）を前提として，「配列の要素アドレス，先導次元，ストライド」の三つ子組を渡すことで，多くの NLA のアルゴリズムの最内側ループが BLAS-1 で記述でき，内側の 2 重ループが BLAS-2 で記述できる．上の例は，倍精度なら次のように記述できる．

```
double precision a(n,n),x(n),y(n)
do k=1,n-1
call dscal(1.d0/a(k,k),a(k+1,k),1)
call dger(-1.d0,a(k+1,k),1,a(k,k+1),n,a(k+1,k+1),n)
enddo
```

dscal は倍精度で $\boldsymbol{x} \leftarrow \alpha\boldsymbol{x}$ を計算し，dger は $A \leftarrow A + \alpha\boldsymbol{x}\boldsymbol{y}^T$ を計算する[*7]．これを受ける側を dger を例に示す．

```
subroutine dger(alpha,x,ix,y,iy,a,lda)
double precision alpha,x(*),y(*),a(lda,*)
do j=1,n
 do i=1,n
  a(i,j)=a(i,j)+alpha*x(i+(i-1)*ix)*y(j+(j-1)*iy)
 enddo
enddo
return
```

引数の 2 次元配列の要素（アドレス）を 1 次元配列（ベクトル）として受取ることは，「数学的には行列の一部がベクトルを形成するので当然」と思われるが，C や C++や BASIC などで同じようにプログラミングすることは，必ずしもやさしくない．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 が確定していなくても，上記のコードは稼働する．しかし 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複雑になる[*8]．

「有理数計算プログラミング環境」は，十進 BASIC の有理数モードとの計算結果を比較しながら開発した時期があった．このため，同じ定式化で同じループ構成を使用する目的から，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 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た [7]．しかしこのような調整を行うと，中間ルーチンを介するなどの調整が必要になり，関数のメニューが増大する弊害を生んだ．この不都合を解消するために，Fortran 90 の部分配列に倣って，部分配列を定義できるようにした．

3.2 Fortran 90 的なインターフェース

軸列を対角項 $a_{k,k}$ で割る操作は $\boldsymbol{x} \leftarrow \alpha\boldsymbol{x}$ を rscal$(\alpha, \boldsymbol{x})$ によって行う．

```
rat::lmatrix_column<rational> c1(a,k-1,k,n-1,1);
rat::rscal(rational::ONE/a[k-1][k-1],c1);
```

行列の「第 k 列目の列ベクトルの，$k+1$ から n 番目でストライド 1 の部分ベクトル」は，C では添字（インデックス）ではなくオフセットで配列要素を指すので，Fortran の k は C++では k-1 と 1 つ小さくなり，a.column(k-1,k,n-1,1) で取得する．書込む場合は，rational 型の部分列ベクトルを lmatrix_column で定義してここに書込む．

階数 1 更新 $A \leftarrow A + \alpha\boldsymbol{x}\boldsymbol{y}^T$ で右下の小行列を更新するところは，更新された軸列のベクトルが $\boldsymbol{x}$ で a.column(k-1,k,n-1,1) で取得でき，第 k 行の行ベクトル $\boldsymbol{y}^T$ が a.row(k-1,k,n-1,1) で取得でき，右下の小行列を sm で定義すれば，階数 1 更新 rger$(\alpha, \boldsymbol{x}, \boldsymbol{y}, A)$ で記述できる．

```
rat::lmatrix_section<rational> sm(a,k,n-1,1,
                                  k,n-1,1);
rat::rger(-rational::ONE,a.column(k-1,k,n-1,1)
                        ,a.row(k-1,k,n-1,1),sm);
```

4. スレッド並列化

浮動小数点演算と比較すると，桁数の多い有理算術演算の計算時間は桁違いに長い．有理算術演算で行列演算を行うと，有理数を構成する分母と分子の

[*7] d は double，scal は scale，ge は general matrix，r は rank-1 update からとられた．

[*8] C の 1999 年の言語標準では，可変長配列（variable length array）が導入され，初期サイズを実行時に指定できるようになった．だが，この規格は，次の 2011 年の言語標準ではオプションに格下げされた．したがって C では，配列にポインタを格納してこれを渡すか，プリプロセッサによる書換えが専ら使用される．

桁数増大のために，問題の次数 n に対する計算量のオーダーが浮動小数点演算の場合よりも高くなり，計算時間は急激に増大する．このため高速化の対象を，有理算術演算による行列演算に集中した．HPC アプリケーションの並列化では OpenMP を用いて反復計算をスレッド並列化するが，BLAS ライブラリの並列化は，下位の階層で行い，プリプロセッサやコンパイラの実装に影響されないようにする（例えばメーカから提供される BLAS ライブラリは，ユーザプログラムの OpenMP による並列化以前に並列化された版が使用できる）．「有理数計算プログラミング環境」でも，並列版 rblas はライブラリとして扱いたいので，テンプレート関数を用いて実装した．「有理数計算プログラミング環境」のユーザは，ライブラリ化された数値線形代数（Numerical Linear Algebra, NLA）のメニューを選び，計算機に適したスレッド数を Makefile の変数 NUM_THREADS にセットして make するだけで並列化できる．それ以外のユーザ自身のコードの並列化は，あまり考えないでよく，数学教育用にも（十進 BASIC の延長上で）利用できることを想定した．

4.1 並列化の概要

有理算術演算では，コンパイラは，浮動小数点演算とは異なり，ブロック化のようなループ変換を伴う最適化を行わない．記号 $+$, $-$, $*$, / で四則演算が書かれていても，変数が有理数の場合は，rational クラスの有理数に対する四則演算ルーチンと GCD ルーチンを呼出す（x86 や EM64T では call 命令）だけである．しかも演算は dot や axpy のように単純である．このような場合，ジェネリックなプログラミングで簡潔に表現し，型を抽象化できる．STL（Standard Template Library）は，「変数型，コンテナ，演算（アルゴリズム）」の 3 点セットを念頭に設計された．要素型（int，double，rational）とコンテナ（ベクトル，部分ベクトル，小行列）に対して，単純な演算（dot や axpy のような反復計算）を行うコードを生成する．rtraits.h に，途中で使用する変数型を取得できるように定義しておき，これとテンプレート関数を用いてコード生成した．コードはすべてヘッダーファイル prblas.h に収めた．

4.2 prblas.h の実装

POSIX のスレッドで，複数のスレッド間で同期をとる操作として提供されているセマフォを用いた．thread_manager.cpp を作成し，この中の execute 関数から，POSIX インターフェースの sem_post，sem_wait などを呼出す．prdot，praxpy，prscal，prger など（rblas の並列版の） prblas の関数は，引数（反復セット）や関数名をコンテナ（task_t 構造体）に作成する．これを NUM_THREADS 個作成して execute 関数を介して実行する．つまり，execute 関数が各スレッドタスクを間接的に実行することで並列実行する．このコードをテンプレート関数を用いて生成するプログラムを prblas.h に書いた．

praxpy を例にする．構造体 axpy_param をテンプレート関数で定義し，引数の型を格納できるようにする．並列タスク本体の反復演算を partial_axpy として作るが，axpy_param は NUM_THREADS 個生成する．演算は，自分に与えられた反復セット begin，end，stride について，“$y_i = \alpha \cdot x_i + y_i$” を，変数型 T に対して実行するコードとして生成する．

```
template<typename T, template<typename> class T1
                   , template<typename> class T2>
class axpy_task {
public:
   static axpy_param<T,T1,T2>
       axpy_params[thread_manager::num_threads];
   static thread_manager::task_t tasks[
                    thread_manager::num_threads];
   static void partial_axpy(void* t) {
      axpy_param<T,T1,T2>* param = static_cast
                      <axpy_param<T,T1,T2>*>(t);
      for (int i=param->is.begin_; i < param->
              is.end_; i += param->is.stride_) {
         param->y_->operator [](i) += *param->
             alpha_*param->x_->operator [](i); }
   }
};
```

T には rtraits.h で定義された変数型が入る．

反復セットは，指定によりブロック分割が選べ，指定がない場合はサイクリック分割が作られる．サイクリック分割を優先した理由は，有理算術演算では，ベクトル要素である有理数の桁数が不揃いである場合に，ブロック分割では負荷バランスが崩れる可能性が高いと考えたからである．

ベクトル長を複数のスレッドで並列計算するので，raxpy の引数に，そのスレッドが担当するベクトル要素の開始要素番号 begin と終了要素番号 end，結果を置く result が追加される．この praxpy_param_t 型の構造体を，スレッド数，配列によって定義して，thread_manager::execute によって実行する．

5. 計算例

連立 1 次方程式を解く問題（LU 分解と代入）を計測した．有理算術演算では，浮動小数点演算とは異なり，行列の階数を正確に知ることができる．浮動小数点演算による行列の階数判定（rank reveal）アルゴリズムが，数値計算の高度な知識を必要とするのに対し，有理算術演算の階数判定は，数学の教科書に書いてあるとおりにプログラミングすればよく，簡単である．非自明解も，数学の教科書の通りにプログラミングすればよい．有理算術演算は，数学の授業でプログラミングを導入して知識の定着に利用する目的には好都合である．

5.1 連立 1 次方程式の解法プログラム

一般（非対称）行列を係数行列とする連立 1 次方程式の解法には，浮動小数点演算では，部分軸選択（partial pivoting）を使用した LU 分解と，それに対応する前進および後進代入が用いられる．正確な計算がサポートされる有理算術演算では，（計算誤差が入らないので）部分軸選択では，独立した 2 つの小行列に分離することにより，正確に階数を得られない場合がある．この問題を回避するために，完全軸選択（complete pivoting）で LU 分解を行う．

5.1.1 LU 分解

$n \times n$ の有理行列を与えると，その領域に完全軸選択で得られた三角行列を返す．引数 ipvt と jpvt には軸選択情報が，最後の引数 rank には階数が返る．

```
void LUdecomp(rational_matrix& a,
           const int n,vector<int>& ipvt,
           vector<int>& jpvt,int& rank)
```

アルゴリズムは，各段の消去で，軸列の軸要素 $a_{k,k}$ が零の場合，右下の小行列で最初に見つかる非零要素 $a_{k+ip,k+jp}$ と交換（k 行と $k+ip$ 行，k 列と $k+jp$ 列を交換）する[*9]．交換は rswap 関数による．次に，軸列を対角項 $a_{k,k}$ で割る．これは $\boldsymbol{x} \leftarrow \alpha\boldsymbol{x}$ を $\mathtt{rscal}(\alpha, \boldsymbol{x})$ によって行う[*10]．

```
rat::lmatrix_column<rational>
                        col1(a,k-1,k,n-1,1);
rat::rscal(rational::ONE/a[k-1][k-1],col1);
```

行列 A の第 k 列目の列ベクトルの，$k+1$ から n 番目の要素からなるベクトルは，a.column(k-1,k,n-1,1) で取得できるが，rscal では書き込むので，（メモリーリークを避けるため）rational 型の部分列ベクトル col1 を定義してここに書き込む．ベクトルの指定ではストライドも指定できるが，ここではストライド 1 である．

次に，階数 1 更新 $A \leftarrow A + \alpha\boldsymbol{x}\boldsymbol{y}^T$ で右下の小行列を更新する[*11]．更新された軸列のベクトルが $\boldsymbol{x}$ で a.column(k-1,k,n-1,1) で取得でき，第 k 行の行ベクトル a.row(k-1,k,n-1,1) が $\boldsymbol{y}^T$ になり，右下の小行列が A で階数 1 更新 $\mathtt{rger}(\alpha, \boldsymbol{x}, \boldsymbol{y}, A)$ される．

```
rat::lmatrix_section<rational> submat(a,k,n-1,
                                    1,k,n-1,1);
rat::rger(-rational::ONE, a.column(k-1,k,n-1,
                1), a.row(k-1,k,n-1,1),submat);
```

行列 A の第 $k+1$ 行から n 行，第 $k+1$ から n 列の小行列は，a.section(k,n-1,1,k,n-1,1,) で取得できるが，rger では書き込むので，submat を定義してここに書込む．

5.1.2 代入ルーチン

階数が次数に等しい場合は，LUsubst ルーチンで，右辺ベクトルを x に，LU 分解の出力である三角行列と行交換情報 ipvt と列交換情報 jpvt とを与えれば，右辺ベクトルを与えた引数に解ベクトルが上書きされて返される．

```
void LUsubst(const rational_matrix& a,
             const int n,
             rational_vector& x,
             const vector<int>& ipvt,
             const vector<int>& jpvt)
```

完全軸選択に伴う代入アルゴリズムは，行交換に従う要素交換を行ってから，前進代入と後進代入を行い，列交換に従う要素交換を行う．行または列要素を交換するための基本変換行列の積を P で表すと，行交換に関しては $PA\boldsymbol{x} = P\boldsymbol{b}$ を解けばよいので，最初に $P\boldsymbol{b}$ と交換してから，行交換された PA で代入計算を行う．列交換に関しては AP で代入するので $(AP)(P^T\boldsymbol{x}) = \boldsymbol{b}$ を解けばよく，代入完了後に $\boldsymbol{x}$ ベクトルの要素を交換する．ベクトル要素の交換は rational::swap(x[i],x[jpvt[i]]) のように行っている．また，前進代入，後進代入ともに最内側ループの計算は raxpy で行っている．

[*9] 浮動小数点演算では，誤差を少なくする目的から，絶対値最大の要素を探すが，有理算術演算では，零割りさえ避ければよいので，最大を探す必要はない．
[*10] 倍精度浮動小数点計算で BLAS の dscal に対応する．
[*11] 倍精度浮動小数点計算で BLAS の dger に対応する．

5.1.3 同次連立 1 次方程式の非自明解

$A\boldsymbol{x} = \boldsymbol{0}$ は $\det(A) = 0$ の場合に $\boldsymbol{x} = \boldsymbol{0}$ 以外の非自明解（non trivial solution）を持つ．n よりも r だけ階数が低い $n \times n$ の行列 A の左上 $(n-r) \times (n-r)$ の小行列 A_{11} が正則とする．

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad (7)$$

上の式は $A_{11}\boldsymbol{x}_1 + A_{12}\boldsymbol{x}_2 = 0$ であり，A_{11} は正則なので，$\boldsymbol{x}_1$ と $\boldsymbol{x}_2$ の間には

$$\boldsymbol{x}_1 = -A_{11}^{-1}A_{12}\boldsymbol{x}_2 \qquad (8)$$

の従属関係がある．ここで $\boldsymbol{x}_2$ を任意の r 要素からなるベクトルとしたとき，ベクトル $\begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix}$ は $A\boldsymbol{x} = 0$ を満たす非ゼロのベクトルである．

階数が 1 だけ低い場合を考える．A の左上 $(n-1) \times (n-1)$ の行列 A_{11} が正則なら，A_{12} は $(n-1) \times 1$ で $n-1$ 要素のベクトルであり，$\boldsymbol{x}_2$ はスカラーになる．$x_2 = 1$ にすると，同次連立 1 次方程式の解は次のように表わされる．

$$\boldsymbol{x} = \begin{pmatrix} -A_{11}^{-1}A_{12} \\ 1 \end{pmatrix} \qquad (9)$$

関数 solhmg によって，同次方程式（homogeneous equation）の非自明解を得られる．

```
void solhmg(rational_matrix& a, const int n,
            int& rank, rational_matrix& x)
```

アルゴリズムは，式 (8) の小行列 A_{12} の要素を探すために，列交換の情報を使用して右辺ベクトルを作成し，これの 1 から rank までの範囲で行交換の情報で要素交換を行ってから LUsubst で代入計算を行う（LUsubst では要素交換をしないので，ダミーの軸選択の引数を与える）．LUsubst 終了後に列交換の情報で要素交換を行う．

```
for (int i=0; i < n; ++i) kk[i]=i;
for (int i=rank-1; i >= 0; --i){
  if(jpvt[i]!=i) std::swap(kk[i],kk[jpvt[i]]);
}
for (int k=rank-1; k <= n-2; ++k){
  int kkk=k+1;
  for(int i=0;i<=rank-1;++i){ // search column
    if(kk[i] == k+1) kkk=i;
  }
  for(int i=0;i<n;++i) y[i]=-c[i][kkk]; // RHS
    for (int i=0; i <= rank-2; ++i){
      kpvt[i]=i;
      if(ipvt[i]!=i)
              rational::swap(y[i],y[ipvt[i]]);
    }
    kpvt[rank-1]=rank-1;
    for (int i=rank; i <= n-1; ++i) y[i]=0;
    y[k+1]=1;                    //set x_2=1
    LUsubst(a,rank,y,kpvt,kpvt);//substitution
    for (int i=rank-1; i >= 0; --i){
      if(jpvt[i]!=i)
              rational::swap(y[i],y[jpvt[i]]);
    }
  for(int i=0; i<n; ++i) x[i][k-rank+1]=y[i];
}
```

なお，行列が特異ではなかった場合には，仮の右辺ベクトルで代入計算を行い，これを返す．

以上，有理算術演算の特長を活かして，浮動小数点演算ではできない，階数を判定して，行列が特異の場合は，非自明解を計算するアルゴリズムとプログラムを紹介した．完全軸選択（complete pivoting）で LU 分解を行うことで階数を得られ，代入計算によって同次方程式の非自明解も正確に得られる．このプログラムによって，固有値問題では，固有値が有理数の場合は，その固有値に対応した固有ベクトルを正確に求めることができる．

5.2 計測結果

計測に使用した計算機は, Intel の Sandy Bridge マイクロアーキテクチャの Xeon CPU E5-2670 2.6GHz を 2 つ搭載する．CPU は 8 コア（16 スレッド）なので，並列度は 16 ウェイ，32 スレッドになる．メモリは DDR-3 を 32GB 搭載している．OS は Linux version 2.6.32，コンパイラは GNU gcc 4.4.7 である．

$n = 500$ で階数も n，行列要素の分子は乗算合同法 $x_k = 16807 \cdot x_{k-1} \bmod (2^{31} - 1)$ で，分母は $2^{31} - 1$ の行列を使用した．右辺は，解ベクトルの全要素が 1 になるように設定した．LU 分解と代入の計算時間（秒）と，逐次処理の LU 分解 1635 秒，代入 10 秒に対する倍率を示す．1 行目の数は Makefile に指定した変数 NUM_THREADS の値である．

	2	4	8	16	32	500
分解	822	434	225	108	119	115
倍率	1.99	3.77	7.33	15.1	13.8	14.2
代入	5.2	2.7	1.5	0.75	1.22	1.74
倍率	1.93	3.65	6.9	13.3	8.2	5.7

16 コアで NUM_THREADS を 16 に設定した場合が最も速い．有理算術演算のカーネルは整数演算主体なのでハイパースレッディングの効果は少ない．なお，反復セットはサイクリック分割で，ブロック分割はやや遅くなった．

6. まとめと今後の課題

既存の数値計算プログラミングの環境に有理算術演算を含めた「有理数計算プログラミング環境」に，スレッド並列化を実装し，数値線形代数のアルゴリズムを計測した．有理算術演算は CPU バウンドであるが，浮動小数点演算を使わないので，ハイパースレッディングの効果が薄いことなどの基本的性質が確認できた．16 ウェイの計算機で 8 倍から 15 倍の高速化が達成され，最初の成果は達成されたと考える．今後，いろいろな問題に対して並列化を試み，安定した計算効率を達成できる方法を探したい．また，多桁数の乗算に FFT アルゴリズムを含めての計測も行いたい．

参考文献

[1] *http://hp.vector.co.jp/authors/VA008683/*.
[2] 飯高 茂, **パソコンで開く数の不思議世界**, 岩波ジュニア新書, 2004.
[3] 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**, pp. 11–22, 2013.
[4] 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.
[5] 寒川 光. 有理算術演算における最大公約数計算について, **情報処理学会ハイパフォーマンスコンピューティング研究会資料**, (154), 2016.
[6] 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.
[7] 寒川 光, 有理数線形代数計算における有理数 BLAS の提案, *HPCS2014* **論文集**, pp. 57–64, 2014.
[8] 寒川 光, **講座　第 *3* 回「有理数計算」**, **シミュレーション**, 34(3):42–50, 2015.
[9] 寒川 光, **講座　第 *4* 回「有理数計算」**, **シミュレーション**, 34(3):46–55, 2015.
[10] 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.