有理数計算プログラミング環境
longint, rational, rblas, ratutil クラスと例題の解説

寒川　光，郷田　修

2017 年 4 月

目 次

本資料は，科学研究費補助金・基盤（C) 課題番号 25330145「有理数計算ライブラリの並列化と誤差診断ツールの開発」から支援を頂いて 2013 年度から 3 年間かけて開発した「有理数計算プログラミング環境」の解説書である．

研究開始当初の背景　有理数計算は計算機科学の古くからのテーマであるが，記憶域を多く使うことと，計算時間が膨大になるために実装が少ない．数学教育を目的とすると，文教大学の白石教授が作成されたプログラミング環境「十進 BASIC」が提供されており，この中で有理数モードを使用すると，数式に無理数が現れない範囲で丸め誤差の生じない正確な計算を実現することができる．十進 BASIC は，高校の『数学 B』の「数値計算とコンピュータ」章でも利用されていた．これを用いると，数学の授業で学んだ内容を，汎用プログラミング言語で確かめることができるので，数学教育とプログラミング教育の両方を学ぶことができる（浮動小数点演算では，丸め誤差が混入するため，数学公式や定理を確認するには，丸め誤差の影響を取り除いて考える必要があるので，大学初年度の学生には難しい）．扱う数式の範囲が，平方根や初等関数を含むものに拡がると，十進 BASIC の有理数モードは計算できない．これは，BASIC の言語仕様の問題で，数値はすべて単一の型で扱われるように定められているからである．有理数と誤差を含む浮動小数点数を混在してプログラミングすることが可能なプログラミング言語であれば，この壁を越えられる．たとえば Fortran や C に，変数型として有理数を加えられれば，誤差なしで計算する部分と，誤差を許して浮動小数点演算で近似計算する部分を使い分けるプログラミングが可能になる．なお，汎用プログラミング言語から外れて，数式処理ソフトウェアや数学アプリである MATLAB などを用いると，数学教育という目的は達成されるかもしれないが，汎用プログラニング言語の教育は別途行わなくてはならない．このような状況の中で，計算機の高速化と大容量化は進行しており，有理算術演算がより身近な時代が到来することも予測された．

研究の目的　「ある数値問題の答を「0.333333574」と表示する浮動小数点数ではなく，正確に 1/3 であると知ることが重要である場合は多い．算術演算を分数の近似値ではなく分数に対して行えば，丸め誤差をまったく蓄積することなしに多くの計算を実行できる.」これはクヌース先生の『The Art of Computer Programming，第 2 巻』の後半 Seminumerical Algorithms の第 4 章「算術演算」の 4.5 節「有理算術演算」の冒頭からの引用である．数値シミュレーションプログラムの開発，数値計算プログラムの開発などを手掛けていると，丸め誤差の影響で，正しい答えが分からず，問題追及に長い時間を要することが少なくない．クヌース先生の書からの引用文は，数値計算プログラムの開発やメンテナンスを仕事とする者であれば誰でも同感するであろう.「有理数計算プログラミング環境」は，通常の Fortran や C などのコンパイラ型のプログラミング言語が備えている数値の整数型や浮動小数点数型に加えて，有理数型を備える．この型に対する算術演算，論理演算，浮動小数点数や整数から有理数への型変換，およびその逆方向の変換をサポートする．これらの機能によって次のことが可能になる．

- 丸め誤差の理論を学ぶ前の学生を対象とした数学教育
- 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で発生した精度に関係する問題を分析するツール
- 計算機援用証明（Computer Assisted Proof，2 つのアルゴリズムの計算結果の一致を確かめる）．
- ベンチマークの検収など，計算機システムの稼働確認

プログラムの概要　「有理数計算プログラミング環境」は，プログラミング言語 C++に，変数型 longint によって多桁の自然数を，変数型 rational によって有理数を付け加えることで**有理算術演算**（rational arithmetic）を可能とした．有理数の四則演算は $+, -, *, /$ の記号で，代入，比較演算，論理演算，入出力も C++ の演算子多重定義（オペレータオーバーロード）の機能により $=, ==, <, <<$ などの演算子記号

でプログラミングできる．多重精度算術演算（multiple-precision arithmetic）をサポートするプログラミング環境（例えば MPFUN など）は存在するが，これと異なる点は，四則演算だけで記述された計算は誤差なしで正確に行える点にある（多重精度算術演算は「より高精度な計算」を実現するが，正確かどうかはプログラマが注意深く調べなくては分からない場合が多い）．

無理数に対しては，変数型 interval を加えることで，**区間算術演算**（interval arithmetic）を $+, -, *$ の記号でプログラミングできる．除算は inv 関数で逆数を求めて，乗算を行う．これにより，無理数を解とする計算式の精度保証をしやすくした．また，多桁数や有理数を，精度 53 ビットの倍精度浮動小数点数と整数による指数部として抽出する変数型 rough も加えた．これによって，倍精度では桁溢れする絶対値の大きな数を用いる近似計算や表示もできる．

「有理数計算プログラミング環境」の目的のひとつは，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にある．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，Basic Linear Algebra Subprograms（BLAS）に代表されるような，数値線形代数（numerical linear algebra, NLA）計算で用いられる行列やベクトルに対する演算が高速化しやすく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，interval クラス，rblas クラス（有理数 BLAS クラス），prblas クラス（並列 rblas クラス）を階層的に構成することで実現される．

本資料は，第 1 章で全体の構成と注意事項を述べる．第 2 章で多桁数の演算を実現する longint クラス，第 3 章で有理算術演算を実現する rational クラス，倍精度演算の精度で概算するが倍精度演算のように桁溢れしない rough 数を扱うための rough クラス，区間演算のための interval クラス，第 4 章で数値線形代数計算をサポートする vector と matrix クラス，rblas クラス， prblas クラス，およびチェックポイントリスタート機能をサポートする iarchive と oarchive クラス，第 5 章で数値計算のための基本的な関数やユーティリティ関数をまとめた ratutil クラス，第 6 章で例題を解説する．

有理算術演算でユーティリティ的な関数と数値計算用関数を ratutil クラスに含めたが，これらの関数の説明の一部は，このプログラミング環境の本来の目的である**浮動小数点演算では解析不能の問題**を扱う例題に対する説明資料「有理数計算による対称行列の固有値問題における特性多項式の因子探索」に回した．例題 dvsrch は Divisor Search のからとった．この例題は，大学初年度の線形代数で学ぶ「基本変換行列」と，高校時代からお馴染みの「根と係数の関係」公式を利用して，対称行列の特性多項式の因数分解を代替するプログラムを作成するプログラミング演習（dvsrch.cpp，dvsrch1.cpp，dvsrch2.cpp を使用する）によって，本プログラミング環境の特徴である，**有理数と浮動小数点数の使い分けによる高速化**を学ぶためのものである．本資料の姉妹編である dvsrch.pdf を合わせて読まれるとこをお勧めする[1]．

本資料で説明する「有理数計算プログラミング環境」はダウンロードパッケージの ratprg2017Apr.tar ファイルから作成できる．

ヘッダーファイル .h: archive, interval, longint, longint_cexp, mempool, numeric_cast, rational, ratutil, rblas, prblas, rmatrix, rough, rtraits, rvector, thread_manager

C++ プログラムファイル .cpp: CG, CGsc, CfracPi, GauLeg, LDL, LLSschmidt, LU, LUhmg, PiCfracAtanInt, PiCfracAtanRint, archive, dvsrch, dvsrch1, dvsrch2, gettimeofday, interval, invmulsum, longint, mempool, numeric_cast, oltest, powersum, rational, ratutil, rough, testroughmat, thread_manager

Makefile: Makefile

[1] dvsrch2.cpp では数値計算プログラムで倍精度浮動小数点演算で作成した対称行列を取り出して，有理算術演算で誤差なしで計算する．数値計算プログラムとして『HPC プログラミング』に記載の 2 次元トラス構造解析プログラム CT2D に固有振動モード解析を加えて使用した．dvsrch2.cpp で CT2D が生成した行列を解析する場合は，CT2D のオーム社ホームページから CT2D/FT2D のダウンロードと，ダウンロードパッケージの CT2Dnorm.tar ファイルから作成する．

データファイル: S13free.txt, S13FreeO3.txt, S25Free.txt, S41Free.txt, S41FreeO3.txt, S13freeA2.chk, S13FreeO3A2.chk, S25FreeA2.chk, S41FreeA2.chk, S41FreeO3A2.chk

十進 BASIC プログラム .BAS: CCardan, CCubicEqu, CQuadEqu, CfracAtanPi, CfracPi, CombSet, Combination, HanoiStackMR, HesFrb, NewtonSDFig, complexityvieta, complexityvietatab, multvieta, symeig

十進 BASIC プログラムの多くは，資料 dvsrch.pdf に記載がある．

本資料の**索引**は，本プログラミング環境の関数名を中心に作成した．プログラムのコードを読んでいるとき，索引から本資料中の説明箇所を探す目的で作成したので，日本語のキーワードは含まない．

本プログラミング環境は，早稲田大学理工学術院，大石進一先生のホームページからダウンロード可能とした．

http://www.oishi.info.waseda.ac.jp/˜ samukawa/index.html/.

最初の版は 2015 年 6 月に作成したが，その後の**更新履歴**を記す．

2015 年 8 月 変数型 interval を追加した．

2017 年 4 月 並列化 prblas クラスと rough クラスと numeric_cast クラスを追加した．

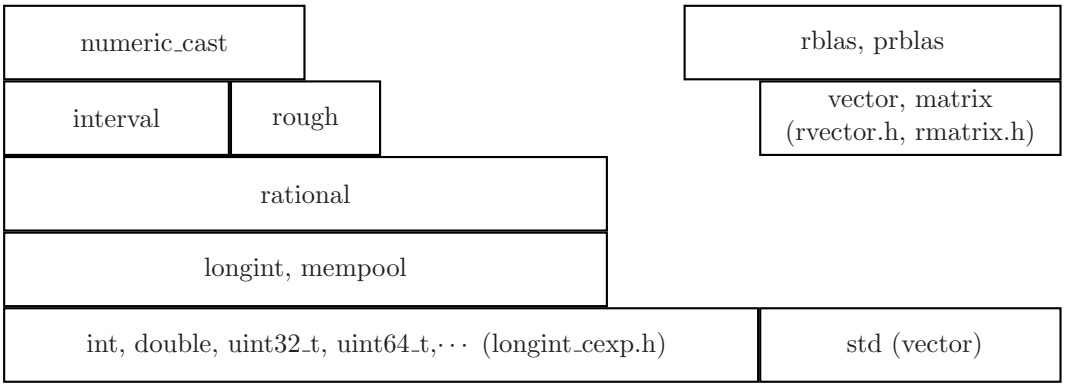


図 1: クラス間の階層構造

1 有理数計算プログラミング環境の構成と注意事項

図 1 に各クラスで扱う変数型を，階層的な関係として示した．図には int 型などの C++ の標準的なプログラミング環境の変数型を最下段に示し，その上に longint，rational，interval，rough クラスを構築したことを示した[2]．行列計算を扱いやすくするために vector，matrix クラスを設けて，その上に数値線形代数の並列化のために rblas と prblas クラスを構築した．名前空間 std に vector クラスがあり，これが vector クラスで扱う変数の型 vector と当たる[3]．そのため，名前空間 rat により，有理数計算プログラミング環境で定義した新しい変数型を含むようにした．使用にあたっては，`using namespace std` を指定しないプログラミングが必要になる．

C 言語で `unsigned long` 型は，32 ビットアーキテクチャでは 32 ビット，64 ビットアーキテクチャでは 64 ビットになる（符号なし整数はアドレスの計算のために用意された）．そこでプログラムがどちらのアーキテクチャでも稼働するように，`unsigned long` 型は使用せず，`uint32_t` と `uint64_t` の 2 つの型を使用することで，32 ビット計算機と 64 ビット計算機の違いを吸収する．この機能は C++ の言語標準の規格 C++11 がサポートされるコンパイラであれば使用可能である．コンパイラがこれに適合していない場合は，コンパイル時に “-std=c++11” のオプションを指定する．

符号付き整数も `int32_t` と `int64_t` を使用している．

なお，longint_cexp.h には，uint32_t が使えない環境では，これを定義するテンプレート関数を入れてある．

```
template <bool cond, typename Then, typename Else>
struct IF
{
    typedef Then RET;
};

template <typename Then, typename Else>
struct IF<false, Then, Else>
{
    typedef Else RET;
};

// for 32bit-64bit compiler compatibility
```

[2]longint 型の変数は，整数ではなく，正の数，自然数しか扱えないため，数学問題をこれで解く機会は少ない．整数を扱う場合は，rational 型を使用すればよい（分母の 1 が加わるだけである）．

[3]「名前空間」はコンパイラ用語で「名前の集合が全体として構造化されている場所」という意味である [1, p. 122]

```
typedef IF<sizeof(int) == 4,
            int,
            long int
          >::RET lint32_t;        // 32-bit signed int

typedef IF<sizeof(unsigned int) == 4,
            unsigned int,
            long unsigned int
          >::RET uint32_t;       // 32-bit unsigned int

typedef IF<sizeof(long int) == 8,
            long int,
            long long int
          >::RET lint64_t;        // 64-bit signed int

typedef IF<sizeof(long unsigned int) == 8,
            long unsigned int,
            long long unsigned int
          >::RET uint64_t;       // 64-bit unsigned int
```

テンプレート引数にマクロプログラムを記述して，その変数のサイズ（sizeof(int)）が 4 バイトか 8 バイトかを知ることで，処理系の計算機が 32 ビットアドレスの命令セットか 64 ビットアドレスの命令セットかを判断し，変数型 uint32_t か uint64_t の中身を決める．これはコンパイル時に，プリプロセッサのマクロ処理と同様のタイミングで行われる．これによって，プログラムは，unsigned long t; の代わりに uint32_t t; を，また unsigned long long t; の代わりに uint64_t t; と書くことで，32 ビットマシンと 64 ビットマシンの違いを吸収できる．

なお，lint32_t t; と lint64_t t; も処理したので，符号つき整数も lint32_t と lint64_t で型を指定できる．

2 longint クラス

多桁の自然数（正の整数，以下,「多桁数」） A は

$$A = \sum_{i=1}^{l} d_i r^{i-1} \tag{1}$$

によって表される．基数（radix または base）は $r = 2^{32}$ である．

longint.h で定義される longint クラスのメンバ変数を示す[4]．定数 TLS はマルチスレッド用である（後述）．

longint クラスのメンバ変数

```
class longint {
private:
   int capacity;
   int l;
   uint32_t *d;
#ifdef TLS                    // Thread Local Storage
   static bool pool_initialized;
#endif // TLS
```

式 (1) の多桁数の 1 桁 d_i は，32 ビット符号なし整数型の配列の 1 要素 d[i] に格納される．

- capacity は，現在使用できる桁数
- l は，現在使用している桁数（$l = 0$ の場合，$A = 0$ ）
- *d に配列のポインタ

が格納されている．なお，**零は桁数が零で判断しているので，実質的にメモリ負荷はほとんどない**．d_i は各桁の数値で $0 \le d_i < r$ で，**可変長の配列 d[...] に，式 (1) の添字と配列のインデックスを一致させて格納**される．d[A.l] に最上位の桁，d[1] に最下位の桁が格納され，d[0] は現在は使用していない（ゼロが入っている）[5]．$r = 2^{32}$ を用いており，プログラムでは変数 BASE によって定義される[6]．

多桁数は，算術演算の実行中に桁数が増えて，可変長の配列 d[...] を拡張する必要が生ずるので，演算実行時に配列を拡張する仕組みを設けた．配列の拡張のたびに new や delete 演算子を頻繁に使用するとオーバーヘッドが大きくなるので，獲得した配列をシステムに返却せずにプールして，後続の配列の獲得要求に再利用させる「プール方式」を採用した．この方式は mempool クラスによって実現され，longint クラスは mempool と連携して可変長配列を実現する．本章でははじめに可変長配列の実装方法を解説する．その後，多桁数の演算をサポートするための基本的なビット単位の処理を行う演算を説明した後，四則演算を解説する．さらにユーティリティ関数と最大公約数やプロファイラについて解説し，最期に簡単な例題を示す．

2.1 mempool クラスによる多桁数の可変長配列による実装

C++ では「4 大関数」と呼ばれる，コンストラクション，デストラクション，コピーコンストラクション，代入演算子（=）を定義する関数が，見えないところで自動的に呼び出されて動く．コンストラクタ

[4] C++では，C 言語の構造体 struct はクラスにする（構造体は C++ ではデータだけのクラスである）．構造体でフィールドと呼ばれた変数は，メンバ変数と呼ばれる．実体化（メモリを確保）したクラスを「オブジェクト」または「インスタンス」と呼ぶ．クラスとオブジェクトの関係は,「クラスが設計図でオブジェクトがそれに従って作られた製品」という説明がよくなされる．

[5] 将来，負数，NaN，無限大などの数を扱うために残した．

[6] 2^{32} を#define 文で定義できないコンパイラがあるので，longint.cpp で uint64_t longint::BASE = (uint64_t)65536u*(uint64_t)65536u; で設定している．

関数は，そのクラスをインスタンス化したとき，自動的に呼び出される特別なメンバ関数である[7]．これらの関数を明示的にプログラミングしなかった場合には，コンパイラによって自動的に生成されるが，動的メモリを使用するようなプログラムでは，これらの関数を書く必要がある [2, p. 220]．

本節ではまずはじめに可変長配列の実装方法を，逐次処理を前提に解説する．次に，可変長配列のマルチスレッド対応を説明し，最後に，コピーコンストラクタ，代入について説明する．

2.1.1 可変長配列のサイズ

「配列のプール方式」は，動的に獲得した配列を解放するとき，システムに対しては解放せずにこれをプールし，後続する獲得要求に対して再利用させる．インクルードしている mempool.h を示す．

mempool.h

```
#include "longint_cexp.h"
#define MINDIGITS 64
#define MAXDIGITS 32768

class mem_pool {
public:
    static const int mindigits = MINDIGITS; // should be power of 2
    static const int maxdigits = MAXDIGITS;
    static const int _poolvsize = Poolvsize<MINDIGITS,MAXDIGITS>::result;
    mem_pool();
    virtual ~mem_pool();      デストラクタは仮想関数にしておく

    int getpoolindex(int size, int id);
    uint32_t* dalloc(int idx);             new で動的（dynamic）に配列を獲得する
    void dfree(uint32_t* p,int idx);
    inline int psize(int idx) { return _psize[idx]; }
private:
    int _midx;
    int* _psize;
    void** _poolv;              管理用配列
};
```

配列サイズは，64，128，256，512，1024，2048，4096，8192，16384，32768 の長さを用いる．最初は 64 を獲得し，多桁数の桁数が増加して不足した時点で 128 を獲得して，新しい配列にコピーする．元の 64 のサイズの配列をシステムに返さず，プールに登録する．

longint_cexp.h にはコンパイル時に 2 底の対数を使用するためのテンプレート関数を含めた．プール方式でテーブルのサイズを求めるのに使用している．

[7]コンストラクタが存在しなければならない理由の 1 つは，クラス内部にスコープが限定されるデータを外部からアクセスできないからである．したがって，新しく作られたオブジェクトに初期値を書き込むための特権を持つ関数がクラスの中に必要なのである．これは，**C から見るとちょっと大きい飛躍である．C では，変数は定義の時に代入して初期化するだけであり，さらに初期化しないでおくことすらできる** [1, p. 66].「自動的」に呼出されるので，オブジェクトが初期化されないで参照されることで発生するエラーを防ぐこともできる．コンストラクタにはクラスと同じ名前を付ける．戻り値は存在しないので，戻り値の型の指定は省略する（void ではいけない）．longint X(1UL) と書くと「引数をとるコンストラクタ」が呼び出される．longint Y=longint(1UL) と書くと「代入演算子」が呼び出される．コピーコンストラクタは，クラスが引数に使われるとき，それを値渡しにするときに自動的に呼び出される．メンバ変数にポインタが含まれる場合，自動的に生成されるコピーコンストラクタはポインタだけをコピーする（shallow copy）．ポインタが指す先の変数もコピーする（deep copy）必要があるか否かは，コンパイラには分からない．指す先の変数もコピーする必要がある場合は，プログラマが明示的にコピーコンストラクタ（copy constructor）を書かなくてはならない．

longint_cexp.h の対数計算

```
template<int i> struct Log2 {
    static const int result = 1 + Log2<i/2>::result;
};
template<> struct Log2<1> {
    static const int result = 0;
};
template<int min, int max> struct Poolvsize {
    static const int result = Log2<max>::result - Log2<min>::result + 1;
};
```

2.1.2 管理用配列

longint クラスを使用するプログラムを起動すると，コンストラクタが起動する．ここで多桁数を可変長配列で管理するための 3 つの配列 _psize，_poolv，_numreallocv を初期化する．mindigits は longint.h で 64 に，maxidx は 32768 に定義されているので，maxidx は 9 となり，長さ 10 の配列が獲得される．

mempool.cpp の初期化ルーチンを示す．

mempool クラスのコンストラクタ

```
#include <iostream>
#include <stdexcept>
#include "mempool.h"

mem_pool::mem_pool() {
    int maxidx, sz;
    _midx = 0;
    _psize = new int[Poolvsize<MINDIGITS,MAXDIGITS>::result];
    _poolv = new void*[Poolvsize<MINDIGITS,MAXDIGITS>::result];
    maxidx = 0;
    sz = mindigits;
    while (sz < maxdigits) { sz *= 2; ++maxidx; }
    for (int i=0, sz = mindigits; i < _poolvsize; sz *= 2, ++i) {
        _psize[i] = sz;
        _poolv[i] = 0;
    }
}
```

図 2 に初期化された状態を示す．

2.1.3 配列の定義と削除（コンストラクタとデストラクタ）

多桁数用の配列 d[...] の長さは，計算の進行に伴って，長くなるが，参照されるだけの行列の要素のように長くならないものもある．このような，長さの異なる多桁数を，計算の進行に応じて，短い数と混在させて，効率よく使用するために，2^{32} 進数で 64 桁から，動的に，倍，倍に拡張して，最大で 32768 桁までを扱う．

longint クラスの変数 A が定義されると，longint クラスのインスタンスが生成される．ここでは例として

```
longint A(23L);
```

によって longint 型変数 A に，long int 型の数 23 を格納するステートメントを用いる．この文が実行されると，コンストラクタが（暗黙に）起動する．

_psize	_poolv	_numreallocv
64	0	0
128	0	0
256	0	0
512	0	0
1024	0	0
2048	0	0
4096	0	0
8192	0	0
16384	0	0
32768	0	0

図 2: 管理用配列の初期化

longint クラスのコンストラクタ

```
longint::longint() {
    try {
        this->d = lalloc(mindigits, this->capacity);
    } catch(std::runtime_error& e) {
        cerr << "longint::longint(): lalloc failed: mindigits=" << mindigits << "endl";
        exit(99);
    }
    this->l = 0;
}
```

3 行目の lalloc 関数の中で，uint32_t 型の長さ 64 の配列が獲得される[8]．lalloc は Longint allocate の意である．lalloc 関数を示す．

longint クラスの lalloc 関数

```
uint32_t* longint::lalloc(int size, int& allocated_size) {
   uint32_t* p;
   try {
      if (size > maxdigits) {
         p = new uint32_t[size];
         allocated_size = size;
      } else {
         mem_pool* mp;
         if ((mp = _mempool) == 0) {
            _mempool = new mem_pool();
            mp = _mempool;
         }
         assert(mp != NULL);
         int idx = mp->getpoolindex(size,150);
         p = mp->dalloc(idx);              <-------- dalloc 関数（mempool クラス）
         allocated_size = mp->psize(idx);
      }
   } catch(std::bad_alloc& e) {
      std::cerr << "bad_alloc in lalloc: size=" << size << std::endl;
      exit(99);
   }
   return p;
}
```

[8]計算の途中では，計算規模が大きくなっている状況では midx に記録されたインデックスサイズの配列長が獲得される．

実際の配列の動的獲得は，mempool クラスの dalloc 関数が行う．

mempool クラスの dalloc 関数

```
uint32_t* mem_pool::dalloc(int idx) {
   uint32_t* d;
   if (_poolv[idx] == 0) {
      int size = _psize[idx];
      d = new uint32_t[size];      new 演算子で符号なし 32 ビットの size 長
   } else {
      d = (uint32_t*)_poolv[idx];
      _poolv[idx] = ((void**)_poolv[idx])[0];
   }
   return d;
}
```

_poolv 配列に利用可能なものがない（“0” が格納されている）場合は，new によってそのサイズの配列を獲得し，そのポインタを返す．利用可能なものがあれば，その（_poolv[idx] に格納されている）ポインタを返す．

プログラムの制御が，変数 A のスコープの外に出る時，変数 A のインスタンスを取り除くデストラクタ ~longint が起動する．

longint クラスのデストラクタ

```
longint::~longint() {
   lfree(this->d,this->capacity);
}
```

lfree 関数を示す．

longint クラスの lfree 関数

```
void longint::lfree(uint32_t* p, int size) {
   if (size > maxdigits) {
      delete[] p;
   } else {
      mem_pool* mp = _mempool;
      assert(mp != 0);
      int idx = mp->getpoolindex(size,151);
      mp->dfree(p,idx);              <-------- dfree 関数（mempool クラス）
   }
}
```

dfree 関数は後述する．

2.1.4 配列の拡張

配列長に対応する管理用配列の添字は，getpoolindex 関数によって得られる．

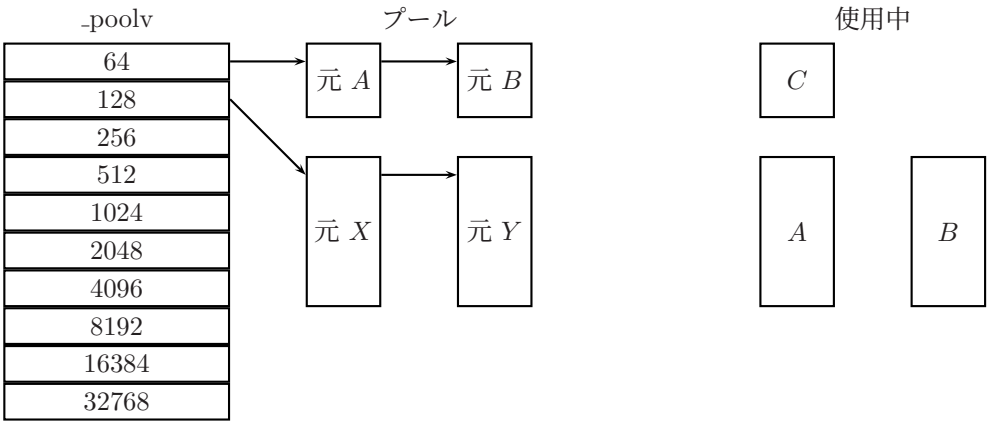


図 3: 配列のプールと再利用（拡張前）

mempool クラスの getpoolindex 関数

```
int mem_pool::getpoolindex(int size, int id) {
    int idx, sz;
    if (size < mindigits || size > maxdigits) {
        cout << "mem_pool::getpoolindex: size out of valid range (size=" << size <<
                ", id="<< id << ")" << endl;
        throw std::runtime_error("memory allocation error");
    }
    idx = 0;
    sz = mindigits;
    while (sz < size) { sz *= 2; ++idx; }
    return idx;
}
```

有理数演算が進行して，n 桁の A と m 桁の B を 128 の長さの配列で計算しているとしよう（n と m は 64 桁より大きく，128 桁以下）．ここで $C = A \cdot B$ を計算するが，C はまだ 64 の長さの配列に置かれている．この場合，乗算ルーチンの中で桁の不足が検知される．図 3 に，使用中の A，B，C と，プールに登録された元 A，B，X，Y が，64 または 128 の長さの配列として管理されている状態を示した．_poolv[0] には 64 の長さの配列（元 A）のポインタが格納され，元 A の配列の最初の要素には別の配列（元 B）のポインタが格納されている．同様に _poolv[1] には 128 の長さの配列（元 X）のポインタが格納され，元 X の配列の最初の要素には別の配列（元 Y）のポインタが格納されている．このように，複数の使用済みの配列が，リンクリストによってつなげられる．

realloc 関数　配列の不足が検知されると，次のように realloc 関数が呼ばれる．

```
C.realloc(C.l+1,1009);
```

realloc 関数の第 1 引数が必要な配列長（サイズ）である．前述したコンストラクタが使用する lalloc 関数によって，必要な配列を獲得し，多桁数のデータ d[...] を現在使用している配列から，新しい配列にコピーし（for ループ），不要になった配列を lfree で返却する．

```
void longint::realloc(int size, int id) {
   int new_size = 0;
   uint32_t* p = lalloc(size, new_size);   <--- lalloc 関数
   for (int i=0; i <= this->l; ++i) {
      p[i] = this->d[i];
```

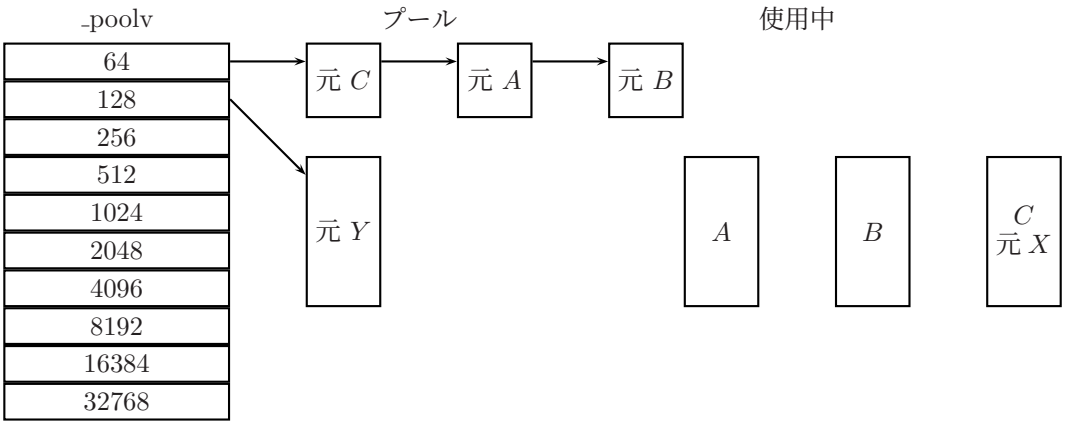


図 4: 配列のプールと再利用（拡張後）

```
    }
    lfree(this->d,this->capacity);
    this->d = p;
    this->capacity = new_size;
}
```

lalloc 関数が呼出す，13 ページに示した mempool クラスの dallocdalloc 関数の中で，次のコードによって_poolv[idx] のリンク先を付け替える．

```
_poolv[idx] = ((void**)_poolv[idx])[0];
```

_poolv[idx] の中身（* _poolv[idx]）が指しているものの中身なので，“*” が 2 つつく．これによって，新しいリンク先は，それまでリンクリストの 2 番目の位置にいた配列になる．

lfree 関数から呼ばれる dfree 関数　古く小さい配列から，新配列へのデータの移動（for ループ）が完了したら，lfree 関数を呼び出す．13 ページに前述した lfree 関数は，size が最大の場合は delete し，そうでない場合は mempool クラスの dfree 関数によって（システムに対しては解放せずに）プールする．dfree 関数を示す．

```
void mem_pool::dfree(uint32_t* d,int idx) {
    ((void**)d)[0] = _poolv[idx];
    _poolv[idx] = (void*)d;
}
```

それまで _poolv[idx] が指していた「元 A」へのポインタを，古い配列「元 C」が指すようにする．これには “((void**)d)[0] = _poolv[idx];” を用いる．そして _poolv[idx] には，古い配列「元 C」へのポインタを格納する（_poolv[idx] = (void*)d;）．これで 64 の長さの配列のリンクリストに，新たな利用可能配列「元 C」を加え，図 4 の状態になる．

2.2　マルチスレッド対応

マルチスレッド対応とする理由は，同時に複数のスレッドが配列拡張を行うときに，ロックで遅くならないようにするためである．ここでは Thread Local Storage を使用して，グローバル変数で共通でみな同じ変数名で参照するが，内容はスレッドごとに別アドレスが持てるようにした．

GNU では __thread キーワードでサポートされる．MacOS ではインターフェースが異なり，スレッドキー番号を受け取る方式になる．

マルチスレッド版 mempool の変数

```
#ifdef MEMPOOL
#ifdef TLS
__thread mem_pool* longint::_mempool;
bool longint::pool_initialized = false;
#else   // TLS
mem_pool* longint::_mempool = NULL;
#endif  // TLS
#endif  // MEMPOOL
uint64_t longint::BASE = (uint64_t)65536u*(uint64_t)65536u;
```

init による初期化ルーチンを示す．

マルチスレッド版の init() 関数

```
bool longint::init() {
#ifdef TLS
    if (!pool_initialized) {
        pool_initialized = true;
    }
#endif  // TLS
    return true;
}
```

lalloc のマルチスレッド版を示す．

マルチスレッド版 lalloc

```
uint32_t* longint::lalloc(int size, int& allocated_size) {
    uint32_t* p;
    try {
        if (size > maxdigits) {
            p = new uint32_t[size];
            allocated_size = size;
        } else {
            mem_pool* mp;
#ifdef TLS
            if (!pool_initialized) {
                longint::init();
            }
            if ((mp = _mempool) == 0) {
                _mempool = new mem_pool();
                mp = _mempool;
            }
#else   // TLS
            if ((mp = _mempool) == 0) {
                _mempool = new mem_pool();
                mp = _mempool;
            }
#endif  // TLS
            assert(mp != NULL);
            int idx = mp->getpoolindex(size,150);
            p = mp->dalloc(idx);
            allocated_size = mp->psize(idx);
        }
    } catch(std::bad_alloc& e) {
        std::cerr << "bad_alloc in lalloc: size=" << size << std::endl;
        exit(99);
    }
    return p;
}
```

lfree マルチスレッド版は逐次処理と同じ．

2.3 コンストラクタ

コンストラクタとデストラクタは前述したので，本節ではその他のコンストラクタを示す．

2.3.1 コピーコンストラクタ

コピーコンストラクタは，オブジェクトの初期化時に呼び出され，クラスをそっくりコピーするために使用する．このコードはオブジェクトのコピーがとられた時に実行する処理を記述する．

longint のコピーコンストラクタ

```
longint::longint(const longint& rhs) {
    try {
        this->d = lalloc(rhs.capacity, this->capacity);
    } catch(std::runtime_error& e) {
        std::cerr << "longint::longint(const longint&): lalloc failed: rhs.capacity=" <<
            rhs.capacity << std::endl;
        assert(false);
        exit(99);
    }
    this->l = rhs.l;
    for (int i=0; i < rhs.l+1; ++i) {
        this->d[i] = rhs.d[i];
    }
}
```

2.3.2 パラメータをとるコンストラクタ

uint32_t 型の数の引数をとるコンストラクタを示す.

uint32_t のコピー

```
longint::longint(uint32_t n) {
    this->d = lalloc(mindigits, this->capacity);
    this->l = 0;
    if(n != 0)  {
        int i = 1;
        do {
            this->d[i] = n % BASE;
            n = n / BASE;
            i = i + 1;
        } while (n > 0);
        this->l = i - 1;
        if(this->l > longint::longintlmax) longint::longintlmax=this->l;
    }
}
```

uint64_t 型の数のコピーコンストラクタも同様である.

string 型の数の引数をとるコンストラクタを示す. longint 型の変数に, 文字列で数値を記入する際に用いる. 使用例は後述する.

string のコピー

```
longint::longint(const std::string& s) {
    uint64_t dd;
    this->d = lalloc(mindigits, this->capacity);
    this->l = 0;
    if (s.size()==0 || s[0]=='0') return;
    this->l = 1;
    this->d[1] = 0;
    for (unsigned int i=0; i < s.size(); ++i) {
        // Multiply d by 10
        dd = 0llu;
        for (int j=1; j <= this->l; ++j) {
            dd = (uint64_t)this->d[j]*(uint64_t)10u + dd;
            this->d[j] = dd % BASE;
            dd = dd / BASE;
        }
        if (dd != 0) {
            this->d[++this->l] = dd;
        }
        // Add a digit
        dd = (uint64_t)(s[i] - '0');
        for (int j=1; j < this->l+1 && dd != 0llu; ++j) {
            dd = (uint64_t)this->d[j] + dd;
            this->d[j] = dd % BASE;
            dd = dd / BASE;
        }
        if (dd != 0llu) {
            if (this->capacity < this->l+2) this->realloc(this->l+2,1002);
            this->d[++this->l] = dd;
        }
    }
}
```

配列と配列長によるコンストラクタを示す.

配列と配列長によるコピー

```
longint::longint(uint32_t* d, int l) {
    this->d = lalloc(l+1 >= mindigits ? l+1 : mindigits , this->capacity);
    this->l = l;
    for (int i=0; i <l; ++i) { this->d[i+1] = d[i]; }
}
```

コンストラクタとデストラクタのメニューを示す.

コンストラクタ

```
    longint();
    longint(const longint&);
    longint(uint32_t);
    longint(uint64_t);
    longint(const std::string&);
    longint(uint32_t*,int);
    virtual ~longint();
```

2.3.3 表示用コンストラクタ

表示（標準出力）用コンストラクタを示す.

16 進数による表示コンストラクタ hexstr

```
std::string longint::hexstr() const {
    std::stringstream result;
    result << "{";
    for (int i=this->l; i >= 1; --i) {
        result << " " << std::hex << this->d[i];
    }
    result << "}";
    return result.str();
}
```

16 進数を表示出力する．

表示コンストラクタ

```
std::string longint::str() const {
    std::string result;
    char s[1000];
    sprintf(s, "{ capacity: %d, l: %d, d=[", this->capacity, this->l);
    result += s;
    result += this->hexstr();
    result += "]}";
    return result;
}
```

2.3.4 使用例 oltest.cpp

使用例を示す．多桁数 longint 型だけを使用する場合，archive.h，longint.h，longint_cexp.h，mempool.h，rmatrix.h，rvector.h がヘッダーファイルとして必要になる．また mempool.cpp と gettimeofday.cpp も必要である．

使用例（oltest.cpp）

```
int main(){
    longint Q,R,A,W;
    longint n("37460982603786467977475937405429251066885674560548730137361643640583752057745228758301740419667133047733907342215746398736555221623489247420874752O");
    longint d("37460982603743579161690478295443400057793111540103673716195688833797690826789394933085434619727650601055312222385516833813086071928918206920439743");
    std::cout << "n=" << n << std::endl;
    std::cout << "d=" << d << std::endl;
    Q = longint::ldivide(n, d, &R);
    std::cout << "Q=" << Q << std::endl;
    std::cout << "n=" << n.str() << std::endl;
    std::cout << "d=" << d.str() << std::endl;
    std::cout << "Q=" << Q.str() << std::endl;
    std::cout << "R=" << R.str() << std::endl;
    W = d * Q;              演算子多重定義を利用した書き方
    A= W + R;
    if( A != n ){           演算子多重定義を利用した書き方
        printf(" BAD\n"); C 流の printf も可だが,...
    }else{
        printf(" OK\n");  cout << に変更したほうがよい.
}
```

string 型の数のコピーコンストラクタにより，多桁数の変数 n と d に十進数で桁数の多い数を格納できる．除算 $n \div d$ を行い，商を Q に，余りを R に得て，2^{32} 進数を 16 進数で表示する．この例は，最上位

の桁が $(77)_{16} = 112$ を $(b)_{16} = 11$ で割るので商 Q は 10 になり，検算の結果 $n = d * Q + R$ が確認されて “OK” の表示を得る．

使用例（oltest の実行結果）

```
n=3746098260378646797747593740542925106688567456054873013736164364058375205774522875830
17404196671330477339073422157463987365552216234892474208747520
d=3746098260374357916169047829544340005779311154010367371619568883379769082678939493308
54346197276506010553122223855168338130860719289182069204397 43
Q=10
n={ capacity: 64, l: 16, d=[{ 77 ffffffff ecd8101a b5c849d1 82679679 8d044625 5f380531
898c1dec 16273708 7f7650d7 bce0a56c 7ffff0d6 a2a7020a 0 0 0}]}
d={ capacity: 64, l: 16, d=[{ b ffffffff eefa8069 a7635a3f 64da182f 6dee80ea a9a155e4
da285c9b 634c91bf f1f02000 d10ae0de 4a08de72 7cf7383e cc214032 f873bd66 705eb7bf}]}
Q={ capacity: 64, l: 1, d=[{ a}]}
R={ capacity: 64, l: 14, d=[{ 970f0bfa 2be6c357 91e2a49f 41b33cfa beeaaa41 3f87fda
35298589 c1510cf 9273dcbd 9ba7405d c0fecf96 6b37e02 4b7a99ff 9c4cd28a}]}
 OK
```

2.4 デバッグ用関数

動的に配列を拡張するために，オーバーフローをチェックするルーチンを用意した．

checkoverflow

```
void longint::checkoverflow(int index, const string& tag) {
    if (this->capacity < index+1) {
        cerr << "Index out of bound at " << tag << ": index=" << index << ", capacity="
             << this->capacity << endl;
        assert(false);
    }
}
```

インデックスが配列の範囲を超える場合を調べる関数を示す．

checkbound

```
void longint::checkbound(int idx, int id) {
    if (idx < 0 || idx > this->l) {
        cerr << "checkbound: index out of bound: id=" << id << ", l=" << this->l
             << ", index=" << idx << endl;
        exit(99);
    }
}
```

これらの関数は，ソースプログラムには多く埋め込まれているが，プリプロセッサへの変数 CHECKOVERFLOW と CHECKBOUND で，異常時にリコンパイルして調べるようにしている．

チェック機能の使用例

```
#ifdef CHECKOVERFLOW
    void checkoverflow(int,const std::string&);
#endif
#ifdef CHECKBOUND
    void checkbound(int,int);
#endif
```

本資料ではこれらのチェックルーチンを呼出す部分は省いてプログラムを示す．

2.5 longint.h で定義されるメンバ関数 slz，swap，clzu，copy_digits，to_ullong

シフトレフト零を行う関数を示す..

slz

```
void slz() {
    int i = l;
    while (i > 0 && d[i] == 0) --i;
    l = i;
}
```

longint 数をスワップするには，データが長い配列の場合，実際に入れ替えると時間がかかるので，ポインターを入れ替える.

swap

```
static inline void swap(longint& a, longint& b) {
    std::swap(a.capacity,b.capacity);
    std::swap(a.l,b.l);
    std::swap(a.d,b.d);
}
```

リーディング零をカウントする関数を示す.

clzu

```
static inline uint32_t clzu(uint32_t a) {
    uint32_t count;
    count = 0;
    while ((a & 0x80000000) == 0u) {
        a <<= 1;
        ++count;
    }
    return count;
}
```

多桁数のコピーを行う関数を示す.

copy_digits

```
static inline void copy_digits(uint32_t* d, uint32_t* s, int len) {
    uint32_t* s_end = &s[len];
    while (s != s_end) {
        *d++ = *s++;
    }
}
```

多桁数の下位の 64 ビットを返す関数を示す[9].

to_ullong()

```
uint64_t to_ullong() const {
    if (l == 0) return 0;
    if (l == 1) return static_cast<uint64_t>(d[1]);
    return (static_cast<uint64_t>(d[2]) << 32) + d[1];
}
```

[9] この関数は rough クラスの longint2double 関数で使用している.

2.6 longint.cpp で定義されるビット演算

longint 数のトレーリング零ビットの数を数える関数を示す.

ctz

```
uint32_t longint::ctz() const {
    uint32_t di, bi;
    uint32_t t;
    if (this->l == 0) return 0;
    for(di=0; this->d[di+1] == 0; ++di) ;
    for(t=this->d[di+1], bi=0; (t & 1)==0; t >>= 1, ++bi);
    return di*32+bi;
}
```

ビット列を左シフトする関数を示す．この関数はシフトするビット数が 32 よりも大きい場合は，語単位でシフトし，残りをビットごとにシフトする.

shl

```
void longint::shl(uint32_t n) {
    uint32_t w = n / 32;
    uint32_t b = n % 32;
    if (l != 0 && n != 0) {
        uint32_t i;
        if (b == 0) {                                    32 で割切れた場合
            if (capacity < l+w+1) realloc(l+w+1, 3);
            for(i = l; i > 0; --i) { d[i+w] = d[i]; }  語単位でシフト
            for (i = w; i > 0; --i) { d[i] = 0; }
            this->l += w;
        } else {
            uint32_t u, v;
            if (capacity < l+w+2) realloc(l+w+2, 3);
            u = d[l];
            d[l+1+w] = (u >> (32-b));
            for (i = l; i > 1; --i) {
                v = d[i-1];
                d[i+w] = (u << b | v >> (32-b));
                u = v;
            }
            d[w+1] = u << b;
            for (i = w; i > 0; --i) { d[i] = 0; }
            this->l += (w+1);
            this->slz();
        }
    }
}
```

ビット列を右シフトする関数を示す..

shr

```
void longint::shr(uint32_t n) {
    uint32_t nw = n/32;
    uint32_t nb = n%32;
    if (l <= nw) {
        this->l = 0;
    } else if (nb == 0) {
        uint32_t m = this->l-nw;
        uint32_t n = 0;
        if (m >= 4) {
            n = m & 0xfffffffc;
            for (uint32_t i=0; i < n; i+=4) {
                this->d[i+1] = this->d[i+nw+1];
                this->d[i+2] = this->d[i+nw+2];
                this->d[i+3] = this->d[i+nw+3];
                this->d[i+4] = this->d[i+nw+4];
            }
        }
        for (uint32_t i=n; i < m; ++i) { this->d[i+1] = this->d[i+nw+1]; }
        this->l = this->l - nw;
    } else {
        uint32_t m = this->l-nw-1;
        uint32_t n = 0;
        uint32_t nbl = 32 - nb;
        if (m >= 4) {
            n = m & 0xfffffffc;
            for (uint32_t i=0; i < n; i+=4) {
                this->d[i+1] = (this->d[i+nw+1] >> nb | this->d[i+nw+2] << nbl);
                this->d[i+2] = (this->d[i+nw+2] >> nb | this->d[i+nw+3] << nbl);
                this->d[i+3] = (this->d[i+nw+3] >> nb | this->d[i+nw+4] << nbl);
                this->d[i+4] = (this->d[i+nw+4] >> nb | this->d[i+nw+5] << nbl);
            }
        }
        for (uint32_t i=n; i < m; ++i) {
            this->d[i+1] = (this->d[i+nw+1] >> nb | this->d[i+nw+2] << nbl);
        }
        this->d[l-nw] = (this->d[l] >> nb);
        this->l -= nw;
    }
    this->slz();
}
```

ループ展開（loop unrolling）による高速化の処理を行っている．

2.7 代入・論理・入出力演算子・定数

演算子を多重定義により定義しているが，四則演算を除く定義を本節で抜粋して紹介する．

2.7.1 演算子多重定義のメニュー

次の演算子を使用できる．

```
// overloaded operators
longint& operator =(const longint&);  代入
longint& operator =(uint32_t);
longint& operator =(uint64_t);
longint& operator <<=(int);            ビットシフト
```

```
longint& operator >>=(int);
longint operator <<(int) const;
longint operator >>(int) const;
bool operator ==(const longint&) const;  比較演算
bool operator !=(const longint&) const;
bool operator >(const longint&) const;
bool operator >=(const longint&) const;
bool operator <=(const longint&) const;
bool operator <(const longint&) const;
longint& operator |=(const longint&);
longint& operator +=(const longint&);  四則演算
longint& operator -=(const longint&);
longint& operator *=(const longint&);
longint& operator /=(const longint&);
longint operator +(const longint&) const;
longint operator -(const longint&) const;
longint operator *(uint32_t) const;
longint operator *(const longint&) const;
longint operator /(const longint&) const;
longint operator |(const longint&) const;
operator double();
```

2.7.2 代入文

longint 型変数の代入文を示す.

代入演算子 = longint

```
longint& longint::operator =(const longint& rhs) {
    int i;
    if (this == &rhs) return *this; // self assignment!
    if (this->capacity < rhs.capacity) {
        this->l = 0;    // to avoid copying
        this->realloc(rhs.capacity,1004);
    }
    this->l = rhs.l;
    for (i=0; i < rhs.l+1; ++i) { this->d[i] = rhs.d[i]; }
    return *this;
}
```

2.7.3 ビット演算

左へのビットシフトを示す．コンパイルオプションで USESHL を指定すると，23 ページに示した shl を用いる[10].

[10] シフトする longint 数の桁数が多い場合，shl による高速化の効果は大きく，10 倍以上の差が出る．

ビットシフト演算子 <<=

```
longint& longint::operator<<=(int n) {
#ifdef USESHL
    this->shl(n);
#else
   uint32_t c;
   for (int j=0; j < n; ++j) {
      c = 0;
      for (int i=1; i <= this->l; ++i) {
         uint32_t msb;
         msb = this->d[i] >> 31;
         this->d[i] = ((this->d[i] << 1) | c);
         c = msb;
      }
      if (c != 0) {
         if (this->capacity < this->l+2) {
            this->realloc(this->l+2,1007);
         }
         this->d[++this->l] = c;
      }
   }
#endif
   return *this;
}
```

右シフトも同様であるが，ここでは省略する．左シフトに対して USESHL コンパイルオプションを指定したように，右シフトに対しても USESHR を指定すると，shr を呼出し，語単位のシフトで高速化する．

= を含まない場合は次の演算子になる．

ビットシフト演算子 >>

```
longint longint::operator>>(int n) const {
   longint result = *this;
   result >>= n;
   return result;
}
```

2.7.4 比較演算

longint 型の 2 数の比較演算子 == を示す．

比較演算子 ==

```
bool longint::operator ==(const longint& rhs) const {
   return lcmp(*this,rhs) == 0;
}
```

lcmp は次のように実装している．

longint.cpp の lcmp 関数

```
int longint::lcmp(const longint& A, const longint& B){
   int i;
   if(A.l > B.l)  return 1;
   if(B.l > A.l)  return -1;
   for (i=A.l; i > 0; --i) {
      if (A.d[i] != B.d[i]) break;
   }
   return (i == 0 ? 0 : (A.d[i] > B.d[i]) ? 1 : -1);
}
```

2.7.5 定数

次の定数を定義している．

longint.cpp の 定数の定義

```
const longint longint::ZERO = (uint32_t)0u;
const longint longint::ONE = (uint32_t)1u;
const longint longint::TWO = (uint32_t)2u;
const longint longint::THREE = (uint32_t)3u;
const longint longint::FOUR = (uint32_t)4u;
const longint longint::FIVE = (uint32_t)5u;
const longint longint::SIX = (uint32_t)6u;
const longint longint::TEN = (uint32_t)10u;
const longint longint::SIXTEEN = (uint32_t)16u;
const longint longint::FOURGIGA = (uint64_t)4294967296u;
const longint longint::FOURPETA = (uint64_t)4503599627370496u;
```

2.8 四則演算

2 つの多桁数の加算，減算，乗算，除算を説明する．乗算と除算は計算速度に注意する必要があるので，コンパイルによって生成されるコードを確認する．

2.8.1 加減算

インクリメント演算子 $+=$ による 2 つの桁数の異なる多桁数の加算 this + rhs を示す．this の桁数と rhs の桁数の比較を行い，大小で 2 通りに分け，前半は被加数 this のほうが大きい場合を扱う．最初のループは，各桁の加算を，下位の桁から 64 ビット演算で行う．反復は，被加数，加数，キャリーを加えて，this->d[i] に格納し，上位 32 ビットをキャリーに保存する．2 つめのループは，被加数の上位桁の移動である．後半は加数のほうが大きい場合で，配列長のチェックを行ってから同様の処理を行う．rhs のほうが桁数が多い場合に，配列長の拡張が必要な場合がある．

```
longint& longint::operator+=(const longint& rhs) {
    uint64_t t;
    int i;
    if (this->l >= rhs.l) {
        i = 1;
        t = 0llu;
        while (i <= rhs.l) {
            t = t + (uint64_t)this->d[i] + (uint64_t)rhs.d[i];
            if (t < longint::BASE) {
```

```
                this->d[i] = t;
                t = 0llu;
            } else {
                this->d[i] = t - longint::BASE;
                t = 1llu;
            }
            ++i;
        }
        while (i <= this->l && t != 0llu) {
            t = t + (uint64_t)this->d[i];
            if (t < longint::BASE) {
                this->d[i] = t;
                t = 0llu;
            } else {
                this->d[i] = t - longint::BASE;
                t = 1llu;
            }
            ++i;
        }
        if (t != 0llu) {
            this->d[++this->l] = t;
        }
    } else {
        if (this->capacity < rhs.l+2) {   配列サイズのチェック
           this->realloc(rhs.l+2,2002);
        }
        i = 1;
        t = 0llu;
        while (i <= this->l) {
            t = t + (uint64_t)this->d[i] + (uint64_t)rhs.d[i];
            if (t < longint::BASE) {
                this->d[i] = t;
                t = 0llu;
            } else {
                this->d[i] = t - longint::BASE;
                t = 1llu;
            }
            ++i;
        }
        this->l = rhs.l;
        while (i <= rhs.l) {
            t = t + (uint64_t)rhs.d[i];
            if (t < longint::BASE) {
                this->d[i] = t;
                t = 0llu;
            } else {
                this->d[i] = t - longint::BASE;
                t = 1llu;
            }
            ++i;
        }
        if (t != 0llu) {
            this->d[++this->l] = t;
        }
    }
    return *this;
}
```

− = は this- rhs を求めるが，ここでは省略する．

2.8.2 乗算

$*=$ は `this*rhs` を求める．加減算が桁数に比例する計算量であるのに対し，乗算は桁数の積に比例するので，計算速度に注意する必要がある[11]．計算量は $O(n^2)$ であるが，FFT を用いると $O(n \log n)$ になる．現在，FFT は使用していない．

ここで示すプログラムは，Intel のプロセッサで，x86 と EM64T の命令セットを想定している．32 ビットマシンの場合は，整数の乗算命令 MUL は，32 ビットの被乗数と乗数を掛けて，64 ビットの積を 2 つのレジスタ eax と edx に返す．2 つの `uint32_t` 型の積は 64 ビットであるが，これを 32 ビット乗算命令で 2 つの 32 ビット長のレジスタに得て，上位 32 ビットを（次に示すプログラムの）変数 `carry` に得る．乗算は MUL 1 回だけで，64 ビット変数 `t` のシフトは行われない[12]．アセンブリコードは 33 ページの除算の mullu 関数で後述する．

```
longint& longint::operator *=(const longint& rhs) {
   if (this->l == 0 || rhs.l == 0) {
       this->l = 0;
   } else {
       longint res;
       int m = this->l;
       int n = rhs.l;
       size_t i, j;

       if (res.capacity < m+n+1) res.realloc(m+n+1, 0);
       res.l = m+n;
       for (i=0; i < res.l; ++i) res.d[i+1] = 0u;

       uint32_t carry;
       res.d[1] = 0;
       for (i=0; i < n; ++i) {
          carry = 0u;
          for (j=0; j < m; ++j) {
             uint64_t t=static_cast<uint64_t>(this->d[j+1])*rhs.d[i+1]+carry+res.d[i+j+1];
             carry = (t >> 32);
             res.d[i+j+1] = static_cast<uint32_t>(t & 0xffffffffu);
          }
          res.d[i+m+1] = carry;
       }
       res.slz();
       longint::swap(res,*this);
   }
   return *this;
}
```

Intel と GNU コンパイラの相性がよいが，他のアーキテクチャで実行する場合は，どのようなコードが生成されるかをチェックする必要がある．多くの 32 ビット計算機アーキテクチャで，32 ビット数の積は 2 つの 32 ビットのレジスタに入るように命令セットが定義されているが，高水準言語で書いたコードが，コンパイラによって効率的なコードに変換されるかどうかは分からない．

[11] n 桁と m の 2 数の乗算を筆算で書くと，m 個の n または $n+1$ 桁の数をずらして並べる．底辺 n で高さ m の平行四辺形を英語では multiplication pyramid という．ピラミッドの平行四辺形の外側を零で補って $n \times m$ の長方形にすると，畳み込み（コンボリューション）になるので FFT につなげられる．

[12] 被乗数と乗数の両者を `uint64_t` 型に変換してから掛けると遅くなる．

2.8.3 除算

除算は乗算以上に重い演算である．乗算と異なり，非線形計算で，仮商を立てて反復する[13]．浮動小数点計算では，除算を使用する機会を少なくするように工夫するが，有理算術演算で GCD を除算で求めるアルゴリズムを使用すると，有理数の四則演算のたびに何度も除算が呼ばれるので，除算がホットスポットになる．したがって，生成される機械語の命令に注意を払う必要がある．

`ldivide` 関数で longint 型の被除数 `a` を除数 `b` で割って，剰余 `r` を引数に，商を戻り値に返す[14]．除算は加減算や乗算にくらべると複雑な演算である．ここではクヌース先生の教科書の「アルゴリズム D」（255 ページから）に従って説明する [3, p. 255].

“longint u(a);” はコピーコンストラクタで，引数 `a` のコピーを `u` に作る．同様に引数 `b` のコピーを `v` に作る．除数が 2^{32} 進数で 1 桁の場合は，次のループプログラムで `divlu` によって longint 型の m 桁の被除数（第 1 引数）を，uint32_t 型の除数で割り，商は第 1 引数を置き換え，余りを返す．

ldivide の 1（前処理と除数が 1 桁の場合）

```
longint longint::ldivide(const longint& a, const longint& b, longint *r) {
    assert(b.l != 0);
    if (b.l == 0) {
        throw std::runtime_error("longint: divide by zero");
    }
    longint u(a);
    if (a < b) {
        if (r != 0) *r = a;
        u.l = 0;
        return u;
    }
    longint v(b);
    longint qv;
    longint q;
    if (b.l == 1) {                                  除数が 1 桁の場合
        uint32_t rr = divlu(&u.d[1],u.l,v.d[1]);
        u.slz();
        if (r != 0) {
            if (rr == 0u) { r->l = 0; }
            else { r->d[1] = rr; r->l = 1; }
        }
        return u;
    }
```

uint32_t 型の 1 桁の商を返す関数 `divlu` を示す．桁数のループでは，64 ビット変数に前の桁の剰余を 32 ビットシフト，下位 32 ビットに被除数の 1 桁を入れ，除数で割る．

[13] 仮商を予想するところは，筆算による割り算で，除数で割ったときに，商の 1 桁に「何が立つか」を決めるところである．被除数の上位 3 桁を，正規化された除数の 2 桁からこれを決定できる．この証明は教科書では演習問題にある．

[14] ldiv は C では予約語なので ldivide とした．

divlu

```
uint32_t longint::divlu(uint32_t* a, int m, uint32_t s){
    uint32_t r;
    uint64_t t;
    int i;
    i = m; r = 0u;
    while (i > 0) {
        uint64_t t;
        --i;
        t = (static_cast<uint64_t>(r)<<32) + a[i];
        a[i] = t / s;
        r = t % s;
    }
    return r;
}
```

被除数が m 桁，除数が n 桁の場合，教科書の「アルゴリズム D」は D1 から D8 のステップに分けて説明されている．D1 は正規化で，被除数も除数も d 倍することで，除数の最上位の桁の最上位のビットが立つようにする．つまり，$a \div b$ の演算を $ad \div bd$ に置き換えるが，bd は 2^{32} 進数で最上位の桁の最上位のビットが 1 になるような d を用いる．プログラムでは d の代わりに変数 `f` に d のリーディング零のビット数を求めてシフトする．この処理を行うことで，仮商を得やすくする（後述）．

ldivide の 2，D1　正規化

```
    int f, k, m, n;
    m = a.l;
    n = b.l;
    if (qv.capacity < n+2) qv.realloc(n+2,0);
    assert(qv.capacity>=(n+2));
    if (q.capacity < (m-n+2)) q.realloc(m-n+2,0);
    q.l = m-n+1;                                    商の桁数は m-n+1
    if (u.capacity < u.l+2) u.realloc(u.l+2,0);
    u.d[m+1] = 0u;
    ++u.l;
    f = clzu(b.d[n]);                   // Count Leading Zero
    if (f != 0) { u <<= f; v <<= f; }   // u / 2^{f}
```

この時点で商の桁数は $m - n + 1$ 桁と決める．

ここからはほぼ Knuth の教科書の D2「j を初期設定」以降のとおりである．`while(k...)` のループで，商 q の `q[k+1]` 桁目を決める．

ldivide の 3（除算の本体のループ）

```
    k = m - n;
    while (k >= 0) {                                    // 商の k+1 桁目を求めるの反復
        uint32_t qh;
        uint32_t borrow, carry;                         // algorithm D3
        qh = estimate_quotient(&u.d[n+k-2+1],&v.d[n-1]);  // 仮商 \hat{q}
        if (qh != 0) {
            copy_digits(&qv.d[1],&v.d[1],n);
            qv.d[n+1] = 0u;
            mullu(&qv.d[1],n+1,qh);                       //  乗算と
            borrow = subll(&u.d[k+1],n+1,&qv.d[1],n+1);   //  減算， algorithm D4
            if (borrow != 0) {                      //  剰余のテスト， algorithm D5
                --qh;
                carry = addll(&u.d[k+1],n+1,&v.d[1],n);
                int addbacks = 0;
                while (carry == 0) {                      //  足し戻す， algorithm D6
                    --qh;
                    carry = addll(&u.d[k+1],n+1,&v.d[1],n);
                }
            }
        }
        q.d[k+1] = qh;                                    // 商の k+1 桁目
        k = k - 1;
    }
    if (r != 0) {
        u.slz();
        if (f != 0) u.shr(f);  // 非正規化（右シフト）， algorithm D8
        swap(u,*r);
    }
    q.slz();
    return q;
}
```

次に教科書から引用する文章は，一般的な問題として，次の単純なステップに分解されている．個々のステップは商を一桁だけ求める問題に書き直され，$n+1$ 桁の被除数 u を n 桁の除数 v で割る．

> まず次のように定義する．$u = (u_n u_{n-1} \cdots u_1 u_0)$ および $v = (v_{n-1} \cdots v_1 v_0)$ は，基数 b で表現した非負の整数で，$u/v < b$ である．$q = \lfloor u/v \rfloor$ を計算するアルゴリズムを求む．

教科書では $\hat{q}$ で表される仮商の予測値は，被除数の上 3 桁と，除数の上 2 桁から求める．

> まず次のように定義する．
>
> $$\hat{q} = \min\left(\left\lfloor \frac{u_n b + u_{n-1}}{v_{n-1}} \right\rfloor, b-1\right) \tag{2}$$
>
> この式は，$\hat{q}$ を u の先頭 2 桁を v の先頭 1 桁で割った値として，その値が b 以上なら $b-1$ とするという意味である．驚くべきことに，これから調べてみると，この値 $\hat{q}$ は，v_{n-1} が**十分大きければ**，常に求めたい q の値の非常によい近似になっている[15]．

これを estimate_quotient 関数で求める．u[2] が教科書では u_{j+n}，u[1] が u_{j+n-1}，u[0] が u_{j+n-2} に該当する．D3 で行うテストは $\hat{q}v_{n-2} > b\hat{r} + u_{j+n-2}$ だが，プログラムでは while 文の条件にある（2^{32} 進数なので b 倍を 32 ビットシフトで行う）．これにより,「仮商が 1 だけ大きすぎる場合のほとんどを見分けられ，2 だけ大きすぎる場合をすべて排除できる」と教科書に記載されているが，これが成立するのは正規化したからである．

[15]「十分大きければ」が，正規化する目的である．

estimate_quotient

```
uint32_t longint::estimate_quotient(uint32_t* u, uint32_t* v) {
    uint64_t t, q, r;

    if (u[2] == v[1]) {
        q = BASE - 1u;
    } else {
        t = (static_cast<uint64_t>(u[2]) << 32) + u[1];
        q = t/v[1];
    }
    r = t - q*v[1]; //r = t%v[1];
    while (r < BASE and q*v[0] > (r << 32) + u[0]) {
        q = q - 1;
        r = r + v[1];
    }
    return static_cast<uint32_t>(q);
}
```

上記の「ldivide の 3（除算の本体のループ）」で使用する 3 つの関数 mullu，subll，addll を示す．mullu は，longint 型の m 桁の被乗数（第 1 引数）に，uint32_t 型の乗数を掛け，積を第 1 引数に置き，キャリーを返す．前述した乗算と同様，32 ビット変数の被乗数を 64 ビットにキャストしてから乗数と掛けることで，32 ビットの整数乗算命令 1 回で処理する．

mullu

```
uint32_t longint::mullu(uint32_t* a, int m, uint32_t s) {
    uint32_t carry;
    int i;
    carry = 0u;
    for (i=0; i < m; ++i) {
        uint64_t t = static_cast<uint64_t>(a[i])*s + carry;
        carry = (t >> 32);
        a[i] = static_cast<uint32_t>(t);
    }
    return carry;
}
```

この関数だけを longintmullu.cpp ファイルに取り出して “g++ -S -O3 longintmullu.cpp -masm=intel” でコンパイルして longintmullu.s のループ反復部分を調べる[16]．32 ビットプロセッサの場合，乗算命令は mul だけで，32 ビットの右シフトは，上位 32 ビットを D レジスタをアクセスするだけになっている．

mullu の -S コンパイルによる生成されたコード

```
L3:
   mov   eax, edi
   xor   ebx, ebx
   mul   DWORD PTR [esi]        ! a[i] * s   結果は edx と eax
   add   eax, ecx               ! carry を加える
   adc   edx, ebx               ! add with carry で CF セットの場合はこれも加える
   add   esi, 4                 ! i のインクリメント
   mov   DWORD PTR [esi-4], eax ! a[i] = t
   cmp   esi, ebp               ! 比較の結果はフラグレジスタに格納される
   mov   ecx, edx               ! carry に edx を入れる（上位 32 ビット）
   jne   L3                     ! フラグレジスタの内容を参照して次の命令か L3 にジャンプ
```

優れたコード生成である．

64 ビットプロセッサの場合は 1 つの imul 命令で積が 64 ビット長のレジスタに得られる．

[16] “-masm=intel” はニーモニックを Intel 形式で表示する．これを指定しないと gcc や g++ は AT&T 形式で表示する．

subll は，longint 型の m 桁の数（第 1 引数）から，longint 型の n 桁の数を引き，差を第 1 引数に置き，バロー（0 か 1）を返す.

subll

```
uint32_t longint::subll(uint32_t* a, int m, uint32_t* b, int n){
    uint32_t borrow;
    int i;
    assert(m>=n);
    borrow = 0;
    for (i=0; i < n; ++i) {
        uint32_t t, u;
        t = a[i] - borrow;
        if (t > a[i]) { // underflow
            assert(a[i]==0u && borrow==1u);
            a[i] = t - b[i];
        } else {
            assert(borrow==0 || a[i]!=0);
            u = t - b[i];
            if (u > t) borrow = 1u; else borrow = 0u;
            a[i] = u;
        }
    }
    for (; i < m && borrow != 0u; ++i) {
        uint32_t v = a[i] - borrow;
        if (v > a[i]) borrow = 1u; else borrow = 0u;
        a[i] = v;
    }
    return borrow;
}
```

addll は longint 型の m 桁の数（第 1 引数）に，longint 型の n 桁の数を加え，和を第 1 引数に置き，キャリー（0 か 1）を返す.

addll

```
uint32_t longint::addll(uint32_t* a, int m, uint32_t* b, int n){
    uint32_t carry;
    int i;
    assert(m >= n);
    carry = 0;
    for (i=0; i < n; ++i) {
        uint32_t t, u;
        t = a[i] + carry;
        if (t < a[i]) { // overflow
            a[i] = t + b[i];
            carry = 1u;
        } else {
            assert(carry==0u || a[i]!=0xffffffff);
            u = t + b[i];
            if (u < t) carry = 1u; else carry = 0u;
            a[i] = u;
        }
    }
    for (; i < m && carry != 0; ++i) {
        uint32_t v = a[i] + carry;
        if (v < a[i]) carry = 1u; else carry = 0u;
        a[i] = v;
    }
    return carry;
}
```

除算で余りがない場合の演算，または余りを無視する場合は ldivide 呼び出しにせず，オペレータオーバーロードで記述できる[17].

余りを無視する除算のオペレータオーバーロード

```
longint longint::operator/(const longint& rhs) const {
    longint r;
    return ldivide(*this, rhs, &r);
}
```

2.8.4 rational クラスから呼ばれる加減算と乗算

前述したオペレータオーバーロードによる四則演算以外に，明示的に rational クラスから呼び出される四則演算などのルーチンがある．例えば，rational クラスの RatAdd 関数は，lcmp，ldivide，lmul，ladd，lsubv，lgcd2(lgcd) などを呼ぶ[18].

```
longint lmul(const longint& A, const longint& B);
longint ladd(const longint&, const longint&);
longint lsub(const longint&, const longint&);
int lsubv(const longint& A, const longint& B, longint& X, int id);
```

lmul は `X=A; x*=B; return X;`，ladd は `X=A; x+=B; return X;` のように実装されている．

2.9 関数

longint 型の引数をもつ min と max 関数，および pow 関数を用意した．

2.9.1 min, max 関数

2 つの longint 型の数の小さいほうを返す関数 `min` と，大きいほうを返す関数 `max` を friend 関数で用意した．プロトタイプ宣言を示す．

min 関数と max 関数のプロトタイプ宣言

```
friend longint const& min(longint const&, longint const&);
friend longint const& max(longint const&, longint const&);
friend longint pow(longint const&, int);
```

2.9.2 pow 関数

longint 型の数 x の n 乗 x^n を返す `pow` 関数を friend 関数で用意した．n を 2 進数表現したとき，最下位から k ビット目が 1 なら x^k を掛けることで x^n を求める．

[17] interval クラスの formatprn 関数などで使用している．

[18]「有理数計算プログラミング環境」は，C 言語で開発を始めたので，初期に開発されたままの部分は，演算子多重定義を使用していない．このため，有理数や多桁数を変数型としてコンパイラが認識しないので，明示的に関数呼出しで多桁演算をプログラミングしていた．

pow 関数

```
longint pow(longint const& x, int n) {// in case of  n=10=(1010)_2
   longint result = longint::ONE;
   longint p = x;
   while (n != 0) {                                  n =     1010   101   10   1
      if ((n & 1) != 0) {                                  ------------------------
         result *= p;                              result=             x^2      x^{10}
      }
      p *= p;                                          p  =     x^2   x^4   x^8
      n >>= 1;
   }
   return result;
}
```

コメントに $n = 10 = (1010)_2$ の場合を示したように，変数 p に $p = x, x^2, x^4, x^8$ を作ってゆき，n を 2 進数表現したとき最下位の桁が 1 になるときに p を掛ける．したがって，計算量のオーダーは $O(\log n)$ である[19]．

2.10 最大公約数

最大公約数（Greatest Common Divisor，GCD）を求める関数は多桁数の計算では使用しないが，有理算術演算で必須である．次章の rational クラスから呼び出される多桁数の四則演算と同列なので，longint.cpp に含めた．

「ユークリッドの互除法」は英語では Euclidian algorithm または Euclid's algorithm で,「の互除」はない．これはユークリッド幾何学は定規とコンパスで行い，除算は使わないからである．位取り記数法（零の発見）は 8 世紀ごろに発明されて，現在のような筆算による除算が始められたが，ユークリッド幾何学はそれよりもはるか古くから存在した．アルゴリズムの基本は "a が b の倍数のとき $\gcd(a, b) = b$" と "$\gcd(a, b) = \gcd(a - b, b)$" にあって，"$\gcd(a, b) = \gcd(b, r)$" は後者を繰り返すことで得られる．

2.10.1 lgcd 関数

「2 つの自然数 x, y, $x \geq y$ について，x の y による剰余を r とすると， x と y の最大公約数は y と r との最大公約数に等しい」という性質が成り立つ[20]．longint 型の引数に対して，GCD を返す関数を示す．

lgcd 関数

```
longint longint::lgcd(longint X, longint Y){
    longint Q,R; int i;
    i = lcmp(X,Y);
    if( i < 0 ) swap(X,Y);
    while(lcmp(Y,longint::ZERO) != 0){
        Q = ldivide(X, Y, &R);
        swap(X,Y); swap(Y,R);
    }
    return X;
}
```

[19] この関数は，interval.cpp で rational 型の数の 10 進変換 pow(longint::TEN,n) で使用している．p に x の 2 のべき乗を求めるところを定数に用意すればさらに高速化できるが，それは行っていない．

[20] $r = a\%b$; while$(r > 0)\{a = b;\ b = r;\ r = a\%b\}$

2.10.2 lgcd2 関数

ユークリッドの互除法に現れる除算を，減算とビットシフト演算に置き換える **2 進 gcd アルゴリズム**を示す [3, p. 317]．このアルゴリズムの考え方は，2 数 a と b の共通因子のうち偶数のものをシフト演算で先に分離しておき，分離された 2 数の奇数の最大公約数 c を，減算とシフト演算で求める $\gcd(a,b) = 2^k \cdot c$.

2 進 gcd アルゴリズム
$k = 0;\ u = a;\ v = b$
while(both(u and v)are even){
　　$k = k + 1$
　　$u = u \div 2; \quad v = v \div 2$
}
if(u is odd) $t = -v$ else $t = u$
while (t ! = 0) {
　　while(t is even) {
　　　　$t = t \div 2$
　　　　if ($t > 0$) $u = t$ else $v = -t$
　　}
　　$t = u - v$
}
$\gcd(a,b) = 2^k \cdot u$

はじめの while ループは，2 数 u と v がともに偶数の間，両者を 2 で割る操作（シフト演算）を繰返し，少なくとも一方が奇数になるようにする．これは，両者が偶数の場合は $\gcd(u,v) = 2 \cdot \gcd\left(\frac{u}{2}, \frac{v}{2}\right)$ に基づいている．このループを抜けた時点で，u と v のうち少なくとも一方は奇数である．u が奇数なら $-v$ を，偶数なら u を t に格納して，前処理を終え，後半は $\gcd(u,v)$ を求める（後半では共通因子は奇数）.

メインのループでは，$\gcd(u,v) = \gcd(u-v,v)$ に基づき，減算で 2 数を小さな数に置き換え，最大公約数の奇数因子を求める．これには $t = u - v$ として，$\max(u,v)$ を，$t > 0$ なら $u = t$ で，$t < 0$ なら $v = -t$ で置き換えて，u と v のうち大きいほうが $|t|$ に入るようにする．ここで t は偶数なら 2 で割るが，これは $\gcd(u,v) = \gcd\left(\frac{u}{2}, v\right)$ に基づいている．アルゴリズムは $|u-v| < \max(u,v)$ なので停止し，$\gcd(a,b) = 2^k \cdot u$ が得られる．

2 進 gcd アルゴリズムを用いると，演算量が桁数の 1 次オーダーである減算が主になるので，その反復回数倍に抑えられるので，除算を用いるユークリッドの互除法よりも，一般的には速い．

2 進 GCD アルゴリズムは，再帰呼び出しによる方法，反復による方法，トレーリングゼロのビット数を数える方法があるが，トレーリングゼロのビット数を数える方法を用いた．

lgcd2

```
longint longint::lgcd2(longint x, longint y){
    if(lcmp(x,longint::ZERO) == 0 ) {
        std::cout << "FATAL ERROR in lgcd2 x zero" << std::endl;
        assert(false);
    }
    assert(y != longint::ZERO);
    if(lcmp(y,longint::ZERO) == 0 ) {
        std::cout << "FATAL ERROR in lgcd2 y zero" << std::endl;
        assert(false);
    }
    uint32_t cf2 = std::min(x.ctz(),y.ctz());

    x.shr(x.ctz());

    for (;;) {
        y.shr(y.ctz());
        int s = lcmp(x,y);
        if (s == 0) break;
        if (s > 0) {    // swap
            swap(x,y);
        }
        if (lcmp(x,longint::ONE) == 0) break;
        y -= x;
    }
    x.shl((int)cf2);
    return x;
}
```

2.10.3 GCD 計算のスキップ

有理算術演算のたびに既約分数にするために分母・分子の GCD を求めているが，正則連分数アルゴリズムのように，計算される有理数が既約になっているものもある．また，表示の目的で基数変換するだけで，それを後続の計算で使用しない場合もあり，このような場合では既約化を省略したい．この目的で，つねに GCD = 1 を返す lgcd1 関数を設けた．

2.10.4 lgcd2，lgcd，lgcd1 関数の切り替え

2 つの GCD を求める関数を，実行時に切り替えて，速いほうを使用する目的で， GCD 関数へのポインタを public で longint.h に宣言し

longint.h の (*lgcd_p)2 の宣言

```
public:
    static longint (*lgcd_p)(longint, longint); // 関数へのポインタ
```

longint.cpp で次のように定義する．

longint.cpp の (*lgcd_p)2 の定義

```
longint (*longint::lgcd_p)(longint, longint) = longint::lgcd2;  // デフォールトは lgcd2
```

GCD の計算は，有理算術演算の計算時間の大半を使用することもあるカーネルだが，オペランドの有理数がどのような数であるかで，計算時間は大きく変化する．倍精度浮動小数点数を有理数変換した場合，

分母は 2 冪になっているので，2 進 gcd は多くの場合，除算による方法よりも速い．比較のために両者を計測する例題は **PiCfracAtanInt.cpp** に含まれる．

2.10.5 ユーティリティルーチン

多桁数のビット数などを返す関数は，longint クラスに属するが，使用方法は `longint::` を付けずに使用したいので，（メンバー関数ではなく）**フレンド関数**とした．

多桁数の桁数を返す関数 numdgts を示す．この関数の使用例は 98 ページに示す（有理数の分母と分子の桁数を調べる）．

桁数を返す numdgts

```
int numdgts(const longint& X){
    return X.l;
}
```

多桁数のビット数を返す関数 numbits を示す．

ビット数を返す numbits

```
uint32_t numbits(const longint& A){  // Number of Bits of longint number
   uint32_t dl=A.d[A.l];
   uint32_t nba=0;
   if(dl != 0){
      uint32_t nbdl=32-longint::clzu(dl);  // number of bits in A.d[A.l]
      nba=32*(numdgts(A)-1)+nbdl;          // number of bits of A
   }
   return nba;
}
```

2.11 プロファイラ

プロファイルの取得用の関数 `eprof` は longint.h に入れた．

longint.h の eprof クラス

```
class eprof {
   long long& acc_;
   long long t;
public:
   eprof(long long& a, long& cnt) : acc_(a) {
      t = gettime(); ++cnt;        現在時間を t に記憶し，カウンタをインクリメント
   }
   ~eprof() { acc_ += (gettime() - t); }      デストラクタ
};
```

プロファイラーの使用例は **dvsrch2.cpp, LDL.cpp, PiCfracAtanInt.cpp** に含まれる．

2 進法で GCD を求める `lgcd2` 関数の場合は `EPROF2` を #define してある．Makefile に EPROF2 を指定し，コンパイル時に “-DEPROF2” を指定する．関数の入口でタイマーがセットされ，制御が出るところで `lgcd2time` に時間が加算される．

```
longint longint::lgcd2(longint x, longint y){
#ifdef EPROF2
   eprof e(longint::lgcd2time,longint::lgcd2count);  関数の先頭でタイマーをセット
#endif
```

変数 `lgcd2time` などは， `EPROF2` の #define と連動して定義してある．使用例は saehfv3.cpp にある．

2.12 入出力ストリーム

<< は左シフト演算子だが，C++ の cout クラスはこの演算子を出力演算子として多重定義している．ここではこれを longint 型の数の出力演算子として定義する．プロトタイプ宣言は longint.h で行う．

longint.h の << 演算子

```
friend std::ostream& operator <<(std::ostream&, const longint&);
```

実装は longint.cpp で 2^{32} 進数を 10 進数に変換した文字列を `string s` に格納してから，出力ストリームに出力する．

longint.cpp の出力ストリーム << 演算子

```
std::ostream& operator <<(std::ostream& stream, const longint& A) {
   longint tmp = A;
   std::string s;
   uint32_t r;
   uint64_t t, q;
   if (tmp.l==0) {
      s += "0";
   } else {
      while (tmp.l != 0) {
         r = 0llu;
         for (int i=tmp.l; i >= 1; --i) {
            t = (uint64_t)r*longint::BASE + tmp.d[i];
            r = t % 10llu;
            q = t / 10llu;
            tmp.d[i] = q;
            if (tmp.d[tmp.l] == 0lu) --tmp.l;
         }
         std::stringstream out;
         out << r;
         s += out.str();
      }
   }
   stream << std::string(s.rbegin(),s.rend());
   return stream;
}
```

2.13 例題：総和計算（多重精度演算との比較）

多桁数 longint は負の整数を扱えない．実際に負数を扱う整数演算を行う場合は，有理数を使用して，分母が 1 の有理算術演算で行う．ここでは負数の現れない例題を示す．

1 から n までの「自然数の 7 乗の総和」を求める例題 powersum.cpp を示す．公式を示す．

$$\sum_{r=1}^{n} r^7 = \frac{1}{24}n^2(n+1)^2(3n^4+6n^3-n^2-4n+2) \tag{3}$$
$$= \frac{1}{8}n^2(n+1)^2(n^4+2n^3-\frac{n^2}{3}-\frac{4n}{3}+\frac{2}{3})$$

総和を実際に計算して，答を公式と比較する．

2.13.1 longint の使用例

プログラムを示す．

```
//  g++ -O3 powersum.cpp  longint.o mempool.o gettimeofday.o
#include <iostream>
#include "longint.h"
using namespace std;
int main(void){
    longint x1,x2,x3,x4,x7,z;
    int i,n;
    cout << "Enter n" << endl;
    cin >> n;
    longint x("0");
    longint y("0");
    for(i=1; i<=n; i++){
        x += longint::ONE;
        x2=x*x;
        x3=x*x2;
        x4=x2*x2;
        x7=x3*x4;
        y=y+x7;
    }
    cout << "SUM_{i=1}^" << n << " (i^7)=" << y << endl;

    uint32_t p = n;
    x = p;
    x1 = x+longint::ONE;
    x2 = x*x;
    x4 = x2*x2;
    z  = longint::THREE * x4 + longint::SIX * x2 * x - x2 - longint::FOUR * x + longint::TWO;
    z  = (x2 * x1 * x1 * z )/( longint::SIX * longint::FOUR);
    if(y != z){
        cout << " Result BAD " << z << endl;
    } else {
        cout << " Result OK " << endl;
    }
    return 1;
}
```

2.13.2 MPFUN の使用例

この問題を，David Bailey の MPFUN の多倍精度演算で行う場合と比較する．

MPFUN パッケージは 87 サブプログラムからなる，約 10,000 行の Fortran 77 プログラムによって構成される．3 種類のカスタムデータ型をもつ．

- 多倍長精度数（実数を近似），MP 数
- 多倍長精度数による複素数（実数を近似），MPC 数
- 指数拡張した倍精度，DPE 数

MP 数は単精度浮動小数点数の配列に格納される．MP 数 A は，基数 $r = 2^{24} = 16,777,216$ の r 進数として，次の形式で表現される．

$$A = \pm r^e (d_0.d_{-1}d_{-2}\cdots d_{-m+2}d_{-m+1}d_{-m})_r = \pm \left(\sum_{i=-m}^{n} d_i r^i \right) r^e \tag{4}$$

$$d_i : \text{digit in } r^i\text{position},\ 0 \le d_i \le r-1$$

配列の 1 ワード目に仮数部の長さ $m+1$ と符号 $\pm$（正なら $+$，負なら $-$，2 ワード目に指数部 e，3 ワード目以降に仮数部 d_i を格納する．多倍長数は，単精度浮動小数点数の配列の 1 ワード目に，仮数部の長さと符号，2 ワード目に指数部，3 ワード目以降に仮数部を格納する．四則演算は mpadd などのサブルーチン呼び出しで行う．

```
      program pwrsum
      character*1 c
      parameter (mx=6, nx=2**mx, ns=9.5*nx+47, nd=7.225*nx)
      dimension c(nd+50),a(nx+4),x(nx+4),y(nx+4),z(nx+4)
      dimension x2(nd+50),x3(nx+4),x4(nx+4),x7(nx+4)
      dimension t(nd+50),t1(nx+4)
      dimension zero(2),one(3),two(3),three(3),four(3),six(3),f24(3)
      dimension f1(5),f2(5)
      common /mpcom1/ nw, idb, ldb, ier, mcr, ird, ics, ihs, ims
      common /mpcom3/ s(ns)
      common /mpcom4/ d(12*nx+6)
      common /mpcom5/ u(8*nx)
      nw=nx
      ims=ns
      call mpinix(mx)
c -----
      zero(1) = 0.;  zero(2) = 0.
      one(1) = 1.; one(2) = 0.; one(3) = 1.
      two(1) = 1.; two(2) = 0.; two(3) = 2.
      three(1) = 1.; three(2) = 0.; three(3) = 3.
      four(1) = 1.; four(2) = 0.; four(3) = 4.
      six(1) = 1.; six(2) = 0.; six(3) = 6.
      f24(1) = 1.; f24(2) = 0.; f24(3) = 24.
c -----
      f1(1) = 1.; f1(2) = 0.; f1(3) = 1.
      x(1) = 0.; x(2) = 0.
      y(1) = 0.; y(2) = 0.
c -----
      write(*,*) 'n'
      read(*,*) n
      do i=1,n
        call mpadd (x, one, x)
        call mpmul (x,  x, x2)
        call mpmul (x,  x2, x3)
        call mpmul (x2, x2, x4)
        call mpmul (x3, x4, x7)
        call mpadd (x7, y, y)
      enddo
      call mpout(6,y,nd,c)
c -----
      call mpadd (x, one, y)    ! n+1
      call mpmul (y, y, z)      ! (n+1)*(n+1)
      call mpmul (x2, z, z)     ! n*n*(n+1)*(n+1)
      call mpmul (three, x4, t) ! 3*n**4
      call mpmul (six, x3, t1)  ! 6*n**3
      call mpadd (t1, t, t)     ! 3*n**4+6*n**3
      call mpsub (t, x2, t)     ! 3*n**4+6*n**3-x**2
      call mpmul (four, x, t1)  ! 4*n
      call mpsub (t, t1, t)     ! 3*n**4+6*n**3-x**2-4*n
      call mpadd (t, two, t)    ! 3*n**4+6*n**3-x**2-4*n+2
      call mpmul (z, t, t)      ! n*n*(n+1)*(n+1)(3*n**4+6*n**3-x**2-4*n+2)
      call mpdiv (t,f24, t)  ! n*n*(n+1)*(n+1)(3*n**4+6*n**3-x**2-4*n+2)/24
      call mpout(6,t,nd,c)
      stop
```

表 1: r^7 の総和

n	$\sum_{r=1}^{n} r^7$	number of '6' in $6\cdots675$
1,000	$1.25\cdots\times10^{23}$	3
10,000	$1.25\cdots\times10^{31}$	5
100,000	$1.25\cdots\times10^{39}$	7
1,000,000	$1.25\cdots\times10^{47}$	9
10,000,000	$1.25\cdots\times10^{55}$	11
100,000,000	$1.25\cdots\times10^{63}$	13
1,000,000,000	$1.25\cdots\times10^{71}$	15

```
        end
```

オペレータオーバーロードが，分かりやすいプログラムを実現する．ただし，MPFUN は，四則演算を $+,-,*,/$ で記述して，これをプリプロセッサにより演算サブルーチン呼び出しのコードに変換する機能もサポートしている．

2.13.3 多桁計算と多重精度算術演算の違い

$n = 10,000$ を与えれば，$1.25\underbrace{0}_{1}5\underbrace{000}_{3}58\underbrace{333333}_{6}041\underbrace{66666}_{5}75\times10^{31}$ が表示される[21]．n として 10 のべきを与えると，公式 (3) による計算結果は，規則的な表示をとり，アンダーブレースした 0，0，3，6 が桁数を増加させていく．n が 10 億では 10 の 71 乗の数値になる（表 1）．公式が n の 8 乗なので n が 10 倍になると 8 桁増加する．

この計算は，結果が自然数になるが，多重精度算術演算では桁数が不足すると正しい結果は得られない．パラメータ mx= 3 では n が 10 億は正しく計算されず，下位の桁は公式による総和が $\cdots66666512$ となる．パラメータ mx= 2 では n が 1000 万も正しく計算されず，下位の桁は公式による総和が $\cdots33333351$ となり，総和を 1 からループ計算するとこれとは異なった値になるので，計算結果が丸められていることは判別できる．

[21] $n^8 \div 8$ に近い値になる．

3 rational クラス，rough クラス，interval クラス

有理数 R は，多桁数 N と D と符号 s で定義される．

$$R = s \times \frac{N}{D} \tag{5}$$

$s = 0$ の場合 $R = 0$ である．

rational 型の有理数は 2 つの longint 型の多桁数と符号で定義される．

```
class rational {
public:
   longint N, D;
   int s;
```

N は分子，D は分母，s は符号を表し，0 か 1 か -1 が入る[22]．

桁溢れしない倍精度浮動小数点数として表示などで利用するために，有理数の有効桁を 53 ビット取得して，倍精度浮動小数点数と整数による指数部で扱う rough 型の変数も設けた．

数値計算が有理数のみを扱う計算式を計算している間は，有理算術演算は正確な結果を与えるが，無理数を正確に計算することはできない．「有理数計算プログラミング環境」では，2 つの有理数で上限と下限を抑える interval 型の変数を設けて，無理数を挟む区間を縮小反復することで，無理数を必要な精度で挟み込むことができるようにした．

本章でははじめに rational 型について解説し，次に rough 型と interval 型について解説する．

3.1 有理数型変数の定義

本節でははじめに，有理数型変数に対するコンストラクタ，デストラクタ，再定義される演算子，メンバー関数のメニュー，定数の定義を紹介する．有理数型の変数を要素にもつ行列とベクトルを扱うので，これについて触れた後，四則演算，入出力演算子，ユーティリティルーチンを説明する．

コンストラクタとデストラクタを示す．

```
rational::rational() : N(0u), D(1u), s(0) { }

rational::rational(int32_t val) {
   this->D = (uint32_t)1U;
   if (val > 0) {
      this->N = (uint32_t) val; this->s=1;
   } else if (val == 0) {
      this->N = 0U; this->s=0;
   } else {
      this->N = (uint32_t)(-val); this->s=-1;
   }
}

static int32_t lgcd32(int32_t m, int32_t n);

rational::rational(int32_t n, int32_t d) {
   if (n == 0) {
      this->s = 0; this->N = longint::ZERO; this->D = longint::ONE;
   } else {
   int sign = 1;
```

[22]「有理数計算プログラミング環境」は，C 言語で開発を始めたので，有理算術演算を，演算子多重定義を使用せずに明示的に四則演算の関数呼出しで行っていた．倍精度浮動小数点数の有理数変換は Rdset，四則演算は RatAdd，RatSub，RatMul，RatDiv で行っていた．後に演算子 $+$，$-$，$*$，/ を使用して有理算術演算をできるように，演算子多重定義を加えた．

```
    if (n < 0) {
       sign = -sign; n = -n;
    }
    if (d < 0) {
       sign = -sign; d = -d;
    }
    int32_t g = lgcd32(n,d);
    this->s = sign;
    this->N = longint((uint32_t)(n/g)); this->D = longint((uint32_t)(d/g));
    }
}

rational::rational(const longint& n, const longint& d) {
    if (n == longint::ZERO) {
       this->s = 0; this->N = longint::ZERO; this->D = longint::ONE;
    } else {
       longint g = longint::lgcd_p(n,d);                    GCD を求めて
       this->s = 1; this->N = n/g; this->D = d/g;           約分する
    }
}

rational::rational(const rational& rhs) : N(rhs.N), D(rhs.D), s(rhs.s) { }

rational::~rational() { }
```

使用例を示す.

```
longint n("123456789012345678901234567890");
longint d("1000000000");
rational x(n,d);
```

この例は2つの longint 型の引数なので，上述した longint 型の引数をとるコンストラクタが動き，rational 型の変数 x に有理数が格納される．後述するユーティリティ関数の RRset は，機能は同じであるが，分母・分子の GCD を求めて約分する操作を行わない．後述する，ユーティリティ関数の format は RRset を使用している．

オペレータオーバーロードされる演算子を示す.

```
    rational& operator=(const rational&);
    rational& operator=(uint32_t);
    rational& operator=(int32_t);
    rational& operator+=(const rational&);
    rational& operator-=(const rational&);
    rational& operator*=(const rational&);
    rational operator+(const rational&) const;   // RatAdd
    rational operator-(const rational&) const;   // RatSub
    rational operator*(const rational&) const;   // RatMul
    rational operator/(const rational&) const;   // RatDiv
    rational operator-() const;                 // unary minus
    bool operator==(const rational&) const;
    bool operator!=(const rational&) const;
    bool operator<(const rational&) const;
    bool operator>(const rational&) const;
    bool operator<=(const rational&) const;
    bool operator>=(const rational&) const;
    operator double() const;
```

メンバー関数を示す．swap 関数は rational.h の中でインライン関数として埋め込んでいる．

```
    longint numerator() const { return N; }          分子を返す
    longint denominator() const { return D; }        分母を返す
    rational abs() const;                            絶対値
    rational inv() const;                            逆数
    rational max(const rational&) const;             最大
    rational min(const rational&) const;             最小
    rational floor() const;                          floor
    rational ceil() const;                           ceil
    void set(const longint& n, const longint& d);
    void sset(int sign, const longint& n, const longint& d);
    int sign() { return s; }
    std::string format(int w, int d) const;
    static void swap(rational& x, rational& y) {       有理数のスワップ
       std::swap(x.s,y.s); longint::swap(x.N,y.N); longint::swap(x.D,y.D);
    }
```

friend 関数を示す.

```
    friend rational const& min(rational const&, rational const&);
    friend rational const& max(rational const&, rational const&);
    friend iarchive& operator>>(iarchive& ia, rational& r);
    friend oarchive& operator<<(oarchive& oa, rational const& r);
```

そのほかの関数を示す.

```
rational Rlset(int32_t);
rational Rfset(float*);
rational Rdset(double*);
rational RRset(longint, longint);
rational RatAbs(const rational&);
rational RatAdd(const rational&, const rational&);
rational RatSub(const rational&, const rational&);
rational RatMul(const rational&, const rational&);
rational RatDiv(const rational&, const rational&);
void RatRed2(rational&);
int RatCmp(const rational&, const rational&);
void RatPrt(char*, rational);
int ndd(int n);
double ndpw2(const longint& a, int& pw2);
double ndpw2(const rational& a, int& pw2);
double roughdec(const double& f, const int& n, int& m);
std::istream& operator>>(std::istream&, rational&);
std::ostream& operator<<(std::ostream&, const rational&);
```

有理数の定数を rational.cpp でいくつか定義している.

```
const rational rational::ZERO((int32_t) 0);
const rational rational::ONE((int32_t) 1);
const rational rational::TWO((int32_t) 2);
const rational rational::FOUR((int32_t) 4);
const rational rational::FIVE((int32_t) 5);
const rational rational::EIGHT((int32_t) 8);
const rational rational::TEN((int32_t) 10);
const rational rational::SIXTEEN((int32_t) 16);
const rational rational::KILO((int32_t) 1000);
const rational rational::KIL((int32_t) 1024);
const rational rational::MILLION((int32_t) 1000000);
const rational rational::MEGA((int32_t) 1048576);
```

3.2 有理数の四則演算

有理数の四則演算は次式で行える．

$$r_1 \pm r_2 = \frac{b}{a} \pm \frac{d}{c} = \frac{bc \pm ad}{ac} \tag{6}$$

$$r_1 \times r_2 = \frac{b}{a} \times \frac{d}{c} = \frac{bd}{ac} \tag{7}$$

$$r_1 \div r_2 = \frac{b}{a} \div \frac{d}{c} = \frac{bc}{ad} \tag{8}$$

ここに r_1, r_2 は有理数で rational 型，a, b, c, d は多桁数で longint 型である．

有理算術演算ルーチンは，rational.h と rational.cpp に実装されている．この実行文に関係するものを示す．

```
rational& rational::operator=(const rational& rhs) {
    this->N = rhs.N;
    this->D = rhs.D;
    this->s = rhs.s;
    return *this;
}

rational& rational::operator+=(const rational& rhs) {
    (*this) = RatAdd(*this,rhs);
    return *this;
}

rational rational::operator+(const rational& rhs) const {
    return RatAdd(*this,rhs);
}

rational rational::operator*(const rational& rhs) const {
    rational result;
    result = RatMul(*this,rhs);
    return result;
}
```

3.2.1 乗算

演算子 * は RatMul 関数による．素朴な乗算のほかに，RAT_REDUCE により，桁数を抑えた方法を選べるようにした．

```
rational RatMul(const rational& X, const rational& Y){
   rational Z;
   if( X.s * Y.s == 0){
      Z = Rlset(0);
   } else {
#ifdef RAT_REDUCE
   longint E, R;
   rational XX, YY;
      XX = X; YY = Y;
      E = longint::lgcd2(XX.N,YY.D);
      if (longint::lcmp(E,longint::ONE) != 0) {
         XX.N = longint::ldivide(XX.N,E,0);
         YY.D = longint::ldivide(YY.D,E,0);
      }
      E = longint::lgcd2(YY.N,XX.D);
```

```
        if (longint::lcmp(E,longint::ONE) != 0) {
           XX.D = longint::ldivide(XX.D,E,0);
           YY.N = longint::ldivide(YY.N,E,0);
        }
        Z.N = longint::lmul(XX.N, YY.N);
        Z.D = longint::lmul(XX.D, YY.D);
        Z.s = X.s * Y.s;
#else
        Z.N = longint::lmul(X.N, Y.N);
        Z.D = longint::lmul(X.D, Y.D);
        Z.s = X.s * Y.s;
        RatRed2(Z);
#endif
    }
    return Z;
}
```

lmul は longint.cpp にある．

`#ifdef _RAT_REDUCE` で切り替えるコードは，被乗数の分子と乗数の分母，または被乗数の分母と乗数の分子に共通因子があると，中間に現れる数の桁を小さく抑える．

$e = \gcd(b, c)$
if$(e > 1)$\{
  $b = b \div e$; $c = c \div e$
\}
$e = \gcd(a, d)$
if$(e > 1)$\{
  $a = a \div e$; $d = d \div e$
\}
Num= bd ; Den= ac

$\frac{15}{7} \times \frac{28}{9}$ の場合，$\frac{5}{1} \times \frac{4}{3}$ を計算することになるので，素朴な計算が約分しなくてはならない $\frac{420}{63}$ を経由しない．ただし，これを指定すると**遅くなる場合が多かった**ので，指定していない．

RatRed2 は分母と分子の GCD を求めて有理数を既約とする．

```
void RatRed2(rational& A){   /* in-place reduce numerator and denominator */
    longint LONE, R, Z;
    if(A.s != 0) {
        Z = (*longint::lgcd_p)(A.N,A.D);
        if( longint::longint::lcmp(Z,longint::ONE) != 0 ) {   // GCD is not one
            A.N = longint::ldivide(A.N,Z,0);
            A.D = longint::ldivide(A.D,Z,0);
        }
    }
}
```

3.2.2 除算

RatDiv で行うが，乗算の乗数の分母と分子を入れ替えたものなので，省略する．

3.2.3 加減算

素朴な加算 $\frac{b}{a} \pm \frac{d}{c} = \frac{bc \pm ad}{ac}$ の桁数を抑えることができる．これは，`RAT_REDADD` により選択する．これは効果があるので，素朴な加算は省略して，桁数を抑える方法だけを示す [3, p. 310].

$e = \gcd(a, c)$
if($e > 1$){
　$t = b \cdot (c \div e) \pm d \cdot (a \div e)$
　$f = \gcd(t, e)$
　Num= $t \div f$
　Den= $(a \div e) \cdot (c \div f)$
} else {
　Num= $bc \pm ad$
　Den= ac
}

$\frac{7}{66}+\frac{17}{12}$ の場合，gcd(66,12) は 6 で，$c \div e$ は $22 \div 6 = 2$，$a \div e$ は $66 \div 6 = 11$ になるから，$t = 7 \cdot 2 + 17 \cdot 11 = 201$ になる．gcd(201,6)=3 なので，分子 Num は $201 \div 3 = 67$，分母 Den は $(66 \div 3) \times (12 \div 3) = 11 \times 4 = 44$ が得られる．最終的な分母 Den と分子 Num は互いに素である[23]．

加算 RatAdd を示す．

```
rational RatAdd(const rational& X, const rational& Y){
   rational Z;
   longint XY,YX,XD,YD,E,F,G,T,R;
   int t,g1;

   if( X.s * Y.s == 0){
      if( X.s == 0){
         Z=Y;                    /*  X=0,(Y=0,Y != 0)  */
      } else {
         Z=X;                    /*  (X=0,X != 0),Y=0, */
      }
   } else {
      G = (*longint::lgcd_p)(X.D, Y.D);
      g1 = longint::longint::lcmp(G, longint::ONE);
      if( g1 > 0){
         XD = longint::ldivide(X.D, G, &R); YD = longint::ldivide(Y.D, G, &R);
         XY = longint::lmul(X.N, YD);   YX = longint::lmul(Y.N, XD);
      } else {
         XY = longint::lmul(X.N, Y.D);  YX = longint::lmul(Y.N, X.D);
      }

      if( X.s * Y.s == 1){            /* X,Y same sign   */
         if( g1 > 0){
            T=longint::ladd(XY,YX); E=(*longint::lgcd_p)(T,G);
            Z.N=longint::ldivide(T,E,&R);
            F=longint::ldivide(Y.D,E,&R);
            Z.D=XD*F;
         } else {
            Z.N = longint::ladd(XY,YX); Z.D = X.D*Y.D;
         }
         if( X.s == 1) Z.s=1; else Z.s=-1;
      } else {              /* X,Y different sign OR result=0 */
         if( g1 > 0){
            t=longint::lsubv(XY,YX,T,2);
            if( t != 0) {
               if( X.s == 1) Z.s= t; else Z.s=-t;
               E=(*longint::lgcd_p)(T,G); Z.N=longint::ldivide(T,E,&R);
               F=longint::ldivide(Y.D,E,&R); Z.D=XD*F;
            } else {
               Z.N = longint::lset(0L); Z.D = longint::lset(1L); Z.s = 0;
```

[23] $a = a' \cdot e$，$c = c' \cdot e$ とおく（$t = bc' + da'$）．p が a' を割り切る素数なら，p は b も c' も割り切らないので，p は $bc' + a'd$ を割り切らない．したがって $a'c'$ のどの素約数も，与えられた gcd に影響せず，$\gcd(bc' + a'd, e) = \gcd(bc' + a'd, ea'c')$ が成立する．

```
            }
         } else {
            t = longint::lsubv(XY,YX,Z.N,3);
            if( t != 0) {
               if( X.s == 1) Z.s= t; else Z.s=-t;
               Z.D=X.D*Y.D;
            } else {
               Z.N = longint::lset(0L); Z.D = longint::lset(1L); Z.s = 0;
            }
         }
      }   /******* X.s * Y.s == 1   ***********/
      if( g1 == 0) RatRed2(Z);
   }    /****  X.s * Y.s == 0   ***********/
   return Z;
}
```

減算 RatSub は加数の符号を反転させて加算するので，省略する．

3.2.4 入出力演算子

std::istream クラスの演算子 >> を多重定義して rational 型の変数を使用可能にする．

```
std::istream& operator>>(std::istream& is, rational& rhs) {
   char c;
   longint inum;
   int iexp;
   inum = (uint32_t)0u; iexp = 0;  // inum*10^iexp
   is.get(c);
   while (c == ' ') {   // BUG: should handle tab, as well
      is.get(c);
   }
   if (c >= '0' && c <= '9') {
      inum = inum*longint::TEN + longint((uint32_t)(c - '0'));
      is.get(c);
      while (c >= '0' && c <= '9') {
         inum = inum*longint::TEN + longint((uint32_t)(c - '0'));
         is.get(c);
      }
      if (c == '.') {
         is.get(c);
         if (c >= '0' && c <= '9') {
            inum = inum*longint::TEN + longint((uint32_t)(c - '0'));
            iexp += 1;
            is.get(c);
            while (c >= '0' && c <='9') {
               inum = inum*longint::TEN + longint((uint32_t)(c - '0'));
               iexp += 1;
               is.get(c);
            }
         }
      }
   }
   is.putback(c);
   rhs.s = 1;
   rhs.N = inum;
   rhs.D = (uint32_t)1u; while (iexp-- > 0) rhs.D = rhs.D*longint::TEN;
   RatRed2(rhs);
   return is;
```

```
}
```

同様に std::ostream クラスの演算子 << も多重定義する．

```
std::ostream& operator<<(std::ostream& stream, const rational& rhs) {
    if (rhs.s < 0) stream << "-";
    stream << rhs.N << "/" << rhs.D;
    return stream;
}
```

3.3 有理算術演算用ユーティリティルーチン

有理数プログラミングで用いる関数のうち，使用頻度の高いものを説明する．浮動小数点数の有理数への変換，整数への floor と ceil，絶対値，ベクトルの桁数削減のためのスケール関数，数値の表示関数，多項式演算関数，チェックポイントとリスタート機能，行列の表示関数，行列の生成関数などである．

3.3.1 有理数を定義する関数

倍精度浮動小数点数を有理数変換する Rdset で行う．32 ビット符号なし整数は Rlset， 32 ビット符号なし整数を分母と分子で指定する場合は RRset で定義することもできる（コンストラクタでも可能だが，明示的に書きたい場合はこれらの関数を使用することもできる）．後述するが（53 ページの「有理数表示の書式関数」），RRset で定義すると，GCD 計算を行わないので，表示目的だけの format 関数ではこちらを使用している．

浮動小数点数の有理数変換ルーチン　IEEE 倍精度浮動小数点数を rational 型に変換して格納する．規格では NaN や Inf を含むが，これらが来た場合にはエラーとして終了している．なお，DEN は通常の有理数に変換される．

```
rational Rdset(const double *x){
   typedef  union{
      double r; uint32_t irr[2];
   } equivalenced;
   equivalenced Dfloat;

   uint32_t ir[2], mant;
   int iexp, i, maxe=1023, mine=-1022, mask=0x000fffff, ibias=1023;
   uint64_t n;
   longint T; rational X;

   Dfloat.r=*x;
   ir[0] = Dfloat.irr[1]; ir[1] = Dfloat.irr[0];

   iexp = ((ir[0] & 0x7ff00000) >> 20) - ibias;
   mant = ( ir[0] & mask ) + 0x00100000;
   if(iexp==(maxe+1)){
      std::cout << "iexp=" << iexp << "=NaN or Inf, *x=" << *x << " stop by exit(3)" << std::endl;
      printf("ir[0]=%x ir[1]=%x\n",ir[0],ir[1]);
      exit(3);
   }
   if(iexp==(mine-1)){
      mant = ir[0] & mask;     /*  DEN  */
      iexp = iexp+1;           /*  DEN  */
   }
```

```
    n = (uint64_t)mant*65536*65536 + ir[1];
    if(*x > 0){
       X.N = longint::llset(n); X.s=1;
    } else if(*x == 0){
       X.N = longint::llset(0L); X.s=0;
    } else {
       X.N = longint::llset(n); X.s=-1;
    }
    T = longint::lset(1U);
    if(*x != 0){
       X.D = T;
       if(iexp > 52){
          T = X.N;
          for(i=1; i<=iexp-52; i++){
             X.N = longint::smul(T,2L); T=X.N;
          }
       }else{
          for(i=1; i<=52-iexp; i++){
             X.D = longint::smul(T,2L); T=X.D;
          }
       }
    }
    RatRed2(X);
    return X;
}
```

単精度数は，倍精度変換してから Rdset を呼出す Rfset を用意した．また，4 倍精度（倍々精度）は，実装が上位と下位の 2 つの倍精度浮動小数点数で実装されている場合は，それぞれを有理数変換の後，両者を加えればよい（GNU の g++ コンパイラが 4 倍精度をサポートしないため，この関数はない）．

3.3.2 floor，ceil，abs，min，max

もっとも近い整数（$-\infty$ 側）に丸める関数 floor を示す．

```
rational rational::floor() const {
   rational res = *this;
   if (res.s == 0) {
      res.N = longint::ZERO;
      res.D = longint::ONE;
   } else if (res.s > 0) {
      res.s = 1;
      res.N = this->numerator()/this->denominator();
      res.D = longint::ONE;
   } else {
      res.s = -1;
      res.N = (this->numerator()+this->denominator()-longint::ONE)/this->denominator();
      res.D = longint::ONE;
   }
   if (res.N == longint::ZERO) { res.s = 0;}
   return res;
}
```

式 (5) の符号 $s = 1$ で正の有理数の場合は，ldivide で $N \div D$ の商を返し， $s = -1$ で負の有理数の場合は，ldivide で $(N + D - 1) \div D$ の商に符号 $s = -1$ を付けて返し，$s = 0$ の場合は零を返す．

もっとも近い整数（$+\infty$ 側）に切上げる関数 ceil を示す．

```
rational rational::ceil() const {
```

```
    rational res = *this;
    if (res.s == 0) {
       res.N = longint::ZERO;
       res.D = longint::ONE;
    } else if (res.s > 0) {
       res.s = 1;
       res.N = (this->numerator()+this->denominator()-longint::ONE)/this->denominator();
       res.D = longint::ONE;
    } else {
       res.s = -1;
       res.N = this->numerator()/this->denominator();
       res.D = longint::ONE;
    }
    if (res.N == longint::ZERO) { res.s = 0;}
    return res;
}
```

式 (5) の符号 $s = 1$ で正の有理数の場合は，ldivide で $(N + D - 1) \div D$ の商を返し， $s = -1$ で負の有理数の場合は，ldivide で $N \div D$ の商に符号 $s = -1$ を付けて返し，$s = 0$ の場合は零を返す．

このほか，絶対値をとるメンバー関数 abs，逆数をとるメンバー関数 inv，大小比較して大を返す max と小を返す min がある．メンバー関数の場合は big=a.max(b) のように使うので big=max(a,b) のように使える関数を friend 関数としても用意した．

```
rational rational::max(const rational& val) const {
   rational res;
   if (*this > val) {
      res = *this;
   } else {
      res = val;
   }
   return res;
}
rational const& max(rational const& x, rational const& y)
{
   if (x > y) return x; else return y;
}
```

3.3.3 有理数表示の書式関数

有理数を 10 進数変換して，途中を飛ばして表示するメンバ関数 format(w,d) を用意した．w は小数点より上の桁に使用する幅（width）で，小数点以下は d 桁表示する．

10 進変換は処理時間がかかる．rational 型の変数を使用して m*=rational::TEN とすると，GCD を求めて約分する処理が入り遅くなるので，longint 型で 10 倍している．また，GCD 計算を避けるために，コンストラクタを使用せずに RRset を使用した．

次の使用例は，rational 型の変数 p に格納されている数値を，有理数，倍精度浮動小数点数，setprecision(16) 指定の倍精度浮動小数点数，小数点以下 12 桁，11 桁，10 桁に四捨五入，の 6 通りで表示するプログラムである．

rational::format 関数の使用例

```
std::cout << " p=" << p << std::endl;
std::cout << " (double)p=" << (double)p << std::endl;
std::cout << " (double)p=" << std::setprecision(16) << (double)p << std::endl;
std::cout << p.format(42,12) << std::endl;
std::cout << p.format(41,11) << std::endl;
std::cout << p.format(40,10) << std::endl;
```

有理数（分数）は分母・分子を正確に表示されても大小関係は分らないので，デバッグなどでは double 型に変換して表示することが多い．表示桁数を指定しないと 6 桁程度しか見られないので，倍精度の桁数を広げて表示することも多い[24]．この方式では絶対値が倍精度数のレンジ 4.941×10^{-324}（DEN を含む）から 1.798×10^{308} の範囲内に収まる数であり，桁数は 17 桁程度まででなければ見ることができない．絶対値がこの範囲を外れた場合や，桁数を多く確かめたいときは，倍精度浮動小数点数に変換したのでは役に立たない．このような場合に，文字列を返す関数 format(w,d) を使用することで，必要な桁を見ることができる．上記のプログラムの出力を示す．

```
 p=215034400000000000000000000000000000000000000/11651357
 (double)p=1.84557e+35
 (double)p=1.845573867490284e+35
18455738 (20 digits) 44398373.511342927695
18455738 (20 digits) 44398373.51134292770
18455738 (20 digits) 44398373.5113429277
```

format 関数は，小数点よりも上の桁数が多い数に対し，最初の数桁と，小数点の上の数桁，小数点以下の指定された桁を，文字列（string 型）として返す．指定する 2 つの引数は，表示の幅 w と小数部の桁数 d である．

format 関数の前半の，小数点以下 $d+1$ 桁目を四捨五入して変数 val に得るところまでを示す．文字列ストリーム stringstream 型はシフト演算子 << で longint 型の数を 10 進文字列に変換するために用い，文字列 string 型の使用は，length() 関数で文字数を得るために使用している．ss はいったん文字列ストリームに 10 進数を文字列に変換して格納し，指定された幅（桁の位置）に合わせて空白を詰めて res に出力するために使用している．

```
std::string rational::format(int w, int d) const{
   std::stringstream res, ss;      文字列ストリーム RESult, StringStream
   rational val = *this;           val に数値を格納
   longint q, r;
   int wi = w-d-1;                 小数点より上に使える桁数
   rational m;                     rational 型
   if (d > 0) {
      longint ml=pow(longint::TEN,d);      10^d を m1 に作る
      m=RRset(longint::ONE,ml);    コンストラクタは GCD を求めて約分するので RRset を使用
      m = m / rational::TWO;       m は (1/10^d) / 2
      if (val.s < 0) val -= m; else val += m;  四捨五入
   }
```

中間部は，小数点よりも上の数字を （分子）÷（分母）で得られる商を，いったん文字列ストリーム ss に格納する．それを文字列 ns に格納し，その文字数を length 関数で得てから，符号や桁合わせの空白の挿入を行ってから，文字列ストリーム res に加える．桁数が多い数の場合は省略する桁数を (20 digits) のように res に加える．与えられた桁数が不足している場合は **** によってオーバーフローを示す．整数部分の最後に小数点を res に加える．なお，ndd 関数は int 型の 10 進桁数をカウントする．

```
      q = longint::ldivide(val.N,val.D,&r);    小数点の上の数を除算で商として得る
```

[24] setprecision 指定では iostream と iomanip をインクルードする．

```
    ss << q;                                longint 型の商を << 演算子で文字列に変換
    std::string ns = ss.str();              ns に 10 進数の文字列として格納
    int nl = ns.length();                   length() で 10 進の桁数を取得
    int sl = val.s < 0 ? 1 : 0;             符号チェック
    if (nl+sl <= wi) {                      整数部が表示可能
       if (val.s < 0) --wi;                 負なら wi を 1 つ減らす
       while (wi > ((int)ns.length())) res << ' '; --wi;        先頭に空白を詰める
       if (sl > 0) res << '-';              負なら  - 符号
       res << ns;                           文字列を res に入れる
    } else if (wi >= 2+ndd(nl)+11) {        整数部を " (XXX digits) " が 11 桁
       int w2 = wi-sl-ndd(nl)-11;           ndd() は XXX を 10 進表示したときの桁数を返す
       int w3 = w2/2;                       w2 (XXX digits) w3
       w2 = w2 - w3;                        <-- d ----------->
       for (int i=0; i < w2; ++i) res << ns[i];
       res << " (" << (nl - w2 - w3) << " digits) ";    "(XXX digits)"を入れる
       for (int i=nl-w3; i < nl; ++i) res << ns[i];
    } else {
       for (int i=0; i < wi; ++i) res << '*'; オーバーフロー
    }
    res << '.';                             小数点
```

後半は小数点よりも下の数字を res に加え，最終的に文字列型に変換した res.str() を返す．

```
    longint N, D;
    rational t;
    int len = 0;
    t = RRset(r,val.D);     val.D は四捨五入済み，GCD 計算しないために RRset
    while (len < d) {
       longint tn = t.N; longint td = t.D;
       tn = tn * longint::TEN;    rational::TEN 倍すると GCD 計算するので
       t = RRset(tn,td);          GCD 計算しないために RRset
       q = longint::ldivide(t.N,t.D,&r);
       res <<  q;                 1 桁ずつ文字型に変換して res に加える
       t = RRset(r,t.D);          GCD 計算しないために剰余 r と t の分母で RRset
       ++len;
    }
    return res.str();             res を文字列として返す
}
```

3.4 rough クラス

デバッグなどで，変数 a に格納されている多桁数または有理数を表示する場合，キャスト演算子 (double)a によって倍精度浮動小数点数に変換することが多い．しかし，浮動小数点数は指数範囲が限られているので，絶対値の大きな数は桁溢れし，絶対値が小さい数は零になるので，値を見ることができない．また，正確な計算に先立って概算によって解の存在する範囲を知りたいこともあるが，このような場合にも浮動小数点数は桁溢れが障害となる．そこで「有理数計算プログラミング環境」では rough 型の数値を用意した．精度は倍精度浮動小数点数と同じ 53 ビットだが，指数部が桁溢れしないので，デバッグ用の表示，収束判定，正確な解を挟み込んだり，初等関数などの非線形方程式の求解での初期値の設定で利用できる．

多桁数または有理数 a は，有意桁（significand）が 53 ビットの rough 数に近似できる．

$$a \fallingdotseq \pm f \cdot 2^n \qquad 1 \leq f < 2 \tag{9}$$

ただし $a = 0$ の場合は $f = 0$ とする．rough は，こじつけだが，ROUnd to 53 bits siGnificand witH power of two の略とした．

新たに実数を模擬する数値型を作ると，一般には，算術演算，論理演算，floor などの組込み関数，平方根の開平などの初等関数，その型の配列での使用，既存の型との間での型変換，10 進変換して表示などの機能を実装しなければならない．rough 型の用途は，表示して数値を確かめることが主目的なので，これらの機能を網羅せず，必要最小限の機能だけを実装した．実装は四則演算，論理演算，2 進数の rough 型の数値を 10 進数変換して cout に出力する表示機能が中心である．そのほか，平方根などの関数の精度指定を 10 進桁数で与えるため，10 進数としての桁数を調べるユーティリティルーチンを用意した．また，次章で述べる配列（マトリックスとベクトル）での使用も可能にした．はじめに double 型の代わりに rough 型で行う行列演算を例示する．

例題 testroughmat.cpp：rough 型対称行列の LDL^T 分解と行列式の計算　rough 型の $n \times n$ の正方行列 A を定義し，連立 1 次方程式を解き，行列式を計算する例題を示す．C や C++ で double 型で書くところを，rough 型に置換えれば，行列式の計算でも桁溢れしない．

rough 型マトリックスの行列式の計算 testroughmat.cpp

```
int main(int argc, char *argv[]){
  int n = atoi(argv[1]);
  rough_matrix a(n,n);  rough 型のマトリックス
  rough_vector b(n);    rough 型のベクトル
  dtgene2(a,b);         熱伝導行列を作成
  rough det=ldl2(a);    A を LDL 分解し，行列式の値を rough 型で返す
  std::cout << "det(A)=" << det << std::endl;
  ldlsubst(a,b);
  vecprn(b);
  return 0;
}
```

dtgene2 で生成する行列は，101 ページの「対称行列を生成する関数」に記した「熱伝導行列」である．行列サイズとして，4, 9, 16, 25, . . . などのように，整数 2, 3, 4, 5, . . . の 2 乗を与えると，そのサイズの熱伝導行列と適当な右辺ベクトルを生成する．

対称行列の三角分解を示す．

rough 型マトリックスの LDL 分解と行列式の計算

```
rough ldl2(matrix<rough>& a){
    int i,j,k;
    rough s,t;
    int n=a.rows();       rmatrix クラスのメンバ関数 rows() による行列の行数取得
    rough d(1.0,0);       引数をとるコンストラクタによる d の初期化
    for (j=0; j < n; ++j) {
        for (i=0; i < j; ++i) {
            s=rough(0,0);
            for (k=0; k < i; ++k)  s = s +  a[k][i] * a[k][j];
            a[i][j] = a[i][j] - s;
        }
        s=rough(0,0);
        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;
        d*=a[j][j];         行列式 det(A)=\prod_{j=1}^n A_{j,j}
    }
    return d;
}
```

行列式は $|A| = \prod_{j=1}^{n} a_{j,j}$ によって計算されるが，普通のオーダーの対角項でも，$n = 625 = 25 \times 25$ になると総積が桁溢れする．倍精度計算では，この部分は桁溢れを考慮したプログラミングをしなければならないが，rough 型ではその必要がない．625 を与えて実行すると，行列式の値 1.09475×10^{322} が得られる．

3.4.1 rough 型変数の定義と基本的な関数

rough クラスの定義を示す．

rough.h の rough 型の定義とメンバ関数

```
class rough {
   double d_;      有意桁（53 ビット）
   int e_;         指数（2 べき）
public:
   rough(): d_(0.0), e_(0) {}     コンストラクタ
   rough(double,int);             引数をとるコンストラクタ
   virtual ~rough() {}            デストラクタ
   double significand() const { return d_; }    メンバ関数，有意桁を倍精度数で返す
   int exponent() const { return e_; }          メンバ関数，指数を返す
   friend double roughdec(const rough&, int& m);     10 進変換
}
```

rough.cpp の引数をとるコンストラクタの実装を示す．Inf を渡したり，オーバーフローが起きた場合等，rough 版の NaN にしている．**rough 版の NaN（以下，rNaN）**は，有意桁が静止 NaN で，指数部がゼロである．Inf を渡したり，オーバーフローが起きた場合等，rNaN にしている[25]．

rough.cpp の コンストラクタ

```
rough::rough(double mant, int exp) {
   int e;
   if (!std::isnormal(mant) && mant != 0.0) {
      mant = std::numeric_limits<double>::quiet_NaN();
      exp = 0;
   } else if (mant == 0.0) {
      exp = 0;
   } else if (std::abs(mant) < 1.0 || std::abs(mant) >= 2.0) {
      double m = frexp(mant, &e);
      if ((exp > 0 && e-1 > 0 && exp+e-1 <= 0) || (exp < 0 && e-1 < 0 && exp+e-1 > 0)){
         mant = (m < 0 ? -1 : 1)*std::numeric_limits<double>::infinity();
         exp = 0;
      } else {
         mant = m * 2.0;
         exp += (e - 1);
      }
   }
   d_ = mant;
   e_ = exp;
}
```

longint 型，または rational 型の数値は，それを 2 進数で表現した場合の上位 53 ビットを取出して rough 数の有意桁とする．有意桁 da は $1 \leq da < 2$ に正規化して d_ にセットし，それに対する指数部を e_ にセットする．rough 型の数の有意桁は significand()，指数部は exponent() のメンバ関数で得られ，フレンド関数 roughdec は 10 進変換を行う．

オペレータオーバーロードされる四則演算子とインクリメント演算子を示す．

[25] rNaN になるのは，rough(Inf,e)，rough(-Inf,e)，rough(NaN,e)，rough(Den,e)，rough(0.0,e) の場合．

```
rough& operator=(const rough&);
rough& operator+=(const rough&);
rough& operator-=(const rough&);
rough& operator*=(const rough&);
rough& operator/=(const rough&);
inline rough operator-() const {   // unary minus operator
   return rough(-d_,e_);
}
rough& operator++(); // prefix ++
rough& operator--(); // prefix --
rough operator++(int);  // postfix ++
rough operator--(int);  // postfix --
```

ここでは加算の実装のみを示す.

rough.cpp の rough 型の数の加算

```
rough& rough::operator+=(const rough& r) {
   if (r.d_ != 0.0) {
      rough t = r;
      if (e_ < t.e_) swap(*this,t);              指数部の大小比較して
      if (e_ == t.e_) {                              指数部が同じなら
         d_ += t.d_;                                 有効桁を加える
      } else if ((e_ - t.e_) < DBL_MANT_DIG) {   違っていれば
         d_ += (t.d_/std::pow(2.0,e_-t.e_));         シフトして揃えてから加える
      }
      if (d_ != 0) normalize();                  正規化
   }
   return *this;
}
```

正規化は次のように実装されている.

rough.cpp の normalize 関数

```
void rough::normalize() {
   assert(d_ != std::numeric_limits<double>::infinity() &&
          d_ != -std::numeric_limits<double>::infinity() &&
          d_ != std::numeric_limits<double>::quiet_NaN());
   if (d_ != 0) {
      if (std::abs(d_) >= 2.0) {
         while (std::abs(d_) >= 2.0) {
            if (e_ == std::numeric_limits<long>::max()) {
               d_ = std::numeric_limits<double>::quiet_NaN();
               e_ = 0; break;
            }
            d_ /= 2.0; e_++;
         }
      } else if  (std::abs(d_) < 1.0) {
         while (std::abs(d_) < 1.0) {
            if (e_ == std::numeric_limits<long>::min()) {
               d_ = std::numeric_limits<double>::quiet_NaN();
               e_ = 0; break;
            }
            d_ *= 2.0; e_--;
         }
      }
   }
}
```

rough は指数の範囲が広いので Inf 等は不要と考え，演算でオーバーフロー等が生じた場合 rNaN にしてある．加算，減算，乗算では 2 つのオペランドのうち 1 つでも rNaN であれば，rNaN が返される．除算ではこれに加えて，正規化された数を零で割った場合も，rNaN が返される．

オペレータオーバーロードされる**論理演算子**を示す．

```
    friend bool operator==(const rough& x, const rough& y);
    friend bool operator!=(const rough& x, const rough& y);
    friend bool operator>(const rough& x, const rough& y);
    friend bool operator<(const rough& x, const rough& y);
    friend bool operator>=(const rough& x, const rough& y);
    friend bool operator<=(const rough& x, const rough& y);
```

比較演算子のみ，rough.cpp の実装を示す．

rough.cpp の == 演算子

```
bool operator==(const rough& x, const rough& y) {
   return (x.significand()==y.significand()) && (x.exponent()== y.exponent());
}
```

swap と，その他の関数のプロトタイプ宣言を示す．

```
    inline static void swap(rough& a, rough& b) {
       std::swap(a.d_,b.d_);
       std::swap(a.e_,b.e_);
    }
    friend double roughdec(const rough&, int& m);
    rough abs() const;
};
```

3.4.2 rough 型への変換プログラム numeric_cast

longint 型，rational 型，uint32_t 型の数値を rough 型へ変換する機能は，テンプレート関数を用いた．それ以外の型からは，rational 型への変換を介して変換する[26]．テンプレート関数では，テンプレート引数を 2 種類（T は target，S は source）記述する．

numeric_cast.h の template 関数

```
template<typename T, typename S> T numeric_cast(S const& s);
template<>                                          2 つの typename を指定するので必要
rough numeric_cast<rough,uint32_t>(uint32_t const& n);   uint32_t 型から rough 型へ
template<>                                          2 つの typename を指定するので必要
rough numeric_cast<rough,longint>(longint const& n);     longint 型から rough 型へ
template<>
rough numeric_cast<rough,rational>(rational const& s);   rational 型から rough 型へ
```

この定義を使用して，関数 numeric_cast<rough>(uint32_t 型引数) を呼ぶと，uint32_t 型を rough 型に変換する．

[26]rough 型から uint64_t 型へは，double 型を経由して変換する例が，120 ページの「基数変換した開平法による sqrt4g 関数」にある．

numeric_cast.cpp の uint32_t 型から rough 型への変換

```
template<>
rough numeric_cast<rough,uint32_t>(uint32_t const& n) {
   return rough(static_cast<double>(n),0);
}
```

関数 numeric_cast<rough>(longint 型引数) を呼ぶと，これが longint2double 関数を呼出して，多桁数の上位 53 ビットを抽出して有意桁 mant に，2 進数の桁を指数 exp に入れる．

numeric_cast.cpp の longint2double 関数と longint 型から rough 型への変換

```
static double longint2double(longint n, int& exp) {
    uint32_t bits = numbits(n);   仮引数 n のビット数を numbits() で取得して bits に格納
    if (bits < DBL_MANT_DIG) {      52 ビット以下なら
        n <<= (DBL_MANT_DIG - bits);     53-bits ビット左シフト
    } else {                        53 ビット以上なら
        n >>= (bits - DBL_MANT_DIG);     bits-53 ビット右シフト
    }
    exp = exp + bits - 1;           指数部に bits-1 を加える
    return static_cast<double>(n.to_ullong())*std::pow(2.0,-52); 仮引数 n の下位 64 ビット
}                    を double 型に変換して 2^{-52} を掛けた double 型の数を返す

template<>
rough numeric_cast<rough,longint>(longint const& n) {
   int exp = 0;
   double mant = longint2double(n,exp);
   return rough(mant,exp);
}
```

to_ullong 関数は，22 ページに示した longint クラスのメンバ関数で，ここではシフトして 53 ビット以下にした longint 型の数を，符号なしの 64 ビット数にする．これを static_cast<double> によって倍精度変換して返す．また，DBL_MANT_DIG は仮数部のビット数で 53 である（#include <cfloat> が必要）．

同様に rational 型の数に対しても次の関数を用意した．

numeric_cast.cpp の rational 型から rough 型への変換

```
template<>
rough numeric_cast<rough,rational>(rational const& s) {
   int sign = s < rational::ZERO ? -1 : 1;
   longint n = s.numerator();
   longint d = s.denominator();
   int nsz = numbits(n);            分子のビット数
   int dsz = numbits(d);            分母のビット数
   int exp = 0;
   if (nsz < dsz + DBL_MANT_DIG + 1) { 分母のビット数+54 が分子のビット数より大きければ
      int shifts = dsz + DBL_MANT_DIG - nsz;
      n <<= shifts;              分子を shifts ビット左にシフト
      exp -= shifts;             指数を shifts 小さくする
   }                              分子/分母の商を longint2double に渡し
   double mant = sign*longint2double(n/d,exp);  rough 型の指数，有意桁を得て
   return rough(mant,exp);                      rough 型を返す
}
```

3.4.3 rough 型の 10 進変換プログラムの実装

rough 型の数も，次のプログラムを実行すると，通常の変数のように 10 進数で表示することができる．

```
double dsqrt2=sqrt(2.0);                        //  double 型の sqrt{2}
```

```
rational rsqrt2=Rdset(&dsqrt2);                 //  rational 型の sqrt{2}
rough roughsqrt2=numeric_cast<rough>(rsqrt2);     //  rough 型の sqrt{2}
std::cout << " roughsqrt2=" << roughsqrt2 << std::endl; // 10 進数で表示
```

実行すると roughsqrt2=1.41421 の表示が得られる．

桁数の多い数の例を示す．longint 型の数 a を rough 型の数 b に変換して，正規化された有意桁を da に，指数部を pw に取り出す例を示す．

```
longint a("123456789876543210123456789876543210");      // 36 桁の数をコンストラクタで作る
std::cout << " a=" << a << std::endl;                    // longint 型の数値を cout で表示
std::cout << " a.str()=" << a.str() << std::endl;        // 内部形式で cout で表示
rough b=numeric_cast<rough>(a);                          // longint 型から rough 型に変換して
std::cout << " b=" << b << std::endl;                    //  rough 型を cout で表示
int pw=b.exponent(); double db=b.significand();          //  rough 型の指数部と有効桁を取得し
std::cout << " db=" << db << " pw=" << pw << std::endl; // それぞれを cout で表示
```

実行すると次の表示を得る．

```
a=123456789876543210123456789876543210                                  こんな桁数では困る
a.str()={ capacity: 64, l: 4, d=[{ 17c6e3 c2a1e8d0 f5431148 531b8eea}]} いくつだか分らない
b=1.23457e+035                                                          10 進数のおよその値
db=1.48606 pw=116                                                       2 進数のおよその値
```

36 桁の数 a は，longint 数として長さ 4 の配列に格納されていて，基数を $r = 2^{32}$ として 16 進法で $a = (\texttt{17c6e3})_{16} \times r^3 + (\texttt{c2a1e8d0})_{16} \times r^2 + (\texttt{f5431148})_{16} \times r + (\texttt{531b8eea})_{16}$ である．これを rough 型の変数 b に，numeric_cast 関数を経由して渡せば，10 進変換して，std::cout がその数字を文字変換して 1.23457e+035 と表示される．rough 型の数は数学的には 2 進数として扱うことができ，有意桁と指数部を significand() と exponent() で得ることができる．$1.23456 \times 10^{35} \fallingdotseq 1.48606 \times 2^{116}$ である．

前項で rough 型への型変換は説明したが，ここでは **rough 型の数を 10 進変換して表示するために，<< 演算子の再定義**を説明する．rough クラスの中に，rough 数の整数部分のビット列を定義する内部クラス intbits，小数部分のビット列を定義する内部クラス fracbits，2 進数を 10 進数に変換する際に必要とする内部クラス decbuf を設けた．

rough.h の内部クラス intbits と fracbits

```
class intbits {                 整数部分のビット列
   size_t size_;
   unsigned long long intg_;
public:
   intbits(rough const&);
   size_t size() const { return size_; }
   bool operator[](size_t pos) const;
};
class fracbits {                小数部分のビット列
   size_t size_;
   unsigned long long frac_;
public:
   fracbits(rough const&);
   size_t size() const { return size_; }
   bool operator[](size_t pos) const;
};
```

rough.cpp の内部クラス intbits の実装を示す．

rough.cpp の intbits の実装

```
rough::intbits::intbits(rough const& r) {
   b64_t t;
   t.dbl = r.significand();
   size_ = std::max(r.exponent()+1,0);
   intg_ = (t.llu & MANT_MASK);
   if (std::isnormal(t.dbl)) intg_ |= (1llu << (DBL_MANT_DIG-1));
}
bool rough::intbits::operator[](size_t pos) const {
   if (DBL_MANT_DIG+pos < size_) {
      return 0;
   }
   return (intg_ >> (DBL_MANT_DIG+pos-size_)) & 1;
}
```

rough.cpp の内部クラス fracbits の実装を示す.

rough.cpp の fracbits の実装

```
rough::fracbits::fracbits(rough const& r) {
   union b64_t t;
   t.dbl = r.significand();
   size_ = std::max(DBL_MANT_DIG-1-r.exponent(),0);
   frac_ = t.llu & MANT_MASK;
   if (std::isnormal(t.dbl)) frac_ |= (1llu<<(DBL_MANT_DIG-1));
}
bool rough::fracbits::operator[](size_t pos) const {
   if (DBL_MANT_DIG+pos < size_) {
      return 0;
   }
   return (frac_ >> (size_-pos-1)) & 1;
}
```

rough.cpp の内部クラス decbuf の実装を示す.

rough.cpp の decbuf の実装

```
rough::decbuf::decbuf(size_t size, size_t dp) {
   assert(size>=dp);
   size_ = size;
   dp_ = dp;
   buf_ = new char[size];
   assert(buf_ != 0);
}
```

次に rough.h にある本体の << 演算子の再定義を示す．はじめの 3 行で， traits 型から文字 T へ変換する template 関数，inline 化の指示，返す型（ostream）が “XX << YY <<” で YY までが ostream になっていなければいけないの 3 点を載せている．

<< 演算子は 2 つの引数 ostream と rough 型の数をとり,「左辺は ostream で，右辺は rough 型」である．

```
template <typename charT, typename traits> // template は << 演算子の ostream のために必要
inline                                     // inline 化を指示
std::basic_ostream<charT,traits>&
operator << (std::basic_ostream<charT,traits>& os, const rough& r) {
   size_t prec = os.precision();           変換する桁数
   rough::intbits n = r.getinteger();      指数部分の桁数
   rough::fracbits f = r.getfractional();  有意桁
   double const Log2 = 0.30103;
```

```
size_t const idigits = static_cast<size_t>(ceil(Log2*n.size())+1);
size_t const fdigits = std::max(static_cast<size_t>(ceil(Log2*f.size()+10.0)),prec+10);
rough::decbuf dd(idigits+fdigits,fdigits);  バッファ dd の定義, dd は decimal digit
dd.clear();                                 dd のクリア

// Convert to fixed decimal                 小数点の上
if (n.size() > 0) {                                 1101 なら
   dd[fdigits] = n[n.size()-1];                        1*2 +1
   for (size_t i=n.size()-1; i > 0; --i) {            1*2 +0
      dd.imul2add(n[i-1]);                           1*2 +1
   }                                               の演算を decbuf に作成
}
if (f.size() > 0) {                         小数点の下
   dd[fdigits-1] = f[f.size()-1]*5;
   for (size_t i=f.size()-1; i > 0; --i) {
      dd.fdiv2add(f[i-1]*5);
   }
}
std::basic_ostringstream<charT,traits> ss;
ss.copyfmt(os);                            カレントモードが hex か dec かとか,.
ss.width(0);                                precision(17) などを取得

size_t msd, lsd;
char carry;

// find most significant digit position
for (msd=dd.size(); msd > 0; --msd) {         formating での桁数の調整
   if (dd[msd-1] != 0) break;
}
lsd = msd < prec ? 0 : msd - prec;
if (r.significand() < 0.0) {
   ss << '-';
}
if (msd == 0) {
   ss << '0';
} else {
   // round up.
   carry = dd.round(msd,lsd);
   if (carry > 0) {
      lsd = msd;
      ++msd;
      dd[msd-1] = carry;
   }
   // suppress trailing zeros.
   if (lsd < dd.dp()) {                   dp() で小数点の位置を得る (decimal point)
      lsd = dd.suppress(dd.dp(),lsd);     lsd は least significant digit
   }
   // insert formatted string             数字の数によって場合分けして ss に文字列を入れる
   if (msd > dd.dp()+prec) {
      lsd = dd.suppress(msd,lsd);
      dd.insert_float(ss,msd,lsd);
   } else if (msd == dd.dp()+prec && lsd >= dd.dp()) {
      dd.insert_fixed(ss,msd,lsd);
   } else if (msd > dd.dp() && lsd < dd.dp()) {
      dd.insert_fixed(ss,msd,lsd);
   } else if (msd == dd.dp()) {
      dd.insert_fixed(ss,msd,lsd);
   } else {
      if (dd.dp()-lsd+2 < msd-lsd+1+5) {       1 は E, 5 は e\pm 5 桁
         dd.insert_fixed(ss,msd,lsd);          固定小数点で
```

```
            } else {
               dd.insert_float(ss,msd,lsd);          浮動小数点で印字
            }
         }
      }
      std::cout << ss.str();
      return os;
   }
```

imul2add 関数や fdiv2add 関数は，rough.h に含まれる．ここでは buf にある数を（シフトにより）2 倍し，それに加える imul2add のみを示す．

rough.h の imul2add 関数

```
      char imul2add(char d) {
         char carry = 0;
         for (size_t j=dp_; j <size_; ++j) {
            char t = (buf_[j] << 1) + carry;
            if (t >= 10) {
               carry = 1;
               t -= 10;
            } else {
               carry = 0;
            }
            buf_[j] = t;
         }
         buf_[dp_] += d;
         return carry;
      }
```

3.4.4 その他の関数

この他，フレンド関数で abs() なども用意した．abs 関数は sqrt4g で使用している．

次の roughdec 関数が rough 型の数 $f \cdot 2^n$ を 10 進変換して $g \cdot 10^m$ にする．

rough.cpp の 10 進変換関数 roughdec

```
double roughdec(const rough& f, int& m)
{
    int q = f.e_ / 10L; int r = f.e_ % 10L;
    double g = f.d_ * pow(2,r) * pow(1.024,q);
    m = 3*q;
    return g;
}
```

この関数は，2 進から 10 進への変換において $2^{10} = 1.024 \cdot 10^3$ を用いる．n を 10 で割った商と余りで，$n = 10 \cdot q + r$ ただし $q = \left\lfloor \frac{n}{10} \right\rfloor$ とする．

$$f \cdot 2^n = f \cdot 2^{10q+r} = f \cdot 2^r \cdot (2^{10})^q = f \cdot 2^r \cdot (1.024 \cdot 10^3)^q = \underbrace{f \cdot 2^r \cdot 1.024^q}_{g} \cdot 10^{3q} = g \cdot 10^m \qquad (10)$$

(double)a で rational 型の数を表示しようとして，倍精度数の桁溢れするような大きな rational 型，または longint 型の数 a に対しても表示可能になる．

roughdec の使用例は，次節 interval クラスの formatprn 関数にあり，rational 数の 10 進数でのおよその桁数を知るために使用している．

3.5 interval クラス

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．無理数を有理算術演算で扱うには，区間算術演算（interval arithmetic）を用いる．「有理数計算プログラミング環境」の interval 型の変数は，2 つの有理数で区間の下限と上限を把握する[27]．これによって区間を数のように扱える．interval 型の変数同士の加減算と乗算，また interval 型と rational 型の変数の乗算は，記号 $+,\ -,\ *$ でプログラミングできる（C++ の演算子多重定義を用いた）．

上限と下限が異符号の（零を含む）区間が入力オペランドにある場合は，エラー扱いする場合がある．これは，非線形方程式の係数が有理数で与えられた問題では，方程式の定数項から，零を含む区間 (a, b) を $(a, 0)$ と $(0, b)$ の 2 つの区間に分けたほうが扱いが簡単になる場合が多く，また区間算術演算のサポートも簡単になるからである．区間 $x = (a,\ b)$ の符号を反転した $y = (-b,\ -a)$ は y=x.neg() と書く．区間 $x = (a, b)$ の逆数 $y = \left(\dfrac{1}{b},\ \dfrac{1}{a}\right)$ は y=x.inv() と書く．interval 型の変数同士の除算は，逆数を掛ける．

本節の第 1 項と 2 項で interval クラスの実装を紹介する．有理算術演算で無理数解を調べるには，解の存在する区間幅との対話（interaction）が重要な役割を担う．そのためのユーティリティルーチンとなる，区間幅の常用対数や区間幅を見て必要な数字の桁の位置を認識して表示する formatprn 関数を，第 3 節で説明する．

3.5.1 区間数 interval の定義

区間 $\alpha = (a, b)$ を 2 つの有理数 a と b で表すには，“interval alpha(a,b)” と書けば，コンストラクタが上限と下限の有理数をセットする．また区間 alpha からその下限を取り出すには alpha.lower()，上限を取り出すには alpha.upper() を使用する．interval クラスのヘッダーファイル interval.h を示す．

```
class interval {
    rational l_; rational u_;                          下限値と上限値を rational 型で保持
public:
    interval() : l_(rational(0u)), u_(rational(0u)) {}  コンストラクタ（初期値は零）
    interval(rational const& a, rational const& b) : l_(a), u_(b) {}
    interval(const interval& b);
    virtual ~interval();                               デストラクタ
    rational lower() const { return l_; }              メンバー関数，下限値を返す
    rational upper() const { return u_; }              メンバー関数，上限値を返す
    interval inv();                                    逆数
    interval neg();                                    符号反転
    interval& operator=(interval b);
    interval operator+(interval const& b) const;  //  interval + interval
    interval operator-(interval const& b) const;  //  interval - interval
    interval operator*(rational const& b) const;  //  interval * rational
    interval operator*(interval const& b) const;  //  interval * interval
    bool operator==(interval const& b) const;          比較演算子
    std::string format(int w, int d);                  書式つき表示
};
double log10int(const interval& a);          区間幅の常用対数
void formatprn(const interval& x);           区間の下限と上限の差を見て必要桁を選んで表示
std::ostream& operator<<(std::ostream& o, interval const& a);
```

演算子 = を示す．左辺は this がポインタなので，アスタリスクを付けて返す．

[27] 2 つの浮動小数点数で下限と上限を把握する精度保証計算では，浮動小数点演算によって区間算術演算を実装するので，プログラミングでは丸め誤差を考慮する必要がある．浮動小数点演算の丸め誤差の理論を熟知していないと難しい．有理算術演算では丸め誤差がないので，区間算術演算の実装は比較的難易度が低いと思われる．

演算子多重定義による = 演算子

```
interval& interval::operator=(interval b) {
    std::swap(l_,b.l_);
    std::swap(u_,b.u_);
    return *this;
}
```

逆数をとる inv 関数を示す．RRset は GCD 計算と約分をしないので rational::ONE を割るよりも速い．

逆数をとる inv

```
interval interval::inv(){
   longint ln=this->l_.numerator(); longint ld=this->l_.denominator();
   longint un=this->u_.numerator(); longint ud=this->u_.denominator();
   rational b=RRset(ld,ln); rational a=RRset(ud,un);
   interval res=interval(a,b);
   return res;
}
```

符号反転する neg 関数を示す．上限が下限に，下限が上限に反転する．

符号反転する neg

```
interval interval::neg(){
   interval res=interval(-this->u_,-this->l_);
   return res;
}
```

3.5.2 区間算術演算

区間 $\alpha_1 = (a_1, b_1)$ と $\alpha_2 = (a_2, b_2)$ で表される 2 数の加算（乗算）は，両区間が正の場合は，下限同士の和（積）と上限同士の和（積）になる．

$$\alpha_1 + \alpha_2 = (a_1 + a_2,\ b_1 + b_2), \quad \alpha_1\alpha_2 = (a_1a_2,\ b_1b_2) \tag{11}$$

区間は常に下限が $-\infty$ 側にあり，上限が $+\infty$ 側にある．

演算子多重定義による区間加算

```
interval interval::operator+(interval const& b) const {
    interval res(this->l_ + b.l_, this->u_ + b.u_);
    return res;
}
```

interval 型の変数に rational 型の変数を加える演算子多重定義も用意したが，ここでは省略する．

減算は，下限は小さいほうから大きいほうを引き，上限は大きいほうから小さいほうを引く．

$$\alpha_1 - \alpha_2 = (a_1 - b_2,\ b_1 - a_2) \tag{12}$$

なお，**区間の上限と下限は同符号または零とし，異符号（区間内部に零をもつ区間）はエラー**とする場合がある．

加減算は簡単だが，乗算は符号を考慮するので複雑である．両区間とも負の場合は積は正になるので反転して $\alpha_1\alpha_2 = (b_1b_2, a_1a_2)$ になる．α_1 が負 α_2 が正の場合は $\alpha_1\alpha_2 = (a_1b_2, b_1a_2)$ になり，α_1 が正 α_2 が負の場合は $\alpha_1\alpha_2 = (b_1a_2, a_1b_2)$ になる．区間 $\alpha_k = (a_k, b_k)$ の符号反転は $-\alpha_k = (-b_k, -a_k)$ になる．

```
interval interval::operator*(interval const& b) const {
    rational ua,ub,za,zb;
```

```
        rational xa=this->l_; rational xb=this->u_;
        rational ya=b.l_;     rational yb=b.u_;
        int xsgn, ysgn;
        if(xa.s != xb.s){
            std::cout << "interval::operator* this sign ERROR STOP" << std::endl; exit(3);
        }
        if(ya.s != yb.s){
            std::cout << "interval::operator* b sign ERROR STOP" << std::endl; exit(3);
        }
        if(xa > xb){
            std::cout << "interval::operator* this->l_ larger than u_ ERROR STOP" << std::endl; exit(3);
        }
        if(ya > yb){
            std::cout << "interval::operator* b.l_ larger than b.u_ ERROR STOP" << std::endl; exit(3);
        }

        xsgn=0;
        if(xa > rational::ZERO){ xsgn=1;
        }else if(xa < rational::ZERO){ xsgn=-1;
        }
        if(xsgn == 0){
            if(xb > rational::ZERO) xsgn=1;
        }         // i.e., xsgn==0 means (Xa,Xb)=(0,0)

        ysgn=0;
        if(ya > rational::ZERO){ ysgn=1;
        }else if(ya < rational::ZERO){ ysgn=-1;
        }
        if(ysgn == 0){
            if(yb > rational::ZERO) ysgn=1;
        }         // i.e., ysgn==0 means (Ya,Yb)=(0,0)

        if(xsgn*ysgn > 0){  // result +
            ua=xa*ya; ub=xb*yb;
            if( xsgn > 0){ za=ua; zb=ub;
            }else{ za=ub; zb=ua;
            }
        }else if(xsgn*ysgn < 0){ //  result -
            ub=xb*ya; ua=xa*yb;
            if( xsgn > 0){ za=ub; zb=ua;
            }else{ za=ua; zb=ub;
            }
        }else if(xsgn*ysgn == 0){ //  result 0
            za=rational::ZERO; zb=rational::ZERO;
        }
        interval res(za,zb);
        return res;
    }
```

3.5.3 区間算術演算用ユーティリティルーチン

区間型の変数は，上限と下限の差から区間幅を検出できるので，rational クラスのメンバー関数を a.format(w,d) のように，表示する桁数の幅 w,d を指定して使用する必要がなくなる．formatprn(a) とするだけで，区間型の変数 a を表示できる．はじめに区間幅の常用対数を得る log10int 関数を示してから formatprn(a) 関数を説明する．

区間型変数の区間幅の常用対数を返す関数 log10int　「有理数計算プログラミング環境」のライブラリ関数の多くは，非線形方程式の解を扱う場合，解を含む区間を interval 型の変数で挟み込み，区間幅を 10 進数で指定された精度まで縮小反復するものが多い．例えば 100 を指定されたら，区間幅が 10^{-100} となるまで縮小反復する．この区間の数値を表示すると，小数点以下 100 桁以降で上限と下限の差が始まる場合が多い．しかし区間が整数を含むような場合は次のようになる．

```
1.9999999999999999999 (87 digits) 99999999999999999
2.0000000000000000000 (87 digits) 00000000000034046
```

このときの区間幅を得るための関数が `log10int` である．区間幅が，倍精度で下位桁溢れしなければ，区間 `a` を `(double)(a.upper()-a.lower())` によって区間幅を倍精度数で求めて，倍精度用の `log10` 関数を用いればよいが，倍精度浮動小数点数は 10 の −324 乗以下は 0 になって使えない．下位桁溢れの可能性のある区間に対しては `log10int` 関数を用いる．この関数は次ように書かれている．

interval クラスの log10int 関数

```
double log10int(const interval& a){
   rational t=a.upper()-a.lower();          区間幅  t
   int nbits; double f=ndpw2(t,nbits);      rough 数に変換 t = f *2^{nbits}
   dbllgt=log2(f)+(double)nbits;            2 進数での桁数
   return 0.30102999566398119 5*dbllgt;     対数の底の変換
}
```

区間表示プログラム formatprn と interval クラスの format 関数　区間を構成する有理数の桁数が多いので，途中を飛ばして，違いが始まる桁の前後の数桁を実行時に検出して表示する formatprn 関数を用意した．53 ページの 3.3.3 項に述べた rational クラスの format 関数は，小数点よりも上の桁数が多い場合に，それを `(20 digits)` のように，飛ばした桁数を表示し，小数点以下は指定された桁数までを省略せずにすべて表示した．これに対し，formatprn 関数は内部で interval クラスのメンバー関数 format を呼び出すことで，区間の上限と下限の差から**区間幅を得ることで，差異の現れる桁の位置を認識し，ここまでの（小数点以下の）桁数を表示する**．ただしこの桁数が多い場合に，これを `(20 digits)` のように省略する．

はじめに表示例を示す．これは，“`std::cout << "eig="; formatprn(eigint[i]);`” と指定した．

```
eig=
       58241765506971327.4587497873925059528 (87 digits) 484436139384907159
       58241765506971327.4587497873925059528 (87 digits) 484436139384934046
```

区間の下限と上限は，小数点以下 119 桁が同じである．

上限と下限が等しい区間に対しては，次のように表示する．

```
22719829205299720.00000000000000000000000000000000000000000000000000000: (a=b)
```

`formatprn` 関数は，interval クラスのメンバー関数 format を呼出し，これが小数点の上と，小数点，それ以下の省略するまでの数（上の例では ···59528 まで）を rational クラスのメンバー関数 format で処理し，省略する桁数とそれ以下の数字の表示（上の例では `(87 digits)`4844···07159）を interval クラスのメンバー関数 format が処理している．

この処理を，素朴に 10 進変換を繰り返して行うと，有理数の分母・分子の桁数が多い場合，かなりの処理時間を要する．はじめに高速化に関する指針を述べる．

- rational 型の算術演算は GCD 計算に時間を要するので，できるだけ longint 型の算術演算を使用する．
- コンストラクタで 2 つの longint 型の数を分母・分子に入れて rational 型の数を作ると GCD を求めて約分するが，RRset(分子，分母) とすると GCD 計算を行わないので速い．

- 10 のべき乗は，何度も 10 倍すると遅くなるので，longint クラスの pow 関数を用いる．

なお，下位の桁の最後の数字は**切捨て**ている．四捨五入すると，formatprn 関数の処理時間が伸びること，また差異の生ずる桁よりも下の 5 桁を表示しているので，四捨五入の必要性が薄いからである．

formatprn 関数　下限と上限の差から表示する桁数を決定して表示する親ルーチン formatprn を示す．

formatprn 関数

```
void formatprn(const interval& x){
   int nplaceup,nplacef,aapw2,aapw10,nplace;
   rational a=x.lower(), b=x.upper();   区間 (a,b)
   rational aa=a.abs();                 区間の下限の絶対値
   double ba=(double)(b-a);             区間幅
   rough raa=numeric_cast<rough>(aa);   rough 数に変換
   aapw2=raa.exponent(); double daa=raa.significand();
   double daa10=roughdec(daa,aapw10);   小数点の上の桁数
   if(a != b){
      double dba=log10int(x);           小数点以下の桁数として区間幅の常用対数をセット
      int nbadgt=-(int)dba;             区間幅の桁数（差異の始まる桁の位置）
      nplacef=nbadgt+5;                 小数点以下の数値の表示桁数
      if(nplacef > 50 || nplacef < 0 ) nplacef=50;       小数点以下の表示は 50 桁まで
   }else{
      nplacef=50;
   }
   nplaceup=aapw10;
   if(nplaceup > 30 || nplaceup < 0 ) nplaceup=30;    30 桁以上は 30 にする
   nplace=10+nplaceup+nplacef;                    全体の桁数
   std::cout << x.format(nplace,nplacef) << std::endl; interval クラスの format 関数呼出
}
```

format 関数　interval クラスの，文字列を返すメンバー関数 format(w,d) の引数 w には全体の桁数，d には小数点以下の桁数が渡す．

はじめに，左シフト，切捨て floor()，右シフトを行って切捨てる roundoff 関数を作成しておく[28]．pow 関数は 35 ページの longint クラスの 10 の n 乗を求める関数である．

10 進数の n 桁以下を切り捨てる roundoff 関数

```
rational roundoff(const rational& x, const int n){
   rational y,z;
   if(x.s==0) y.s=0;
   longint xn=x.N, xd=x.D;
   longint t=pow(longint::TEN,n); //  t=10^n
   xn=xn*t; y=RRset(xn,xd);       //  shift left
   z=y.rational::floor();         //  floor() で切り捨て
   y=RRset(z.N,z.D*t);            //  shift right
   if(x.s<0) y.s=-1;
   return y;
}
```

最初の処理は，interval 型の引数 this に対して，区間 (a, b) の区間幅 $b - a$ を rational 型の正確な演算により変数 diff に格納し，その小数点以下の桁数を，区間幅を常用対数 log10int 関数によって変数 lz に入れる．

区間の上限と下限が等しい（diff が零）場合は，下限を表示のあとに (a=b) を表示して処理を終える．

```
std::string interval::format(int w, int d) {
```

[28] 同じ桁数を左と右に shl10 と shr10 でシフトすると，10 のべき乗を 2 回生成する無駄があるのでまとめた．

```
    std::stringstream ss;
    int lz;
//      xxxx.xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyy
//      xxxx.xxxxxxxxxxxxxxxxzzzzzzzzzzzzzzzz
//           <-- lz --------->
//
    rational diff = this->upper()-this->lower();  // width of this interval
    if (diff != rational::ZERO) {
       rational t = diff;
       double dbllgten=log10int(*this);
       lz=-(int)dbllgten;  // count number of leading zero digits after point.
    }else{                 //  a==b の場合
       ss << "\n";
       ss << this->lower().format(w,d) << ": (a=b)";  // 一方のみと (a=b) を表示して
       return ss.str();    //  処理を終える
    }
```

lz が指定文字数 d よりも小さければそのまま表示に回すが，そうでない場合は省略する数字を決める．数字が数 1000 桁あっても，実際に表示するのは，上の 10 数桁と，上限と下限で差が出る桁を含む下の 10 数桁である．上の 10 数桁の桁数は w2，下の 10 数桁の桁数は w3 に決める[29]．

```
    if (lz < d) {          // lz が 表示幅よりも少ない場合
       ss << "\n";
       ss << this->lower().format(w,d) << ":\n" << this->upper().format(w,d);
       return ss.str();    //  両方をを表示して処理を終える
    } else {
       int nplaceup=w-6-d; // recover the number of digits upper than point.
       int dt = lz + 9;    // use (leading zero + 9) for under the point
       int wt = w+lz;
       int k = ndd(dt) + 11; // ndd(dt) is number of digits of XX, 11 is ( XX digits )
       if( d < k ) k = d;
       int w2 = (d - k)/2;   // w2 is digits before ( XX digits )
       int w3 = (d - k - w2);// w3 is digits after ( XX digits )
       if (ndd(dt-w2-w3)+11 < k) ++w2;
       ss << "\n";
```

プログラムの次の部分は，tt0=this->lower() で interval 型変数の下限を rational 型変数 tt0 に格納し，その下位の桁を roundoff 関数で切捨て，rational クラスの format 関数で文字列に変換して，文字列ストリーム型の ss に << 演算子で 10 進数の文字列に変換して格納し，その後に省略する文字数を (xxx digits) と格納する．

次に省略する桁を含む dt-w3 桁に対し，同様に切捨てた数を tt3 に取り出し，元の数 tt0 から引くことで，下位の w3 桁を tt1 に取り出す．これを dt 桁シフトしてから，除算で切捨て，商 qt3 を文字変換して ss に加える[30]．

```
       rational tt0=this->lower();     // get lower rational number of this interval
       rational tt3=roundoff(tt0,w2);  // shift left w2 digits, floor(), shift right w2 digits
       ss << tt3.format(nplaceup+6+w2,w2);// print upper portion
       ss << " (" << dt-w2-w3 << " digits) " ;   // digits to skip are ( xxx digits)
       tt3=roundoff(tt0,dt-w3);   // shift left dt-w3 digits, floor(), shift right dt-w3 digits
       rational tt1=tt0-tt3;      // subtract off to get lower w3 digits
       rational tt2=shl10(tt1,dt);// shift left dt digits
       longint qt3=tt2.N / tt2.D; // no rounding-up, i.e., always round-off
       ss <<  qt3;                // print lower portion
```

[29] ndd 関数は Number of Decimal Digits の略で，10 進数の桁数を返す rational クラスの関数である．

[30] 実際に 10 進数の文字列を作ると遅くなるので，有理数を構成する**分母を** 10^n **倍して右シフト**する shr10 関数と，**分子を** 10^n **倍して左シフト**する shl10 関数関数を使用した（本資料では省略する）．

```
        ss << "\n";                // new line
```

最後に，上限値に対しても，下限値と同様の処理を行い ss に加えて表示が完了する．

```
        tt0=this->upper();                    // get upper rational number of this interval
        tt3=roundoff(tt0,w2);                 // shift left w2 digits, floor(), shift right w2 digits
        ss << tt3.format(nplaceup+6+w2,w2);   // print upper portion
        ss << " (" << dt-w2-w3 << " digits) " ; // digits to skip are ( xxx digits)
        tt3=roundoff(tt0,dt-w3); // shift left dt-w3 digits, floor(), shift right dt-w3 digits
        tt1=tt0-tt3;             // subtract off to get lower w3 digits
        tt2=shl10(tt1,dt);       // shift left dt digits
        qt3=tt2.N / tt2.D;
        ss <<  qt3;              // print lower portion
        return ss.str();
    }
}
```

表示例　$\tan^{-1}$ の連分数展開を使用して $\tan^{-1} 1 = \dfrac{\pi}{4}$ を，計算すると，連分数展開の特質から，π を挟む区間が得られる．プログラム PiCfracAtanRint.cpp により，連分数展開の 2000 項と 2001 項の結果を interval 型の変数 y の上限と下限に入れて “std::cout << " pi="; formatprn(y);” により表示した．このように，下限と上限を有理数で抑えられ，1531 桁まで正しいことが一目で分かる．このとき π を表す有理数の分母と分子の多桁数は，それぞれ 2^{32} 進数で 200 桁であった．

```
input n=
 pi=
        3.14159265358979323 (1504 digits) 236480665088297165
        3.14159265358979323 (1504 digits) 236480665569605919
```

小数点の上の桁数の多い場合は次のように，小数点の上と下に省略した桁数が表示される．

```
Lpi=
     314159 (134 digits) 25359.40812848111745028 (1356 digits) 596023648066508829
     314159 (134 digits) 25359.40812848111745028 (1356 digits) 596023648066556960
```

“314159 (134 digits) 25359.40812848111745028” は inteval::format 関数が呼出す rational::format 関数が作成し，後半の “(1356 digits) 596023648066508829” は interval::format 関数自身で作成している．

デバッグでの使用例　interval 型の 2 つの区間が z と Z に得られているとき，両者の差異を比較したい場合の例を示す．

formatprn 関数を利用したデバッグ例

```
rational Zl=Z.lower(); rational Zu=Z.upper();
if(zl<Zl){
   interval zzl(zl,Zl);
   std::cout << " zZl=" <<  std::endl; formatprn(zzl);
}else{
   interval zzl(Zl,zl);
   std::cout << " Zzl=" <<  std::endl; formatprn(zzl);
}
if(zu<Zu){
   interval zzu(zu,Zu);
   std::cout << " zZu=" <<  std::endl; formatprn(zzu);
}else{
   interval zzu(Zu,zu);
   std::cout << " Zzu=" <<  std::endl; formatprn(zzu);
}
```

桁数の多い rational 型の数値の何桁目を見たらよいかが分からない場合，差異のある桁を formatprn 関数に探させることが可能な点が，作業の効率を向上させる．

4 行列演算：rvector，rmatrix，rblas，prblas クラスとチェックポイントリスタート機能

「有理数計算プログラミング環境」は，整数，浮動小数点数に加えて有理数を用いて数値計算を行い，正確な計算を実現することで，浮動小数点演算による数値計算プログラムの誤差解析に利用することを主たる目的として開発された．有理算術演算では，有理数に対する四則演算は非常に計算時間を要する．浮動小数点演算や 32 ビット整数の演算と比較すると，桁数の多い有理算術演算の計算時間は桁違いである[31]．このため高速化の対象を，有理算術演算による数値計算，特に行列演算に絞った．OpenMP によるようなループ演算のスレッド並列化はとりあえず考えずに，数値線形代数を扱う関数内部の反復計算だけを並列化する．これによって「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，計算機のコア数に適したスレッド数を Makefile の `NUM_THREADS` パラメータにセットして 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考えないでよく，数学教育用にも（十進 BASIC の延長上で）利用することを想定した．

NLA では Basic Linear Algebra Subprograms（BLAS）のインターフェースが定着している．BLAS はベクトル型スーパーコンピュータの時代に BLAS-1 と -2 が策定されたが，この時代は浮動小数点演算は整数演算よりも計算時間がかかる演算であった．行列演算も密行列に対する計算が高速化の対象であった．現在では倍精度浮動小数点演算は，整数演算に比べて特に遅い演算ではなくなり，また，四則演算よりもキャッシュミスを伴うメモリ参照のほうがはるかに計算時間を費やすようになった．有理算術演算では，有理数に対する計算が桁違いに計算時間を要する．また「有理数計算プログラミング環境」では，行列はすべて長方形行列としているので，密行列のインターフェースが使用できる．したがって，BLAS インターフェースは，rblas の並列版である prblas を作成するためのインターフェースとして有効であると考えた．

初期 BLAS は FORTRAN 66 や 77 の，FORTRAN の列優先の行列要素の順序付けを前提に，行列要素 a_{ij}，先導次元，ストライドを引数で渡すインターフェースが作られた．Fortran 90 の時代では，行列やベクトルの一部分（行ベクトル，列ベクトル，小行列）を定義できるようになった．「有理数計算プログラミング環境」では，C++の参照渡しを使用しているので，Fortran 90 のような部分行列を作ることができる．ヘッダーファイル rvector.h と rmatrix.h によって vector クラスと matrix クラスを設けた．この中で vector_section や row，column などの Fortran 90 と類似した部分行列の扱い方をサポートする．また，オブジェクト指向言語の特長である「継承」を使って，異なる変数型に対する操作を簡潔にプログラミングした．

本章では，第 1 節で Fortran がサポートするような BLAS インターフェースをサポートするための rvector と rmatrix クラスを解説し，第 2 節で rblas クラスを，第 3 節で prblas クラスを解説する．第 4 節で，チェックポイントリスタート機能を解説する．

4.1 rvector と rmatrix クラス

本節でははじめに rtaits.h に定義した内容を示す．次に rvector.h，rmatrix.h で定義した内容を示す．最後に，部分行列や部分ベクトルを定義する方法を示す．

[31] 対称行列の固有値の多重度を解析する dvsrch2.cpp プログラムの場合，倍精度浮動小数点演算による 2 次元トラス構造解析プログラムの中間結果である行列を有理数変換して，ヘッセンベルグ変換 $A \to H$，フロベニウス変換 $H \to F$ を行って正確な特性多項式を得る．これに，浮動小数点演算で得られた近似固有値の組合せを根と係数の関係公式に代入し，逆問題を解くようにして因子探しを行い，特性多項式の因数分解を得る．節点数が 13 の最も小さいデータでは，全体の計算時間 11.7 秒中，ヘッセンベルグ変換とフロベニウス変換に 11.2 秒かかり，他は 0.5 秒である．41 節点数の行列では，行列を変換して特性多項式を得るまでに 32.2 時間かかり，他の計算には 3 分弱の時間しかかからない（これは重根が存在して，ヘッセンベルグ変換の結果が複数のヘッセンベルグ小行列に分けられる問題を扱う場合の計算時間の例である）．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桁数増大のため）**問題の次数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い**ので計算時間の増大は急激である．

4.1.1 rtraits.h による型の定義

変数型を抽象化するために，STL（Standard Template Library）の型特性メタ関数 type traits を真似て type rtraits を作成した．rtraits.h に,「有理数計算プログラミング環境」で新たに作成した変数型（rational，rough，interval）を要素とするベクトル，部分ベクトル，行列，部分行列，列ベクトル，行ベクトルをコンパイルするために，型特性メタ関数 rtraits を作る．rtraits.h はかなり大きなファイルなので，ここでは rough 型など，少数の型についてだけ引用する．

```
template<typename T> class vector;              ベクトル，T にベクトル型が返るようにする
template<typename T> class vector_section;      部分ベクトル
template<typename T> class lvector_section;     部分ベクトル（書込み可能）
template<typename T> class matrix;              マトリックス（行列）
template<typename T> class matrix_section;      小行列
template<typename T> class lmatrix_section;     小行列（書込み可能）
template<typename T> class matrix_column;       列ベクトル
template<typename T> class lmatrix_column;      列ベクトル（書込み可能）
template<typename T> class matrix_row;          行ベクトル
template<typename T> class lmatrix_row;         行ベクトル（書込み可能）
template<typename T> class matrix_diagonal;     行列の対角要素ベクトル

template<typename T> class vector_traits;
template<typename T> class matrix_traits;

class interval;

template<>
class vector_traits<vector<int> > {
public:
        typedef int ElementT;
};
略

template<>
class vector_traits<matrix_row<rough> >         rough 型の matrix のために
{
public:
    typedef rough ElementT;                      要素型 T は rough です
};
以下，略
```

rtraits により，型に関連した属性を求めることができる．たとえば，rational型のマトリックス matrix<rational> から，その行の型である rational 型ベクトル vector<rational>を求めるのに使う．

rtraits.h は，rvector.h にインクルードされ，rvector.h は rmatrix.h，rblas.h，prblas.h，ratutil.h にインクルードされる．

4.1.2 rvector.h による vector クラス

整数型，浮動小数点数型，rough 型，多桁数 longint 型，有理数 rational 型，区間 interval 型の変数の配列（ベクトル）を扱うために設けた．基本となる ivector クラスを定義する．

```
#include "rtraits.h"
#include "archive.h"
template<typename DerivedT>
class ivector {
   DerivedT& true_this() {
      return static_cast<DerivedT&>(*this);
```

```
    }
    DerivedT const& true_this() const {
       return static_cast<const DerivedT&>(*this);
    }
public:
    typename vector_traits<DerivedT>::ElementT const& operator[] (int idx) const {
       return true_this().operator[](idx);
    }
    typename vector_traits<DerivedT>::ElementT& operator[] (int idx) {
       return true_this().operator[](idx);
    }
    int len() const {
       return true_this().len();
    }
};
```

1 次元配列である vector を ivector の派生型として定義する[32] .

```
template<typename T> class vector_section;
template<typename T> class lvector_section;
template<typename T>
class vector : public ivector<vector<T> > {
   T* vec_;                    // T 型へのポインタ vec_ の宣言
   int len_;
public:
   vector() : vec_(0), len_(0) {}                      //  コンストラクタ
   vector(int len) : vec_(new T[len]), len_(len) {}    //  new 演算子で vec_ のメモリを獲得
   ~vector() {                                         //  デストラクタ
      if (vec_ != 0) delete[] vec_;
   }
   vector<T>& operator =(const vector<T>& rhs) {       //  代入演算子
      if (this != &rhs) {
         assert(len_ == 0 || len_ == rhs.len_);
         if (len_ == 0) {
            vec_ = new T[rhs.len_];
            len_ = rhs.len_;
         }
         for (int i=0; i < len_; ++i) this->vec_[i] = rhs[i];
      }
      return *this;
   }
   template<template<typename> class U>
   vector<T>& operator =(ivector<U<T> > const& rhs) {     //  vector のコピー
      assert(len_ == 0 || len_ == rhs.len());
      if (len_ == 0) {
         vec_ = new T[rhs.len()];
         len_ = rhs.len();
      }
      for (int i=0; i < len_; ++i) this->vec_[i] = rhs[i];
      return *this;
   }
   const T& operator [](int idx) const { return vec_[idx]; }      // 添字演算子の定義
   T& operator [](int idx) { return vec_[idx]; }
   int len() const { return len_; }
   const vector_section<T>& section(int beg, int end, int stride) const{//section の定義
      vector_section<T>*  res = new vector_section<T>(*this, beg, end, stride);
```

[32] ivector を基本クラス（スーパークラス）とし，派生クラス（サブクラス）を :（コロン）で subclass : superclass と記述する．Java では extends キーワードで subclass extends superclass と記述する．継承（inheritance）は「サブクラスはスーパークラスの変形で，スーパークラスの意味を何らかの形で拡張したもの」である [1, p. 155]．スーパーとサブは，数学用語のスーパーセット，サブセットからとられた．

```
        return *res;
    }
    lvector_section<T>& lsection(int beg, int end, int stride){         //lsection の定義
        lvector_section<T>* res = new lvector_section<T>(*this, beg, end, stride);
        return *res;
    }
};
```

次に部分ベクトルの要素をアクセスするための vector_section を ivector の派生型として定義する．beg は最初の要素，end は最後の要素，stride はストライドである．関数の引数を ivector とすることにより vector も vector_section も扱えるようになる．

変数型は整数型， double 型，rough 型，rational 型，interval 型 が使用できる．ある型 <T> に対して定義して，CRTP（curiously recurring template pattern, CRTP，奇妙に再帰したテンプレートパターン）によって，ポルモリフィズムをコンパイル時に解決する．CRTP は，基本クラスのテンプレート引数として自分自身を代入する（ここでは基本クラスのテンプレートは，メンバ関数の本体（定義）はその宣言から非常に後になるまでインスタンス化されない性質を活用している）．基本クラスの機能の一部を変えてコードの再利用を行う場合，オーバーライドが使用される．これは変えたい機能を提供するメンバ関数を仮想関数にしてそれを派生クラスとして，動的に置き換える（オーバーライドする）[33]．しかしオーバーライドは実行時に動的に置換えるので HPC では使いたくない．CRTP を使えば仮想関数を用いずに行えるのでコンパイル時に静的に解決できる．

なお，rvector.h には，チェックポイント機能のために演算子 << や >> でアーカイブファイルを扱う定義を含む．

4.1.3 マトリックス rmatrix.h

整数型，double 型，rough 型，rational 型，interval 型を要素とするマトリックスをコンパイルするために設けた．「有理数計算プログラミング環境」ではマトリックスは**長方形行列**である．したがって，三角行列や疎行列を扱うにも，零要素を保存することになる．有理数計算では，非零の有理数は桁数が大きいので，零要素をもつことのデメリットは無視できる．

基本クラスとなる imatrix クラスを定義する．

```
#include <iostream>
#include <cstdlib>
#include <cassert>
#include <typeinfo>
#include "rvector.h"

template<typename DerivedT>          typename で DerivedT としているので
class imatrix {
    DerivedT& true_this() {
        return static_cast<DerivedT&>(*this);
    }
    DerivedT const& true_this() const {
```

[33] オブジェクト指向言語の特長の 1 つに「多相性（ポルモルフィズム）」（polymorphism）がある．この言葉は「たくさんの形」という意味をわかりにくく言った言葉で，同じ名前で複数の関数を指すという意味である [1, p. 160]．コンパイル時に解決できる多相性をオーバーロード（overload）という．これは「多重定義」で，同じ名前でシグネチャ（引数の数，型，順番）が異なる関数を，コンパイラがコンパイル時に区別する機能を指す．言語の I/O 機能はオーバーロード多相性のよく現れる場所であるが，C では `print(anytype)` とは書けず，`printf` に変数 `anytype` の型に従った書式を指定する（型ごとに書式を変える必要がある）．オーバーライド（override）は全く同じシグネチャを持つ関数を派生クラスで「再定義」したときに生じる．基本クラスのメンバ関数を再定義して仮想関数を作る（仮想関数は，基本クラスで再定義可能であるということを明示するために，virtual キーワードを付ける．virtual キーワードは基本クラスのみで指定する）．仮想関数の呼び出しは，通常のメンバ関数と代わらない．再定義されたメンバ関数は，探索順序に従って呼び出されるが，この判断をコンパイル時ではなく実行時に行う．オーバーロードはコンパイル時の静的な多相性だが，オーバーライドは実行時に決定される動的な多相性である．

```
      return static_cast<DerivedT const&>(*this);
   }
public:
   typename matrix_traits<DerivedT>::RowT const& operator[] (int idx) const {
      return true_this()[idx];          上の DerivedT にはコンパイル時には
   }                                       int, rough, rational などが返る
   typename matrix_traits<DerivedT>::RowT& operator[] (int idx) {
      return true_this()[idx];
   }
   int rows() const {
      return true_this().rows();
   }
   int columns() const {
      return true_this().columns();
   }
};
```

配列参照演算子 [ ] を再定義しているので，行ベクトルのオフセットを指せる．

imatrix 型に対する変更で，ある型 T のマトリックスのコンストラクタ，デストラクタを定義する．

```
template<typename T>
class matrix : public imatrix<matrix<T> > {
   int rows_;
   int columns_;
   vector<T>* m_;           //    型 T のベクトルの変数名が m_
public:
   matrix(int rows, int columns) : rows_(rows), columns_(columns), m_(0) {
      m_ = new vector<T>[rows_];         //  ベクトル長 rows のベクトルを m_ に new で獲得
      for (int i=0; i < rows_; ++i) {    //  rows 本の
         new (m_+i) vector<T>(columns_); //  ベクトル長 column のベクトルを new 演算子でとり
       }                                 //  m_ ベクトルに各ベクトルのアドレスを格納
   }
   matrix(const matrix<T>& rhs) : rows_(rhs.rows_), columns_(rhs.columns_), m_(0) {
      m_ = new vector<T>[rows_];                                          // アサイン演算子
      for (int i=0; i < rows_; ++i) {
         new (m_+i) vector<T>(columns_);
       }
      *this = rhs;
   }
   ~matrix() {             // デストラクタ
      delete[] m_;
   }
```

C++では鍵括弧 [ ] を演算子として再定義できるので，型 T のベクトル m_ のオフセット idx を指せる．

```
   const vector<T>& operator [](int idx) const { return m_[idx]; }
   vector<T>& operator [](int idx) { return m_[idx]; }
```

これらを用いて，$n \times n$ の正方行列 A を次のように定義し，フランク行列を作成する例を示す．

有理数型マトリックスの定義と行列要素へのアクセス

```
matrix<rational> a(n,n);
for (int i=0; i < n; ++i){           // n - max(i,j) + 1
   for (int j=0; j < n; ++j) a[i][j] = (n - ((i+1)>(j+1)?(i+1):(j+1))+1);
}
```

定義では括弧 () を使用し，要素の指定は C++の標準の [ ] によるオフセット（零から数える）を使用する．

これを使用すると，次元数 n が確定してから matrix<rational> a(n,n); や vector<rational> b(n);

によって動的にメモリを獲得できる[34]．C++では演算子 new によって動的にヒープに配列を獲得することもできるが，このようにして獲得したメモリ領域は delete 演算子によって削除する必要がある．定義文を用いると，コンストラクタが獲得して，制御がメモリのスコープ（有効範囲）を出た時点でデストラクタによて自動的に削除されるので，バグ（メモリリーク）を防ぐことができる．

4.1.4 列ベクトル，行ベクトル，小行列の定義

マトリックスの行数，列数を取得する関数 rows と columns を示す．

```
int rows() const { return rows_; }
int columns() const { return columns_; }
```

これを使用してマトリックスの列ベクトルを定義する．例えば，小行列（書き込み可能）の場合は次の lmatrix を使用する．

```
template<typename T>
class lmatrix_section : public imatrix<lmatrix_section<T> > {
    int rows_;
    int columns_;
    lvector_section<T>* rv_;        書込み可能な部分ベクトル
public:
    lmatrix_section() : rows_(0), columns_(0), rv_(0) {}
    lmatrix_section(matrix<T>& m, int row_begin, int row_end, int row_stride,
                    int col_begin, int col_end, int col_stride)
        : rows_((row_end-row_begin+row_stride)/row_stride),
    columns_((col_end-col_begin+col_stride)/col_stride), rv_(0) {
        rv_ = new lvector_section<T>[rows_];
        for (int i=0; i < rows_; ++i) {
            new (rv_+i) lvector_section<T>(m[row_begin + row_stride*i],col_begin,col_end,col_stride);
        }
    }
    ~lmatrix_section() {
        delete[] rv_;
    }
    lvector_section<T> const& operator[] (int idx) const { return rv_[idx]; }
    lvector_section<T>& operator[](int idx) { return rv_[idx]; }
    int rows() const { return rows_; }
    int columns() const { return columns_; }
};
```

これを用いると，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) で定義できる (数学では行や列は 1 から数えるが，C では 0 から数える)．

有理数型マトリックスの部分列ベクトルの定義

```
rational_matrix a(n,n);  A の定義
a.column(k-1,k,n-1,1));  第 k 列目の，k+1 から n 番目の要素からなるベクトル
```

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．**参照のみの場合は column でよいが，書き込みをする場合は lcolumn とする**．

行ベクトルも同様に定義する．

```
matrix_row<T>& row(int index) const {
   matrix_row<T>* res = new matrix_row<T>(*this,index);
   return *res;
}
matrix_row<T>& row(int index, int col_begin, int col_end, int col_stride) const {
```

[34] インタープリータ型の言語である Perl や十進 BASIC のように，必要なところで配列を獲得できる．

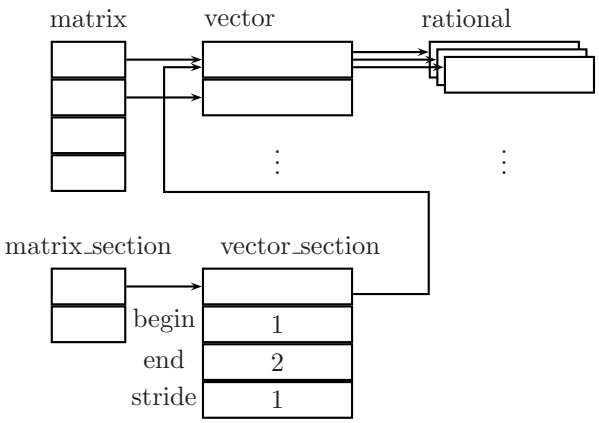


図 5: 小行列 section の実装

```
        matrix_row<T>* res = new matrix_row<T>(*this,index,col_begin,col_end,col_stride);
        return *res;
    }
```

これを用いると，行列 A の第 k 行の行ベクトルの $k+1$ から n 番目の要素からなるベクトルは，`a.row(k-1,k,n-1,1)` で定義できる．書き込みがある場合は lrow とする．

小行列は section で定義する．

```
matrix_section<T> const& section(int row_begin, int row_end, int row_stride,
                        int col_begin, int col_end, int col_stride);
```

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)` で定義できる．書き込みがある場合は lsection とする．

図 5 に，4 行からなる行列 A の，第 1 行と第 2 行からなる小行列 `section(1,2,1,1,2,1)` を示した．実装は，行列がベクトルの集合であり，各ベクトルは有理数の集合であるが，小行列はポインタでリンクを張ることで把握される．

なお，rmatrix.h には，チェックポイント機能のために演算子 << や >> でアーカイブファイルを扱う定義を含む．

4.2 rblas クラス

基本線形代数演算（Basic Linear Algebra Subprograms, BLAS）は，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 Sequence Association（SA）を前提にインターフェースが定義された．したがって，副プログラムに配列の途中の要素のアドレス，先導次元（leading dimension of array, LDA），ストライドを渡すことで，小行列の行ベクトルや列ベクトルを BLAS 副プログラムで扱うことができ，これが BLAS のメニューを少なくすることに効果があった．Fortran 90 では，行列やベクトルの一部分（行ベクトル，列ベクトル，小行列）を定義できる．前節で述べたように，C++の参照渡しを使用しているので，Fortran 90 のような部分行列が使用できる．「有理数計算プログラミング環境」では，逐次処理を行う rblas（Rational BLAS）と，その並列処理版である prblas（Parallel Rational BLAS）を作成した．ここに収めるメニューは，並列版の prblas によってスレッド並列で高速化できる基本線形代数演算で，BLAS での dot, axpy, copy, gemv などに，必要に応じてメニューを追加した．

行列の形状は，すべて長方形行列である（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．

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．シンボル $+$, $-$, $*$, / 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と GCD ルーチンを呼出す（Intel では call 命令）だけである．そこでプログラムを簡潔にする目的で，テンプレート関数を用いた[35]．数値の複数の型に対して，数学的に同等（mathematically equivalent）な処理を行うが，ある型に対する処理を記述しておき，これに対する変更をプログラミングすることで，複数の型に対するコードが生成できる．

4.2.1 rblas のメニュー

次の行列やベクトルの演算を行う．rational 型の場合の引数も示す．元祖 BLAS はベクトル長やストライドを引数に含めるが，rblas ではこれらは引数の中のベクトルや小行列の形状の中に織り込まれているので，calling sequence は異なる．

swap: 要素やベクトルの交換

scal: $\boldsymbol{x} \leftarrow \alpha\boldsymbol{x}$, rscal$(\alpha, \boldsymbol{x})$, 使用例は 104 ページを参照.

copy: $\boldsymbol{y} \leftarrow \boldsymbol{x}$, rcopy$(\boldsymbol{x}, \boldsymbol{y})$, 使用例は 106 ページの LDL を参照.

vdiv: $y_i \leftarrow y_i \div x_i,\ i = 1, \cdots$, rvdiv$(\boldsymbol{x}, \boldsymbol{y})$, 使用例は LDL 分解に対応する代入 LDLsubst を参照.

vdivs: $z_i \leftarrow y_i \div x_i,\ i = 1, \cdots$, rvdivs$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, 使用例は LDL 分解にあり，106 ページを参照.

vma: $z_i \leftarrow z_i + y_i \times x_i,\ i = 1, \cdots$, rvma$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, 使用例は 110 ページの hesfrb を参照.

axpy: $\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha\boldsymbol{x}$, raxpy$(\alpha, \boldsymbol{x}, \boldsymbol{y})$

dot: $\boldsymbol{x}^T\boldsymbol{y}$, rdot$(\boldsymbol{x}, \boldsymbol{y})$

asum: $\sum |x(i)|$

iamax: 絶対値最大の要素のインデックス

gemv: $\boldsymbol{y} \leftarrow \beta\boldsymbol{y} + \alpha A\boldsymbol{x}$, rgemv$(\alpha, A, \boldsymbol{x}, \beta, \boldsymbol{y})$, 使用例は 96 ページの cg を参照.

gemtv: $\boldsymbol{y} \leftarrow \beta\boldsymbol{y} + \alpha A^T\boldsymbol{x}$, rgemtv$(\alpha, A, \boldsymbol{x}, \beta, \boldsymbol{y})$, 使用例は 106 ページの LDL の後半を参照.

ger: $A \leftarrow A + \alpha\boldsymbol{x}\boldsymbol{y}^T$, rger$(\alpha, \boldsymbol{x}, \boldsymbol{y}, A)$, 使用例は 104 ページの LUdecomp の後半を参照.

4.2.2 rblas の実装

rblas のメニューを，複数のデータ型に対して多重定義する方法はとらずに，コンパイラに任せるテンプレート関数によって，汎用関数として作成した．

axpy を例にする．変数型を typename T で記述する[36]．ベースとなる手続きを _Raxpy で定義する[37]．

[35] テンプレート関数はコンパイラから見ると「コード生成のテンプレートとなる関数」である．

[36] C++の template 宣言は，C のプリプロセッサの「引数のついた関数マクロ」とほぼ同じであるが，定義段階，生成段階，使用段階の 3 段階に分かれるコード生成のうち，生成段階は自動的に行われるので必要ない [2, p. 472].

[37] typename キーワードは class キーワードを使用することもできる．

_Raxpy

```
template<typename T, template<typename> class T1, template<typename> class T2>
void _Raxpy(T const& alpha, ivector<T1<T> > const& x, ivector<T2<T> >& y) {
   for (int i=0; i < y.len(); ++i)  y[i] += alpha*x[i];
}
```

要素の型である T を double と rational にした関数として，倍精度型の daxpy と 有理数型の raxpy を定義する．

daxpy と raxpy

```
template<template<typename> class T1, template<typename> class T2>
void daxpy(double const& alpha, ivector<T1<double> > const& x, ivector<T2<double> >&y) {
   _Raxpy<double,T1,T2>(alpha,x,y);
}
template<template<typename> class T1, template<typename> class T2>
void raxpy(rational const& alpha, ivector<T1<rational> > const& x, ivector<T2<rational> >&y) {
   _Raxpy<rational,T1,T2>(alpha,x,y);
}
```

raxpy の引数 ivector には，マトリックスの行ベクトルやその一部が記述できる．

内積 _Rdot を示す．

_Rdot

```
template<typename T, template<typename> class T1, template<typename> class T2>
T _Rdot(const ivector<T1<T> >& a, const ivector<T2<T> >& b) {
   assert(a.len()==b.len());
   T dprod = T();
   for (size_t i=0; i < a.len(); ++i) dprod += a[i]*b[i];
   return dprod;
}
```

要素の型である T を double と rational にした関数として，整数型の idot と倍精度型の ddot と 有理数型の rdot を定義する．

daxpy と raxpy

```
template<template<typename> class T1, template<typename> class T2>
double idot(const ivector<T1<int> >& a, const ivector<T2<int> >& b) {
   return _Rdot<int,T1,T2>(a,b);
}
template<template<typename> class T1, template<typename> class T2>
double ddot(const ivector<T1<double> >& a, const ivector<T2<double> >& b) {
   return _Rdot<double,T1,T2>(a,b);
}
template<template<typename> class T1, template<typename> class T2>
rational rdot(const ivector<T1<rational> >& a, const ivector<T2<rational> >& b) {
   return _Rdot<rational,T1,T2>(a,b);
}
```

4.2.3 使用例

ガウス消去法の LU 分解 LUdecomp に対応する代入ルーチン LUsubst を例にする．軸選択に対する行と列の交換を省いた形で示す．コメントが rblas を使用しない場合のコードで，これを rblas 版の raxpy に置換える．行列の列ベクトル，ベクトルの部分ベクトルについては 78 ページの 4.1.4 項「列ベクトル，行ベクトル，小行列の定義」を参照されたい．前進代入も後進代入も axpy で並列化できるが，後進代入には対角項 u_{ii} で割る操作が入る．

LUsubst.cpp の raxpy 使用例

```
 void LUsubst(const rational_matrix& a, const int n, rational_vector& x,  略){
   int i,j,k;
   for (j=0; j < n-1; ++j) {
//    for (i=j+1; i < n; ++i) x[i]=x[i]-a[i][j]*x[j];
      rat::lvector_section<rational> xs1(x,j+1,n-1,1);  //  xs1 を定義して
      rat::raxpy(-x[j],a.column(j,j+1,n-1,1),xs1);      //  引数に xs1 を入れる
   }
   for (i=n-1; i >= 0; --i) {
      x[i] = x[i]/a[i][i];
//    for (k=0; k <= i-1; ++k) x[k]=x[k]-x[i]*a[k][i];
      rat::lvector_section<rational> xs2(x,0,i-1,1);    //  xs2 を定義して
      rat::raxpy(-x[i],a.column(i,0,i-1,1),xs2);        //  引数に xs2 を入れる
   }
 }
```

raxpy の最後の引数である xs1 は，直接 x.lsection(j+1,n-1,1) と記述しても計算可能であるが，制御が抜けたときの認識ができなくて，**デストラクタが起動せず，メモリーリークが発生**する．それで 1 行余計になるが，いったん rat::lvector_section で rational 型の部分ベクトルを定義して，これを引数に記入する．こうすると，xs1 を定義したブロックから制御が抜けるとき，デストラクタが動き，コンストラクタの new に対応する delete が起動する．

浮動小数点演算で，BLAS を使用した数値線形代数プログラムは，機械的に rblas に置換えることができる．

rdot の使用例は 96 ページの CG 法の例題を参照されたい．

4.3 prblas クラス

rblas の raxpy や rdot などの関数を，複数のスレッドで並列実行する．ループ添字をブロック分割またはサイクリック分割し，各スレッドは自分の担当の添字についてだけ計算する[38]．同期処理は，セマフォの提供する排他制御機能を使用した．本節では，はじめに同期処理の方法を説明し，その後 prblas の実装方式を説明する．

4.3.1 セマフォによる同期処理

スレッドの生成には pthread_create() 関数を用いる．この関数に pthread_t 型のスレッド変数，属性変数，スレッド関数と引数，それぞれのポインタを渡せば，スレッドが生成される．ベクトル長 80 の内積計算を rdot 関数で，ブロック分割して 8 ウェイで計算する場合を例とする．ここでは BLAS の ddot を参考に，仮のインターフェースを想定する．各スレッドの担当する反復セットは，1:10，11:20，···，71:80 になり，prdot 関数で，開始要素番号（begin），終了要素番号（end）と結果（ベクトル要素で begin から end までの部分内積）を rdot 関数の引数に追加すれば，並列計算できそうである．

```
void* prdot(begin,end,m,x,ix,y,iy,result);
```

この prdot をスレッド関数として起動するには，thread_maneger.cpp のコンストラクタからスレッド生成する．

```
rc = pthread_create(&threads[i], 0, prdot, (void*)&thread_num[i]));
```

[38] 反復回数が 10 回のループを 4 ウェイでブロック分割する場合，各スレッドが担当するループ添字（反復セット）は，スレッド 0 は (1,2,3), スレッド 1 は (4,5,6), スレッド 2 は (7,8), スレッド 3 は (9,10) となる．サイクリック分割する場合，スレッド 0 は (1,5,9), スレッド 1 は (2,6,10) スレッド 2 は (3,7), スレッド 3 は (4,8) となる．

ただし，並列化したい関数は，rdot だけではなく，BLAS の種々のルーチンなので，**もうワンクッション入れて，汎用的に task_t 構造体に，関数名とその関数の引数を預け**，pthread_create では prdot や praxpy の替わりに ptask を指定する．ptask は後述する thread_manager.cpp に置かれる．

POSIX thread 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．ここではセマフォを用いる[39]．

POSIX セマフォで定義されるインターフェースには次のようなものがある．

- `int sem_init(sem_t *sem, int pshared, undigned int value);`

  名前なしセマフォを初期化する．セマフォを，全スレッドが参照可能なメモリ領域に確保し初期化する．

- `int sem_post(sem_t *sem);`

  セマフォsem をインクリメント（ロックを解除）する．その結果，sem_wait で停止しているスレッドが呼び起こされてセマフォをロックできるようになる．1 回の sem_post によって呼び起こされるのは 1 スレッドだけで，どのスレッドを起動するかは非決定的である．

- `int sem_wait(sem_t *sem);`

  sem> 0 なら セマフォsem をデクリメントする．sem= 0 なら，呼び出したスレッドはブロックされる．

- `int sem_destroy(sem_t *sem);`

  セマフォオブジェクトを破壊する．

thread_manager.h　この中に sem_t 構造体を定義する．

プリプロセッサ変数 NUM_THREADS にスレッド数を入れる（省略時は 2 ウェイ）．

prblas では，1 つの関数を並列実行するのではなく，rdot，raxpy など複数の関数に対してそれらを並列実行するために task_t 型の変数を用意して，ここに並列実行する rdot などの関数の関数ポインタと引数をいったん格納して，後述する execute 関数によってそれらを並列実行する方式をとる．

thread_manager.h

```
class thread_manager {
public:
#ifdef NUM_THREADS
    static const int num_threads = NUM_THREADS;   // the number of threads including main thread
#else
    static const int num_threads = 2;   // the number of threads including main thread
#endif
    typedef struct {
        void (*f)(void*);         // function executing task
        void* params;             // parameters to task
    } task_t;
    thread_manager();
    ~thread_manager();
    static void execute(task_t* tasks, int num_tasks);
};
```

並列化する関数（スレッド関数）の関数ポインタを識別子 f で，スレッド関数に渡す引数を params でとる task_t 型の構造体を定義する．コンストラクタ，デストラクタ，およびこの構造体を引数としてとる execute 関数のプロトタイプ宣言を入れる．

[39]pthread.h や semaphore.h をインクルードし，リンクオプションで -lpthread を指定する．

thread_manager.cpp　thread_manager.cpp を 3 つに分けて示す．最初の部分は ptask 関数で，後述する prdot などの prblas の関数から thread_manager::execute を介して間接的に起動させられるスレッド関数である．

```
#include <pthread.h>
#include <semaphore.h>
#include <iostream>
#include <cassert>
#include <stdexcept>
#include "longint.h"
#include "mempool.h"
#include "thread_manager.h"
    pthread_t threads[thread_manager::num_threads];
    sem_t sem_start[thread_manager::num_threads];          // スレッド起動用セマフォ
    sem_t sem_result[thread_manager::num_threads];         // 結果を受取るためのセマフォ
    int thread_num[thread_manager::num_threads];
    thread_manager::task_t tasks_[thread_manager::num_threads];

    void* ptask(void* p){             //  スレッド並列で稼働する関数
        int tid = *((int*)p);
        mem_pool* mp = new mem_pool();
        longint::_mempool = mp;
        while(true) {
            sem_wait(&sem_start[tid]);
            if (tasks_[tid].f == 0) break;      // terminate thread
            tasks_[tid].f(tasks_[tid].params);  // execute the task
            sem_post(&sem_result[tid]);
        }
        pthread_exit(0);
        return 0;
    }
```

2 番目の部分は コンストラクタ，デストラクタまでとする．コンストラクタで pthread_create によって num_threads 個のスレッドを生成する．生成されたスレッドはスリープ状態に入る．

```
    thread_manager::thread_manager(){                  //  コンストラクタ
        int i, rc;
        for (i=1; i < num_threads; ++i) {
            if (sem_init(&sem_start[i],0,0) != 0) {     //  start セマフォの初期化
                throw runtime_error("thread_manager: sem_init() failed.");
            }
            if (sem_init(&sem_result[i],0,0) != 0) {    //  result セマフォの初期化
                throw runtime_error("thread_manager: sem_init() failed.");
            }
        }
        for(i=1; i < num_threads; ++i) {                 //  スレッドの生成
            thread_num[i] = i;
            if ((rc = pthread_create(&threads[i], 0, ptask, (void*)&thread_num[i])) == -1) {
                throw runtime_error("thread_manager: pthread_create() failed");
            }
        }
    }
    thread_manager::~thread_manager() {                  //  デストラクタ
        for(int i=1; i < num_threads; ++i) {
            tasks_[i].f = 0;
            tasks_[i].params = 0;
            sem_post(&sem_start[i]);
        }
        for (int i=1; i < num_threads; ++i)  pthread_join(threads[i],0);
```

```
    }
```

セマフォは不可分操作でインクリメントできるカウンタである[40]．デクリメントしようとしたときに，セマフォが 0 だとブロックされる．

3 番目の部分は スレッド関数を起動する thread_manager::execute である．

```
    void thread_manager::execute(task_t* tasks, int num_tasks){
        int i;
        i = 0;
        while (i < num_tasks) {
            int nt;
            if ((num_tasks - i) > num_threads) {
                nt = num_threads;
            } else {
                nt = num_tasks - i;
            }

            tasks_[0].f = tasks[i].f;                //  thread function pointer
            tasks_[0].params = tasks[i].params;      //  parameters
            for (int j=1; j < nt; ++j) {
                tasks_[j].f = tasks[i+j].f;
                tasks_[j].params = tasks[j].params;
                sem_post(&sem_start[j]);       // post semaphore 1 to nt, i.e., start thread j
            }

            tasks_[0].f(tasks_[0].params);           // execute 0 thread
                                             //  thread parallel computing
            for (int j=1; j < nt; ++j) {
                sem_wait(&sem_result[j]);    //  wait thread j
            }
            i += nt;
        }
    }
    thread_manager _tm = thread_manager();
```

4.3.2 prblas の実装

axpy を例にする．引数の型を格納する構造体をテンプレート関数で定義する．

axpy_param

```
template<typename T, template<typename> class T1, template<typename> class T2>
struct axpy_param {
   iterset is;
   T const* alpha_;
   ivector<T1<T> > const* x_;
   ivector<T2<T> >* y_;
};
```

タスクを定義するクラスをテンプレート関数で定義する．各タスク（partial_axpy）は，自分に与えられた反復セット begin，end，stride について，反復計算 “$y_i = \alpha \cdot x_i + y_i$” を，変数型 T に対して実行するコードとして生成される．

[40]Intel の x86 および EM64T アーキテクチャでは，32 ビット境界にアライメントが調整されたアドレスへの書き込みはアトミックに行われることが保証されている．

class axpy_task

```
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);
      }
   }
};
```

thread_manager に渡す関数ポインタと引数を格納する task_t 型の変数を，テンプレート関数で定義する．

thread_manager task

```
template<typename T,template<typename> class T1, template<typename> class T2>
thread_manager::task_t axpy_task<T,T1,T2>::tasks[thread_manager::num_threads];
```

その引数などを格納する構造体をテンプレート関数で生成する．

axpy_param task

```
template<typename T,template<typename> class T1, template<typename> class T2>
axpy_param<T,T1,T2> axpy_task<T,T1,T2>::axpy_params[thread_manager::num_threads];
```

基本クラスの_Paxpy をテンプレート関数で定義する．引数を格納する構造体に，自分のスレッド分の引数をセットする．スレッド並列化されるループ添字は，BLOCKDIST を指定するとブロック分割・分散を作成し，指定がない場合はサイクリック分割・分散とした．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．

_Paxpy

```
template<typename T, template<typename> class T1, template<typename> class T2>
void _Paxpy(T const& alpha, ivector<T1<T> > const& x, ivector<T2<T> >& y) {
   assert(x.len()==y.len());
#ifdef BLOCKDIST
   set_iterset_block(thread_manager::num_threads,x.len(),axpy_task<T,T1,T2>::axpy_params);
#else
   set_iterset_cyclic(thread_manager::num_threads,x.len(),axpy_task<T,T1,T2>::axpy_params);
#endif
   for (int i=0; i < thread_manager::num_threads; ++i) {
      axpy_task<T,T1,T2>::axpy_params[i].alpha_ = &alpha;
      axpy_task<T,T1,T2>::axpy_params[i].x_ = &x;
      axpy_task<T,T1,T2>::axpy_params[i].y_ = &y;
      axpy_task<T,T1,T2>::tasks[i].f = axpy_task<T,T1,T2>::partial_axpy;
      axpy_task<T,T1,T2>::tasks[i].params = &axpy_task<T,T1,T2>::axpy_params[i];
   }
   // Execute parallel tasks.
   thread_manager::execute(axpy_task<T,T1,T2>::tasks,thread_manager::num_threads);
}
```

これを継承して有理数型の praxpy を定義する．

paxpy

```
template<template<typename> class T1, template<typename> class T2>
void praxpy(rational const& alpha, ivector<T1<rational> > const& x, ivector<T2<rational> >&y) {
   _Paxpy<rational,T1,T2>(alpha,x,y);
}
```

4.3.3 反復セット

反復の分割を，ブロック分割の場合は次のようにセットする．

block 分割の反復セット

```
template<typename T>
void set_iterset_block(int nthreads, int n, T* task_param) {
   int ichunk = (n + nthreads - 1) / nthreads;
   int s = 0;
   for (int i=0; i < nthreads; ++i) {
      task_param[i].is.begin_= s;
      task_param[i].is.end_ = (s + ichunk) > n ? n : s + ichunk;
      task_param[i].is.stride_ = 1;
      s += ichunk;
   }
}
```

サイクリック分割の場合は次のようにセットする．

cyclic 分割の反復セット

```
template<typename T>
void set_iterset_cyclic(int nthreads, int n, T* task_param) {
   for (int i=0; i < nthreads; ++i) {
      task_param[i].is.begin_= i;
      task_param[i].is.end_ = (i < n) ? i + ((n - i - 1)/nthreads+1)*nthreads : i;
      task_param[i].is.stride_ = nthreads;
   }
}
```

4.4 チェックポイント・リスタート機能

有理数計算は計算時間がかかるので，再計算を避けるために，計算の途中結果のチェックポイントをとって，リスタート計算する機能が必要である．C++ のファイル I/O は，入力に istream，出力に ostream，入出力に iostream クラスを使用する[41]．ディスクファイルに対しては ifstream，ofstream，fstream を使用する（ifstream クラスは istream クラスから継承されたものである）．

本節の前半では，有理数計算プログラミング環境で新たに定義した，多桁数，有理数，ベクトル，行列などの変数に対するファイル入出力を << と >> 演算子多重定義による実装を示す[42]．後半で使用例を示す．

4.4.1 iarchive と oarchive クラス

はじめに archive.h に iarchive と oarchive クラスを定義する．この中で `getIs()` と `getOs()` により，引数のバイト列を返す関数を定義する．

[41] C++ではファイルをストリーム（バイトの連なり）と考え，ビットシフト演算子 << と >> を入力，出力演算子として多重定義している．

[42] interval 型の変数については，まだ例題にチェックポイント機能を使用するものがないので，実装していない．

archive.h

```
#include <iostream>
class iarchive {
public:
   iarchive() : is_(std::cin) {}
   iarchive(std::istream& is) : is_(is) {}
   std::istream& getIs() { return is_; }
private:
   std::istream& is_;
};
class oarchive {
public:
   oarchive() : os_(std::cout) {}
   oarchive(std::ostream& os) : os_(os) {}
   std::ostream& getOs() { return os_; }
private:
   std::ostream& os_;
};
iarchive& operator>>(iarchive&, int&);
oarchive& operator<<(oarchive&, const int&);
```

longint クラスと rational クラスの archive 機能　longint クラスと rational クラスに iarchive と oarchive ルーチンを設けた．oarchive 関数は，longint 型の変数の，桁数と各桁の数値をチェックポイントファイルへ書き出す．

longint クラスの oarchive

```
oarchive& operator<<(oarchive& oa, const longint& x){
    oa.getOs() << " " << x.l;
    for (int i=0; i < x.l; ++i) oa.getOs() << " " << x.d[i+1];
    return oa;
}
```

iarchive はチェックポイントファイルから読込む．

longint クラスの iarchive

```
iarchive& operator>>(iarchive& ia, longint& x) {
        int l;
        ia.getIs() >> l;
        if (l+1 >= x.capacity) x.realloc(l+1,3001);
        for (int i=0; i < l; ++i) ia.getIs()  >> x.d[i+1];
        x.l = l;
    return ia;
}
```

同様に rational クラスの同機能を示す．チェックポイントファイルへの書き出しを示す．

rational クラスの oarchive

```
oarchive& operator<<(oarchive& oa, const rational& r) {
    oa.getOs() << " " << r.s;
    oa << r.N << r.D;
    return oa;
}
```

チェックポイントファイルからの読込みを示す．

rational クラスの iarchive

```
iarchive& operator>>(iarchive& ia, rational& r) {
    ia.getIs() >> r.s;
    ia >> r.N >> r.D;
    return ia;
}
```

vector クラスと matrix クラスの archive 機能　vector クラスでは，整数，倍精度浮動小数点数，有理数の各変数に対して機能させるために，型を抽象化して，これらの任意の型で稼働するように，テンプレート関数を用いた．rvector.h の最後に，演算子 << と >> の多重定義を行い，vector 型変数の iarchive 機能を含めた．

vector の >> 演算子多重定義（rvector.h）

```
template<typename T>
iarchive& operator>>(iarchive& ia, vector<T>& v) {
    int size;
    ia.getIs() >> size;
    if (size != v.len()) {
      std::stringstream what;
      what << "archived vector size [" << size << "] != input vector size [" << v.len() << "]";
        throw std::runtime_error(what.str());
    }
    for (int i=0; i < v.len(); ++i) ia >> v[i];
    return ia;
}
```

oarchive のための << 演算子多重定義と oarchive 機能を示す．

vector の << 演算子多重定義（rvector.h）

```
template<typename T>
oarchive& operator<<(oarchive& oa, vector<T> const& v) {
    oa.getOs() << " " << v.len();
    for (int i=0; i < v.len(); ++i) {
        oa << v[i];
    }
    return oa;
}

template<typename T>
std::ostream& operator<<(std::ostream& o, ivector<T> const& b){
   o << "{";
   if (b.len() > 0) {
      o << b[0];
      for (int i=1; i < b.len(); ++i) {
         o << "," << b[i];
      }
   }
   o << "}";
   return o;
}
```

同様に matrix クラスの同機能を示す．

matrix の archive（rmatrix.h）

```
template<class T> iarchive& operator>>(iarchive& ia, matrix<T>& x) {
   int rows = x.rows();
   ia.getIs() >> rows;
   for (int i=0; i < rows; ++i) ia >> x[i];
   return ia;
}
template<class T> oarchive& operator<<(oarchive& oa, matrix<T> const&  x) {
   size_t rows = x.rows();
   oa.getOs() << " " << rows;
   for (size_t i=0; i < rows; ++i) oa << x[i];
   return oa;
}
template<typename T>
std::ostream& operator<<(std::ostream& os, const imatrix<T>& m) {
   os << "{";
   if (m.rows() > 0) {
      os << m[0];
      for (int i=1; i < m.rows(); ++i)  os << m[i];
   }
   os << "}";
   return os;
}
```

4.4.2 使用例

チェックポイントとリスタートの使用例は例題 dvsrch2.cpp，に含まれる．dvsrch2.cpp では，対称有理行列 A をヘッセンベルグ変換とフロベニウス変換し，特性多項式の因数分解を得て，行列 A の代数的構造を調べる．初回の実行では，ヘッセンベルグ行列とフロベニウス行列（特性多項式の係数）を別ファイルに書出す．フロベニウス変換が完了しないで，ジョブが終了した場合は，リスタートジョブの実行時にコマンド行引数で “1” を与えて，ヘッセンベルグ行列を読み込み，続きを計算する．フロベニウス変換も完了した場合は，リスタートジョブの実行時にコマンド行引数で “2” を与えて，フロベニウス行列を読み込み，続きの計算を実行する．ヘッセンベルグ行列はサイズが大きいのと，フロベニウス変換が完了すれば，計算に必要がないので，チェックポイントは 2 ファイルに分けた．ファイルのオープンとファイル名の決定を行い，ヘッセンベルグ行列をファイルに書出す部分と，対応するリスタートでコマンド行引数で “1” を与えた場合にこれを読み込む部分を示す[43]．

```
#include "archive.h"

rational_matrix ar(n+1,n+1);
ifstream fin;
std::ofstream ofs;
std::fstream ifs;
iarchive ia(ifs);
oarchive oa(ofs);
  中略
   if(chkres==0){
      // ----- convert real symmetric matrix to Hessenberg form
      tstart0=gettime();          /*********************************/
      elmhes(ar,n);               /***   ELMHES  ******************/
```

[43] 入力行列名が S13free.txt のとき，行列 $A = S^{-1}KS^{-1}$ が指定された場合は，ヘッセンベルグ行列は S13freeA.chk に，特性多項式は S13freeA2.chk に書かれる．行列 $A = K$ が指定された場合は，ヘッセンベルグ行列は S13freeK.chk に，特性多項式は S13freeK2.chk に行列 $A = M(-1)K$ が指定された場合は，ヘッセンベルグ行列は S13freeMinvK.chk に，特性多項式は S13freeMinvK2.chk に書かれる．

```
      std::cout << std::endl;     /*********************************/
      std::cout << " ELMHES complete, time=" << (double)(gettime()-tstart0)/1000000.0 略
      // ----- checkpoint H
      std::stringstream ofn;
      if(mattype==0){
         ofn << matname << "A" << ".chk";
         ofn2<< matname << "A2" << ".chk";
      }else if(mattype==1){
         ofn << matname << "K" << ".chk";
         ofn2<< matname << "K2" << ".chk";
      }else{
         ofn << matname << "MinvK" << ".chk";
         ofn2<< matname << "MinvK2" << ".chk";
      }
      ofs.open(ofn.str().c_str(),std::fstream::trunc);
      if (!ofs) {
         throw std::runtime_error("Write open failed for " + ofn.str());
      }
      oa << ar;
      ofs.close();
      ofs.open(ofn2.str().c_str(),std::fstream::trunc);  //  archive file for Frobenius matrix
      if (!ofs) {
         throw std::runtime_error("Write open failed for " + ofn2.str());
      }
   }else if(chkres==1){              //   read H from checkpoint file
      std::stringstream ifn;
      if(mattype==0){
         ifn << matname << "A" << ".chk";
      }else if(mattype==1){
         ifn << matname << "K" << ".chk";
      }else{
         ifn << matname << "MinvK" << ".chk";
      }
      ifs.open(ifn.str().c_str(),std::ofstream::in);
      if (!ifs) {
         throw std::runtime_error("Read open failed for " + ifn.str());
      }
      ia >> ar;
      std::cout << "Read Hessenberg matrix from checkpoint file" << std::endl;
      ifs.close();
   }
```

次にフロベニウス変換の結果である多項式をチェックポイントファイルに書き出す部分と，リスタートジョブでコマンド行引数に “2” を与えた場合にこれを読込む部分を示す．変数 nmat と配列 ith があれば特性多項式は処理できる．

```
   if(chkres==0){
      oa << nmat;
      oa << ith;
   }else if(chkres>=2){
      std::stringstream ifn;
      if(mattype==0){
         ifn << matname << "A2" << ".chk";
      }else if(mattype==1){
         ifn << matname << "K2" << ".chk";
      }else{
         ifn << matname << "MinvK2" << ".chk";
      }
      ifs.open(ifn.str().c_str(),std::ofstream::in);
```

```
        if (!ifs) {
           throw std::runtime_error("Read open failed for " + ifn.str());
        }
        ia >> nmat;
        ia >> ith;
        std::cout << "Read checkpoint file2 nmat=" << nmat << " ith=" << ith << std::endl;
     }
```

フロベニウス変換はヘッセンベルグ小行列ごとに行う（ループ変数 k のループの中で行う）．フロベニウス行列に対するチェックポイントとその読込み部分を示す．

```
        if(chkres<=1){
           tstart0=gettime();
           // compute characteristic polynomial of Hessenberg submatrix
           hesfrb(b,nsize,p);  /**  HESFRB  *********************/
           std::cout << "hesfrb complete, time=" << (double)(gettime()-tstart0)/1000000.0 << "(sec)" << std
           oa << p;
           std::flush(ofs);            //  バッファをフラッシュする
           std::cout << " wrote checkpoint file" << std::endl;
           std::cout << std::endl;
           if(k==0) ofs.close();       //   ファイルをクローズする
        }else{
           ia >> p;
           std::cout << "Read characterestic polynomial from checkpoint file" << std::endl;
        }
```

ループ変数 k が 0 のときが最後の小行列なので，ファイルをクローズする．

5 ratutil クラス

ratutil に含まれる関数を示す.

行列の変換，表示，有理数の分母・分子の桁数を調べる getdgts，有理行列の要素の分母・分子の桁数を調べる MatDgtCount，行列やベクトルの共通因子によってスケールするなどのユーティリティルーチンを示す.

ユーティリティルーチン

```
void cnvmat(const rat::matrix<double>& a, rat::matrix<rational>& ar, const int m, const int n);
void copymat(const rat::matrix<rational>& a, const int m, const int n, rat::matrix<rational>& c);
rational rat2int(const rat::matrix<rational>& a, const int n, rat::matrix<rational>& b);

void matprn(const rat::matrix<double>& a, const int m, const int n);
void matprn(const rat::matrix<rational>& a, const int m, const int n);
void matprt(const rat::matrix<rational>& a, const int m, const int n);

int getdgts(rational v);
void MatDgtCount(const rat::matrix<rational>& a, const int m, const int n);
void MatDgtCountPrt(const rat::matrix<rational>& a, const int m, const int n);
void MtrDgtCount(const rat::matrix<rational>& a, const int n);
void MtrDgtCountPrt(const rat::matrix<rational>& a, const int n);
void VctDgtCount(const rat::vector<rational>& a, const int m);
void SeeMatDiag(const rat::matrix<rational>& a, const int n);

void ScalMat(const rat::matrix<rational>& a, const int m, const int n, rat::vector<rational>& as,
             rat::matrix<rational>& aa);
void ScalVec(const rat::vector<rational>& a, const int n, rational& s, rat::vector<rational>& b,
             const int update);
rational ScalMatCol(rat::matrix<rational>& a, const int m, const int n, const int j, const int update);
int askinty(const double t, const double epsint, rational& vrat);
int askinty(const rational& t, const double epsint, rational& vrat);
void nxtcmb(const int n, const int r, int ith[]);
long long comb(const int n, const int r);
```

多項式処理ルーチン polydivchk などを示す.

多項式処理ルーチン

```
int polydivchk(rat::vector<rational>& u, const int m, const rat::vector<rational>& v, const int n,
               rat::vector<rational>& q,  rat::vector<rational>& r);
int getpolynomial(const int k, rat::vector<rational>& cof, const int npolynomial,
                  const rat::vector<int>& ppolynomial, const rat::vector<rational>& cpolynomial);
int putpolynomial(const int nsize, const rat::vector<rational>& cof, int npolynomial,
                  rat::vector<int>& ppolynomial, rat::vector<rational>& cpolynomial);

void polytexform(const rat::vector<rational>& p,const  int nsize);
void polytexformr(const rat::vector<rational>& p,const  int nsize);
void polytexform2(const rat::vector<rational>& p,const int nsize, const rational& lcm);
void polytexformr2(const rat::vector<rational>& p, const int nsize, const rational& lcm);
void polytexformint(const rat::vector<rational>& p, const int nsize);
void polytexformintfmt(const rat::vector<rational>& p, const int nsize);
void polytexformintr(const rat::vector<rational>& p, const int nsize);

double vietaterm(const double e[], const int n, const int r);
rational vietaterm(const rat::vector<rational>& e, const int n, const int r);
int vietaterm(const rat::vector<rational>& ea, const rat::vector<rational>& eb, const int n,
              const int r, rational& ta, rational& tb);
int vietaterm(const rat::vector<rational>& e, const int n, const int r, rational& ta, rational& tb);
int vietaterm(const rat::vector<rational>& ea, const rat::vector<rational>& eb, const int n,
              const int r, rational& ta, rational& tb, const rat::vector<rational>& spa,
      const rat::vector<rational>& spb, rat::vector<rational>& ssa, rat::vector<rational>& ssb);
int cmpary(const int a[], const int b[], const int n);
void intmul(const rational& xa, const rational& xb, const rational& ya, const rational& yb,
            rational& za, rational& zb);
```

行列生成ルーチンを示す．S で始まる関数は対称行列を生成する.

行列生成ルーチン

```
void GenRand(rat::matrix<rational>& a, const int n, const int r, long long& iseed);
void GenMat(rat::matrix<rational>& a, const int m, const int n, const int r, long long& iseed);
void SetRand2(rat::matrix<rational>& a, const int n, long long& iseed);
void SetRand(rat::matrix<rational>& a, const int n, long long& iseed);
void MyRND(long long& iseed, rational& drand);
void MyIRND(long long& iseed);
double urand( void );
void FloatUrandR(rat::matrix<rational>& a, const int m, const int n);
void FloatUrand(rat::matrix<rational>& a, const int n);
void SetFrank(rat::matrix<double>& a, const int n);
void SetFrank(rat::matrix<rational>& a, const int n);
void SetHilbert(rat::matrix<double>& a, const int n);
void SetHilbert(rat::matrix<rational>& a, int n);
void SetFtherm(rat::matrix<rational>& a, const int m, const int cont);
void SetFtherm(rat::matrix<double>& a, const int m);
void SetGLmat(rat::matrix<double>& a, const int m);
```

倍精度浮動小数点演算ルーチンを示す[44].

倍精度浮動小数点演算ルーチン

```
int jacobev(rat::matrix<double>& A, rat::matrix<double>& V,double tol,int n,int maxit);
void bsort(rat::vector<double>& x, int n, int * ind);
void bsort(rat::vector<double>& x, int n);
```

行列に対する数値線形代数用ルーチンを示す.

数値線形代数用ルーチン

```
void elmhes(rat::matrix<rational>& a, const int n);
void hesfrb(rat::matrix<rational>& a, const int n, rat::vector<rational>& p);
void solhmg(rat::matrix<rational>& a, const int n, int& rank, rat::matrix<rational>& x);
void LUdecomp(rat::matrix<rational>& a, const int n, rat::vector<int>& ipvt, rat::vector<int>& jpvt,
              int& krank);
void LUsubst(const rat::matrix<rational>& a, const int n, rat::vector<rational>& x,
             const rat::vector<int>& ipvt, const rat::vector<int>& jpvt);
void LDL(rat::matrix<rational>& a, int n);
void LDLsubst(const rat::matrix<rational>& a, int n, rat::vector<rational>& x);
```

連分数ルーチン

```
rational sqrt2(longint X, uint32_t n);
interval setpi(const uint32_t tol);
rational bestappfrac(const rational& c,const double tol);
```

平方根を求める関数を示す.

平方根ルーチン

```
rational sqrt4g(longint X, uint32_t n);
rational SQR(const rational& a, const uint32_t tol);
interval SQRint(const rational& a, const uint32_t tol);
```

多項式の零点を求める 2 分法や行列演算ルーチンを示す. bisect は 2 分法で多項式の零点を探索し, bisectrf は変形された挟み撃ち法の変形で bisect より少ない反復で多項式の零点を探索する. Horner と bisect と bisectrf は intarval 型の引数をとるものがある.

[44] これらのルーチンの本文での説明は省略する. `jacobev` はヤコビ法で固有値, 固有ベクトルを求める. `bsort` はバブルソートルーチンで, 固有値の大小をソートするために dvsrch.cpp などで使用している. ヤコビ法による対称行列の固有値・固有ベクトルの計算方法については,『有理数計算による対称行列の固有値問題における特性多項式の因子探索』dvsrch.pdf の付録に説明した.

関数値と非線形方程式の求解ルーチン

```
interval roundrat(const rational& x, const int n);
interval roundrat2(const rational& x, const int n, const int m);

int newton(const rat::vector<rational>& p, const int n, const int tol, const int maxit, rational& a, rational& fa, ration
int bisectrf(const rat::vector<rational>& p, const int n, const double tol, const int maxit,
          rational& a, rational& fa, rational& b, rational& fb, rational& c);
int bisect(const rat::vector<rational>& p, const int n, const double tol, const int maxit, rational& a,
          rational& fa, rational& b, rational& fb, rational& c);
rational Horner(const int n, const rat::vector<rational>& coef, const rational& x);
void Horner3(const int n, const rat::vector<rational>& coef, const rational& x, rational& r,
            rational& s, rational& t);

void Horner(const int n, const rat::vector<rational>& coef, const interval& x, rational& r, rational& s);
int bisectrf(const rat::vector<rational>& p, const int n, const double tol, const int maxit,
          interval& ab, rational& fa, rational& fb, rational& c);
int bisect(const rat::vector<rational>& p, const int n, const double tol, const int maxit,
          interval& ab, rational& fa, rational& fb, rational& c);
```

5.1 ユーティリティルーチン

5.1.1 行列の変換 cnvmat 複写 copymat 表示 matprn など

倍精度浮動小数点数の $m \times n$ の行列を，有理数型に変換するには，cnvmat 関数を使用する．

また，有理行列の全要素の分母の LCM を求めてこれをかけ，整数行列（変数型は rational）にする関数を用意した．LCM を返す．

```
rational rat2int(const rational_matrix& a, const int n, rational_matrix& b) {
    longint gcdd, sd, lcm;
    gcdd = a[0][0].denominator();
    lcm = longint::ONE;
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            if(a[i][j] != rational::ZERO){
               sd = a[i][j].denominator();
               if(gcdd != longint::ZERO && lcm != longint::ZERO) gcdd=longint::lgcd2(sd,lcm);
               lcm=lcm*sd; lcm=lcm/gcdd;
            }
        }
    }
    rational Cof = RRset(lcm,longint::ONE);
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++) b[i][j] = a[i][j]*Cof;
    }
    return  Cof;
}
```

これらの使用例は dvsrch.cpp，dvsrch1.cpp，dvsrch2.cpp を参照されたい．

有理数型の行列を複写するには，copymat 関数を使用する．

rational 型の行列は `matprn(a,m,n);` で表示する．

```
void matprn(rational_matrix& a, int m, int n){
   for(int i=0; i < m; i++){
      for(int j=0; j < n; j++) std::cout << a[i][j] << " ";
      std::cout << std::endl;
   }
}
```

関数多重定義で第 1 引数が倍精度浮動小数点型の場合は浮動小数点数を表示する.
桁数が大きい場合は matprt 関数を使用すると倍精度型の表示を使用できる.

```
void matprt(rational_matrix& a, int m, int n){
略
         std::cout << (double)a[i][j] << " ";
```

5.1.2 例題 CG 法とベクトルの桁数削減 ScalVec，桁数カウント MatDgtCount など

共役勾配法（conjugate gradient method,CG 法 ）のアルゴリズムを示す.
$\boldsymbol{p}_i^T A\boldsymbol{p}_j = 0$ 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（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 [4]． 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.

CG 法アルゴリズム

$k = 0;\ \boldsymbol{x}_0 = 0;\ \boldsymbol{r}_0 = \boldsymbol{b}$
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
　$\alpha_k = \dfrac{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}{\boldsymbol{p}_k^T A\boldsymbol{p}_k}$　　！残差ノルムの最小化条件より
　$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$
　$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k A\boldsymbol{p}_k$
end
$\boldsymbol{x} = \boldsymbol{x}_k$

CG 法の関数を示す（cg.cpp）．有理数計算は正確な計算なので，CG 法は n 回以内の反復で収束するので，for ループで $n + 1$ 回反復して，そのまま反復が完了したらエラーとしている．収束判定は，浮動小数点演算で用いる誤差理論から得られるマシンイプシロンなどは用いることなく，零で行っている[45].

```
void cg(const rat::matrix<rational>& a, const rat::vector<rational>& b,
        rat::vector<rational>& x, const int n, int& nit) {
   rat::vector<rational> r(n), z(n), p(n), q(n);
   rational alpha, beta, pq, bnrm2, rr, resid, rho, rho1;
   int i, k;
   rat::rgemv(rational::ONE,a,x,rational::ZERO,r);
   for (i=0; i < n; ++i) r[i] = b[i]-r[i];
   bnrm2 = rat::rdot(b,b);
   rr = bnrm2;   //  <----------    ノルムの 2 乗

   for (k=1; k <= n+1; ++k) {     //  <----------    反復
      resid=RatAbs(rr/bnrm2);
      std::cout  << "iteration " << k << ", resid=" << (double)resid << std::endl;
      rho = rr;
      if (k > 1) {
```

[45]rdot 関数はページの rblas の rdot 関数を参照されたい.

```
            beta = rho/rho1;
            rat::rcopy(r,z);
            rat::raxpy(beta,p,z);
            rat::rcopy(z,p);
         } else {
            beta = rational::ZERO;
            rat::rcopy(r,p);
         }
         rat::rgemv(rational::ONE,a,p,rational::ZERO,q);
         pq = rat::rdot(p,q);
         alpha = rho/pq;
         rat::raxpy(alpha,p,x);   //  current solution vector
         rat::raxpy(-alpha,q,r);  //  residual vector
         rr = rat::rdot(r,r);
         resid = rr/bnrm2;
         longint rsd = resid.denominator();
         longint rsn = resid.numerator();
         std::cout << " Num of Dgts (resid)=" << numdgts(rsd)+numdgts(rsn) <<  std::endl;
         if (resid <= rational::ZERO) break;       //  <----    収束判定
         rho1 = rho;
      }
      if (resid > rational::ZERO) {
         std::cerr << "Iteration fail in subroutine cg" << std::endl;   //  <----- Error
         exit(1);
      }
      std::cout << "Converged CG k=" << k << std::endl;
      nit = k;
   }
```

桁数削減は，特に共役勾配法で大きな効果が確認された．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[46].

$$\boldsymbol{p} = \begin{pmatrix} \frac{b_1}{a_1} \\ \frac{b_2}{a_2} \\ \vdots \\ \frac{b_n}{a_n} \end{pmatrix} = s_p \begin{pmatrix} \frac{d_1}{c_1} \\ \frac{d_2}{c_2} \\ \vdots \\ \frac{d_n}{c_n} \end{pmatrix} = s_p \boldsymbol{p}_s, \quad s_p = \frac{\mathrm{GCD}(b_1, b_2, \cdots, b_n)}{\mathrm{GCD}(a_1, a_2, \cdots, a_n)} \tag{13}$$

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 $A\boldsymbol{p}$ の計算を，桁数の少ないベクトル $\boldsymbol{p}_s$ によって計算することができる[47].

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

[46] 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.

[47] 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.

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 (13) 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場合がある．

行列に対して同様のスケーリングを行う関数が ScalMat であり，行列の特定の列を選択的に更新する関数が ScalMatCol である．

```
void ScalMat(const rational_matrix& a, const int m, const int n, rational_vector& as, rational_matrix& aa);
rational ScalMatCol(rational_matrix& a, const int m, const int n, const int j, const int update);
```

これらの使用例は最小 2 乗法の例題 LLSschmidt.cpp を参照されたい．

桁数の表示ルーチン　桁数は次ようにして覗くことができる．ここでは

numdgts の使用例

```
rational_vector p(n+1);

polycofmax=0;
for(i=1; i<=nsize;i++) {
     longint xd = p[i].denominator();
     longint xn = p[i].numerator();
     if(numdgts(xd)+numdgts(xn)>polycofmax) polycofmax=numdgts(xd)+numdgts(xn);
}
```

関数 numdgts は longint クラスに含まれ，多桁数の 2^{32} 進数の桁数を返す．

行列の全要素の平均桁数を表示する MatDgtCount，行列の全要素の桁数を表示する MatDgtCountPrt，行列が三角行列の場合は上三角行列のみをカウントする MtrDgtCount，上三角行列の全要素の桁数を表示する MtrDgtCountPrt，ベクトルに対して行う VctDgtCount を用意した．

```
void MatDgtCount(const rat::matrix<rational>& a, const int m, const int n);
void MatDgtCountPrt(const rat::matrix<rational>& a, const int m, const int n);
void MtrDgtCount(const rat::matrix<rational>& a, const int n);
void MtrDgtCountPrt(const rat::matrix<rational>& a, const int n);
void VctDgtCount(const rat::vector<rational>& a, const int m);
```

これらの使用例は対称行列の三角分解の例題 LDL.cpp，dvsrch2.cpp を参照されたい．

行列の対角項を表示する SeeMatDiag は，対角行列の値と桁数を表示する．使用例は対称行列の三角分解の例題 LDL.cpp を参照されたい．

5.1.3　多項式演算ルーチン polydivchk, putpolynomial，polytexform など

有理数計算では数式処理が正確に行えるので，多項式の演算ルーチンを加えた．

有理数係数の多項式の格納形式は，高次の項を上（n 次多項式の n 次の項を配列要素 `p[n]` に，定数項を配列要素 `p[0]`）に格納する形式で除算を行う．

多項式の除算　polydivchk は int 型関数で，$u(x) = u_m x^m + u_{m-1}x^{m-1} + \cdots + u_0$ を $v(x) = v_n x^n + v_{n-1}x^{n-1} + \cdots + v_0$ で割って，商を配列 $q(x)$ に，剰余を配列 $r(x)$ に返し，割り切れた（剰余が零の）場合に 1 を，そうでない場合に 0 を関数値として返す．この関数については，講習会用の資料 dvsrch.pdf を参照されたい．

多項式の格納と取り出し　複数（npolynomial 個）の多項式の係数を 1 次元配列 cpolynomial に格納して，各多項式へのポインタを配列 ppolynomial が保存する．

多項式を格納する変数と配列の定義

```
int npolynomial=0;                /* number of polynomials found  */
int ppolynomial[n+2];             /* pointer to  polynomials      */
rational_vector cpolynomial(n*2); /* coefficients of polynomials  */
```

多項式を格納する putpolynomial と 取り出す getpolynomial 関数については，講習会用の資料 dvsrch.pdf を参照されたい．

多項式の LaTeX 形式での表示は，次のルーチンで行う．

polytexform

```
void polytexform(const rational_vector& p,const  int nsize)
{
   for (int i=0; i<=nsize; i++){
      if( p[i] > rational::ZERO ){
         cout << "+" << p[i] << "*x^" << nsize-i << " " << endl;
      }else{
         cout << p[i] << "*x^" << nsize-i << " " << endl;
      }
   }
   cout << endl;
}
```

これを用いると次のような出力が得られるので，コピー&ペーストで数式処理システムや LaTeX の原稿に入力することができる．この出力は 50 ページの 3.2.4 節に述べた出力 << で得られる．

polytexform の出力例

```
+1/1*x^2
-28823037615171176/1*x^1
+1817303900487189871746937021425 26/1*x^0
```

polytexformr は，多項式の係数が上と下で逆順の場合に使用する．polytexformint と polytexformintr は，多項式の係数が整数の場合に，分母の “/1” を付けないで表示する．

桁数の多い場合は，途中の桁を format 関数を使用して省略して表示する polytexformintfmt を使用するとよい．

polytexformintfmt

```
void polytexformintfmt(const rat::vector<rational>& p, const int nsize) {
  for (int i=0; i<=nsize; i++){
     if( p[i] > rational::ZERO ){
        std::cout << "+" << p[i].format(30,0) << "*x^{" << nsize-i << "} " << std::endl;
     }else{
        double pdble=-p[i]; if(log10(pdble)>30){
           std::cout << "-" << p[i].format(30,0) << "*x^{" << nsize-i << "} " << std::endl;
        }else{
           std::cout <<  p[i].format(30,0) << "*x^{" << nsize-i << "} " << std::endl;
        }
     }
  }
  std::cout << std::endl;
}
```

dvsrch2.cpp で，特性多項式を出力した例を示す.

```
+                           1.*x^{16}
          -430062106353116724.*x^{15}
+83553425 (19 digits) 36970780.*x^{14}
-97036891 (37 digits) 6917072.*x^{13}
+75137810 (53 digits) 21362960.*x^{12}
-40969626 (71 digits) 4212928.*x^{11}
+16197962 (88 digits) 3451584.*x^{10}
-4710791 (105 digits) 1402496.*x^{9}
+10114187 (121 digits) 4411264.*x^{8}
-1593492 (138 digits) 8609792.*x^{7}
+18114449 (153 digits) 2952448.*x^{6}
-1439127 (170 digits) 7851904.*x^{5}
+75444892 (184 digits) 0749824.*x^{4}
-2333230 (201 digits) 6658304.*x^{3}
+32081479 (215 digits) 0377600.*x^{2}
+19415442 (215 digits) 5427584.*x^{1}
-2839829 (215 digits) 5172992.*x^{0}
```

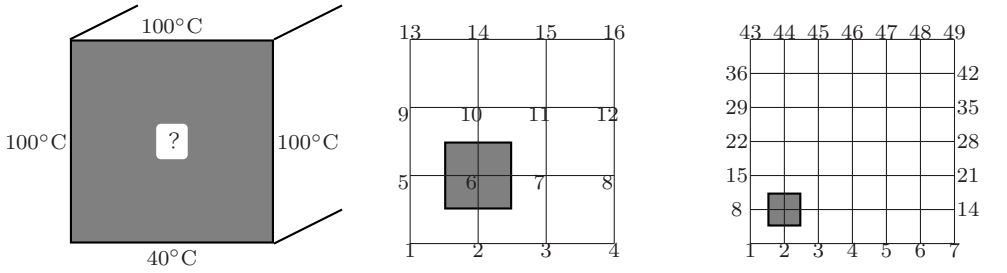
多項式関数の使用例は例題 dvsrch.cpp，dvsrch1.cpp，dvsrch2.cpp にある.

5.1.4 整数性に関する askinty 関数

浮動小数点数または有理数が ϵ の範囲内で整数に入るかどうかをチェックし，引数に近い整数として取り出して返す関数 `askinty` を用意した．整数性の関数と TEX 形式の出力の使用例は例題 dvsrch.cpp, dvsrch1.cpp，dvsrch2.cpp を参照されたい．また解説は，資料 dvsrch.pdf を参照されたい.

5.1.5 組合せ nxtcmb

数え上げのプログラミングで使用する組合せの数 ${}_nC_r$ を求めるルーチン `comb` と，実際に組合わせを作る r 重にネストしたループプログラムを 1 重にループ変換した際に使用する，次の組合せを返す `nxtcmb` 関数を rational クラスに含めた．これらの関数これらの関数については，資料 dvsrch.pdf を参照されたい.



(a) boundary condition (b) 3×3 mesh, 2×2 matrix (b) 6×6 mesh, 5×5 matrix

図 6: 四角柱の内部の温度分布

5.2 行列生成ルーチン

5.2.1 対称行列を生成する関数

対称行列を生成する関数を示す[48].

- フランク行列：SetFrank　$a_{ij} = n - \max(i, j) + 1$ である.
- ヒルベルト行列：SetHilbert　$a_{ij} = \dfrac{1}{i + j - 1}$ である.
- 熱伝導行列：SetFtherm である．解析領域の幾何学的な対称性のために係数行列は縮重しており，CG 法が少ない回数で収束し，また右辺依存性も確認できる.
- 正方格子に対するグラフラプラシアン行列：SetGLmat
- 乗算合同法による擬似乱数行列：SetRand
- 分母・分子が独立した乗算合同法による擬似乱数行列：SetRand2

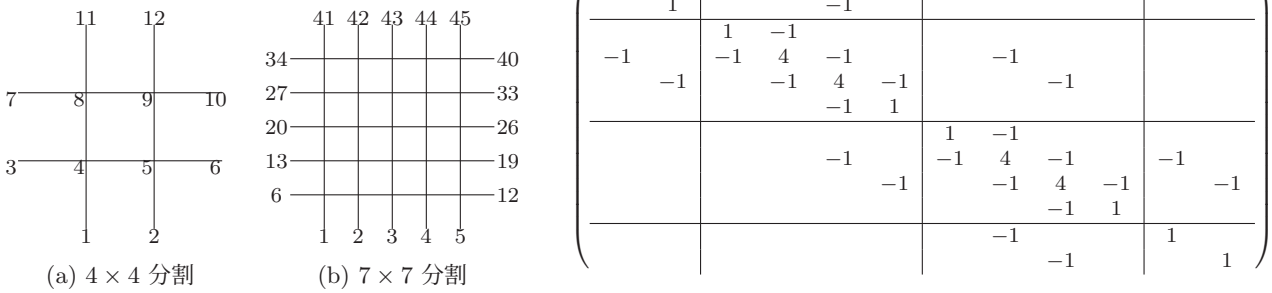
SetFrank，SetHilbert，SetFtherm は有理行列と倍精度行列を引数に指定できる（関数の多重定義）.

熱伝導行列は，正方形の断面をもつ四角柱で，表面の温度を与えられた場合の熱伝導問題を，断面領域で考える [4, p. 6]．図 6 に，内側のメッシュが 2×2 節点と 5×5 節点の場合を示す．メッシュ分割された中図と右図に 1 つの差分要素を示したが，上下左右の要素から温度勾配と熱伝導率に比例して熱が出入りする 5 点差分スキームを用いた．未知数は内側の節点にあるので，行列の行・列番号は，境界節点の番号を詰めて振られる．解析領域を $m \times m$ に分割すると，内部は $(m-1) \times (m-1)$ に詰められ，行列のサイズは $n = (m-1) \times (m-1)$ になる．3×3 のメッシュでは内側は 2×2 になるので，行列の行・列番号では節点 6 が最初の行になり，節点 7 が 2 番目，節点 10 が 3 番目，節点 11 が 4 番目になる．熱伝導係数を 1 にすると係数行列は次のようになる.

$$A = \begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix} \tag{14}$$

正方格子に対するグラフラプラシアン行列は，熱伝導行列の境界条件を外した特異行列である．正方格子に対するグラフ・ラプラシアン行列に，境界条件を設定することで得られる対称行列から，多重度の高い問題が得られる．12 節点に図 7 の左のように与えられた接続に対応するグラフ・ラプラシアン行列を図の右に示した．節点 i と j が接続すると，$a_{ii} = a_{jj} = 1$ で，$a_{ij} = a_{ji} = -1$ とするので，上下左右に接続をもつ節点の対角項は 4 になり，周辺の節点の対角項は 1 になる．この行列の周辺の節点を拘束する

[48] Symmetric の “S” と Set の “S” で記憶されたい.



(a) 4×4 分割　(b) 7×7 分割

図 7: グラフ・ラプラシアン行列

（取り除く）と，対角項が 4 の行と列だけの行列が残る．図の左での 4×4 の格子では，内側の 2×2 の節点だけが残り，次数 4 の対称行列が得られる．中図のように 7×7 の格子では，内側の 5×5 の節点だけが残り，次数 25 の対称行列が得られる．固有値の多重度は内側の分割数に一致し，$\lambda = 4$ の固有値が，左図の 2×2 では重根に，中図の 5×5 では 5 重根になる．

擬似乱数行列は，数の素性に注意が必要である．浮動小数点計算では，一様乱数ならどれでも計算時間に大差はないが，有理計算では乱数のもとになる整数の生成方法で桁数が大きく変わるからである．そこでシステムの提供する urand を使用すると，システムによって計算時間に変わり，比較できなくなる．そこで例題のプログラムでは，乗算合同法で SetRand をコーディングした．

```
 void SetRand2(rat::matrix<rational>& a, const int n, long long& iseed)
 {          // numerator and denominator are independent randum number.
   rational drandn, drandd;
   for (int i=0; i < n; ++i) {
      for (int j=0; j <= i; ++j) {
         MyRND(iseed,drandn);
         MyRND(iseed,drandd);
         a[i][j] = drandn / drandd;
         a[j][i] = a[i][j];
      }
   }
 }

void SetRand(rational_matrix& a, int n, long long& iseed){
   double aij;
   rational drand;
   for (int i=0; i < n; ++i) {
      for (int j=0; j <= i; ++j) {
         MyRND(iseed,drand);
         a[i][j] = drand; a[j][i] = a[i][j];
      }
   }
}

void MyRND(long long& iseed, rational& drand){
   MyIRND(iseed);
   longint n = (uint64_t) iseed;
   longint d = (uint64_t) 2147483647L;
   drand.set(n,d);
}

void MyIRND(long long& iseed){
```

```
    long long k = iseed*16807;
    iseed = k % 2147483647;
}
```

これらの使用例は，LDL.cpp を参照されたい．

5.2.2 非対称行列を生成する関数

連立 1 次方程式の例題 LU.cpp および LUhmg.cpp，また最小 2 乗法の例題 LLSschmidt.cpp で使用している．

GenRand 関数は $n \times n$ の行列に，階数 r の行列を生成する．擬似乱数によって $r \times r$ の行列を生成し，$r+1$ から n 列に，r 本の線形独立な列ベクトルに従属する $n-r$ 本の列ベクトルを生成する．

```
void GenRand(rational_matrix& a, const int n, const int r, long long& iseed){
   rational drand;
   for (int i=0; i < n; ++i){
      for (int j=0; j < r; ++j){
         MyRND(iseed,drand);   // rational class routine
         a[i][j] = drand;
      }
   }
   for (int j=r; j < n; ++j){              // make column j
      for (int i=0; i < n; ++i){      // lineary dependent
         a[i][j]=rational::ZERO; // from previous columns
         for (int k=0; k < n; ++k) a[i][j]=a[i][j]+a[i][k];
      }
   }
}
```

GenMat 関数は $m \times m$ の行列の主小行列に，階数 r の $n \times n$ 行列を生成し，外側を擬似乱数で埋める．したがって $m=4$，$n=3$，$r=1$ で使用すると，3×3 で階数 1 の行列の外側を乱数で埋めて，階数 3 の行列が生成される．この例題は，同次連立 1 次方程式を解く LUhmg.cpp を参照されたい．

5.3 数値線形代数

5.3.1 ガウス消去法 LUdecomp，LUsubst, solhmg

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 つの小行列を分離して，正確に階数を得られない場合がある．正確な計算の利点を活かして，完全軸選択（complete pivoting）で LU 分解を行う LUdecomp と，対応する代入計算 LUsubst を用意した[49]．

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);
```

[49] 例題 LUhmg.cpp は，特異行列に対して，階数を調べて，同次連立 1 次方程式の非自明解を求める．LU.cpp は正則行列を解く．行列の生成は `GenMat` 関数を使用している．また，対称行列の固有値を求める例題 dvsrch1.cpp，dvsrch2.cpp で，有理数の固有値に対応する固有ベクトルを求める例も含めた．

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．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 rscal$(\alpha, \boldsymbol{x})$ によって行う[50]．

```
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[51]．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 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．

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 に与え，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，$P\boldsymbol{b}$ と最初に交換してから，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（次章に示す）．

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．

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{15}$$

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

$$\boldsymbol{x}_1 = -A_{11}^{-1}A_{12}\boldsymbol{x}_2 \tag{16}$$

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．

[50] 倍精度浮動小数点計算で BLAS の dscal に対応する．
[51] 倍精度浮動小数点計算で BLAS の dger に対応する．

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．

$$\boldsymbol{x}=\begin{pmatrix} -A_{11}^{-1}A_{12} \\ 1 \end{pmatrix} \tag{17}$$

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 (16) 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．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．

5.3.2 対称行列のガウス消去法 LDL，LDLsubst

対称行列の擬似コレスキー分解は LDL，対応する代入ルーチンは LDLsubst とした[52]．

LDL 分解　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，

$$A=LDL^T \tag{18}$$

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．

$$u_{ij}=a_{ij}-\sum_{k=1}^{i-1}t_{ki}u_{kj}, \qquad (i\le j) \tag{19}$$

$$t_{ij}=\frac{u_{ij}}{u_{ii}} \qquad (i<j) \tag{20}$$

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];          下三角の要素を転置位置に作り
            s = s + t * a[k][j];          上三角の要素と掛けて内積に足し込み
            a[k][j] = t;                  上三角要素の位置に置く
        }
```

[52] LDL.cpp に，4 通りの対称行列の生成ルーチンから行列を選択して生成し，対称行列を係数行列とする連立 1 次方程式を解いて解を求める例題を示した．この例題には，プロファイルの取得の EPROF2 の指定を含めた．

```
        a[j][j] = a[j][j] - s;              内積を引いて対角項を求める
    }
}
```

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 [4]．アルゴリズムは，軸列（第 j 列）の分解と，更新されたベクトルによる行列ベクトル積による更新よりなる[53]．

第 j 列（軸列）の分解には依存性があり，このままでは並列化できない．依存性は，対角項 a_{kk} で割った下三角の要素 l_{ik} を，割られる前の上三角の要素 u_{kj} の積を内積変数 s に足しこんでから，上三角の要素 u_{kj} を t で置換えるところにある．並列化のために一時ベクトル w[0:n] を使用した．

LDL 分解の軸列の分解

```
rat::vector<rational> w(n);  // added for parallelization
for (j=1; j < n; ++j) {
   rat::lvector_section<rational> ws1(w,0,j-1,1);
   rat::rvdivs(a.diagonal(),a.column(j,0,j-1,1),ws1);
   a[j][j] = a[j][j] - rat::rdot(w.section(0,j-1,1),a.column(j,0,j-1,1));
   rat::lmatrix_column<rational> acol1(a,j,0,j-1,1);
   rat::rcopy(w.section(0,j-1,1),acol1);
```

rvdivs は 3 項のベクトルの除算 $z_i = y_i \div x_i$，rdot は内積を計算し，rcopy はベクトルのコピーを行う．軸列の分解ができると，その列ベクトルと，小行列（1 から i 行，$i+1$ 列から n 列）の転置の積を，第 i 行ベクトルの i から n 番目の要素から引く．転置行列とベクトルの積和 $\boldsymbol{y} \leftarrow \alpha A^T \boldsymbol{x} + \beta \boldsymbol{y}$ は rgemtv$(\alpha, A, \boldsymbol{x}, \beta, \boldsymbol{y})$ によって行う．コメントで rblas の関数を使用する前のコードを残した．

LDL 分解の転置行列ベクトル積による更新計算

```
   for (i=1; i < n; ++i) {
//    for (j=i+1; j < n; ++j) {
//       for (k=0; k < i; ++k)  a[i][j] = a[i][j] - a[k][i] * a[k][j];
//    }
      rat::lmatrix_row<rational> arow1(a,i,i+1,n-1,1);
      rat::rgemtv(-rational::ONE,a.section(0,i-1,1,i+1,n-1,1),a.column(i,0,i-1,1),
                  rational::ONE,arow1);
   }
```

なお，特異性のチェック（軸選択）は省略してあるので，特異行列を与えると問題が発生する．

代入ルーチン　LDLsubst で代入計算を行う．

```
void LDLsubst(const rational_matrix& a, int n, rational_vector& x)
```

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．対角項による除算は rvdiv によって 2 項のベクトルの除算 $y_i = y_i \div x_i$ によって行う．

5.3.3　ヘッセンベルグ変換 elmhes

ヘッセンベルグ変換と次項のフロベニウス変換のサンプルプログラムは HesFrb.BAS を参照されたい．

[53] 倍精度浮動小数点計算の BLAS-1 を用いた Fortran プログラムでは，“//” で印を付けた 3 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．

次数 5 のヘッセンベルグ行列を示す.

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} \\ k_2 & h_{22} & h_{23} & h_{24} & h_{25} \\ & k_3 & h_{33} & h_{34} & h_{35} \\ & & k_4 & h_{44} & h_{45} \\ & & & k_5 & h_{55} \end{pmatrix}. \tag{21}$$

対角項の下の副対角項 $h_{s+1,s}$ を k_{s+1} で表す．ヘッセンベルグ行列への変換アルゴリズムは次のようになる [5, p. 478];

第 s 段階までに，もとの行列 $A \equiv A_1$ は A_s になっており，その最初の $(s-1)$ 行および列は上ヘッセンベルグ形である．第 s 段階は次の一連の操作からなる．

- 第 s 列の対角要素より下側で，絶対値最大の要素を捜す．それが零なら，次の 2 項目を飛ばし，この段階は終わる．そうでないなら，その最大要素のあった行の番号を s' と置く．
- 第 s' 行と第 $(s+1)$ 行を交換する．これは軸選択の手順である．この交換を相似変換にするため，列 s' と列 $(s+1)$ も交換する．
- $i = s+2, s+3, \ldots, n$ に対して，乗数

$$r_{i,s+1} = \frac{a_{is}}{a_{s+1,s}} \tag{22}$$

を計算する．行 $(s+1)$ に $r_{i,s+1}$ を掛けて行 i から引く．この消去を相似変換にするため，列 i に $r_{i,s+1}$ を掛けて列 $(s+1)$ に加える．

4×4 の例を示すと，第 1 段の消去は $a_{3,1}$ と $a_{4,1}$ を消去する．

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -r_{32} & 1 & 0 \\ 0 & -r_{42} & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & r_{32} & 1 & 0 \\ 0 & r_{42} & 0 & 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & a'_{13} & a'_{14} \\ k_2 & h_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{pmatrix} \tag{23}$$

第 s 列の要素 a_{is} を消去するために R の $s+1$ 列目に非零項を使うところが，ガウス消去法で用いる基本変換とは異なる．この段階（ステップ）を $(n-2)$ 回繰り返す.

プログラム例　elmhes は，$n \times n$ の有理数行列を与えると，基本変換を $n-2$ 回繰り返して，ヘッセンベルグ行列に変換する[54].

```
void elmhes(rational_matrix& a, const int n)
   rational x, y;
   int i;
   for (int m=2; m <= n-1; ++m) {
      x = rational::ZERO;
      i = m;
      for(int j=m; j<=n; ++j) {
         if(fabs(a[j-1][m-2]) > fabs(x)){
            x = a[j-1][m-2]; i = j;
         }
      }
      if (i != m) {
         for(int j=m-1; j<=n; ++j) {
```

[54] 使用例は dvsrch2.cpp を参照されたい.

```
            y = a[i-1][j-1]; a[i-1][j-1] = a[m-1][j-1]; a[m-1][j-1] = y;
         }
         for(int j=1; j<=n; ++j) {
            y = a[j-1][i-1]; a[j-1][i-1] = a[j-1][m-1]; a[j-1][m-1] = y;
         }
      }
      if (x != rational::ZERO) {
         for(int i=m+1; i<=n; ++i) {
            y = a[i-1][m-2];
            if (y != rational::ZERO) {
               y = y / x;
               a[i-1][m-2] = y;            変換を相似変換にするために 2 回行う.
               for(int j=m; j<=n; ++j) a[i-1][j-1] = a[i-1][j-1] - y * a[m-1][j-1];
               for(int j=1; j<=n; ++j) a[j-1][m-1] += y * a[j-1][i-1];
            }
            a[i-1][m-2] = rational::ZERO;
         }
      }
   }
}
```

rblas 化は次のように行う.

elmhes の基本変換

```
               for(int i=m+1; i<=n; ++i) {
                  y = a[i-1][m-2];
                  if (y != rational::ZERO) {
                     y = y / x;
                     a[i-1][m-2] = y;
      //             for(int j=m; j<=n; ++j)  a[i-1][j-1] = a[i-1][j-1] - y * a[m-1][j-1];
                     rat::lmatrix_row<rational> arow2(a,i-1,m-1,n-1,1);
                     rat::raxpy(-y,a.row(m-1,m-1,n-1,1),arow2);
      //             for(int j=1; j<=n; ++j)  a[j-1][m-1] += y * a[j-1][i-1];
                     rat::lmatrix_column<rational> acol2(a,m-1,0,n-1,1);
                     rat::raxpy(y,a.column(i-1,0,n-1,1),acol2);
```

どちらも raxpy で RBLAS 化できる.

なお，プリプロセッサに対する変数 SEEMATNUMDGTS を指定することで，ヘッセンベルグ変換の過程での行列要素の桁数を表示できる.

5.3.4 フロベニウス変換 hesfrb

hesfrb は，$n \times n$ のヘッセンベルグ行列を与えると，基本変換を繰り返すことでフロベニウス行列に変換する．**フロベニウス変換**は $(n-1)$ 段の主たるステップからなり，s 段のステップではヘッセンベルグ行列の第 s 列の対角項よりも上の要素を零にする．もとの行列は $H \equiv H_1$ とする [6, p. 406]．$n = 5$, $s = 3$, の場合には $R_3 H_3 R_3^{-1}$ を計算する[55]．この変換操作を $(n-1)$ 回繰り返し，相似変換を完結させる．行列

[55] $R_3 = \begin{pmatrix} 1 & & & -r_{14} & \\ & 1 & & -r_{24} & \\ & & 1 & -r_{34} & \\ & & & 1 & \\ & & & & 1 \end{pmatrix}$

は次の形になる．

$$X = \begin{pmatrix} 0 & 0 & 0 & 0 & x_0 \\ k_2 & 0 & 0 & 0 & x_1 \\ & k_3 & 0 & 0 & x_2 \\ & & k_4 & 0 & x_3 \\ & & & k_5 & x_4 \end{pmatrix}.$$

フロベニウス行列は対角行列を相似変換 $F = D^{-1}XD$ の形で掛けて得られる．ここに対角行列 D は

$$d_1 = 1,\ d_2 = k_2,\ d_3 = k_2k_3,\ d_4 = k_2k_3k_4,\ d_5 = k_2k_3k_4k_5 \tag{24}$$

である．D を右から掛ける操作は，1 列目はそのまま，2 列目は k_2 倍，3 列目は k_2k_3 倍，…，n 列目は $k_2k_3\cdots k_n$ 倍する．D^{-1} を左から掛ける操作は，1 行目はそのまま，2 行目は k_2 で割り，3 列目は k_2k_3 で割り，…，n 列目は $k_2k_3\cdots k_n$ で割る．したがって $p_0 = k_2k_3k_4k_5x_0$，$p_1 = k_3k_4k_5x_1$，$p_2 = k_4k_5x_2$，$p_3 = k_5x_3$，$p_4 = x_4$ になる[56]．

$$F = \begin{pmatrix} 0 & 0 & 0 & 0 & p_0 \\ 1 & 0 & 0 & 0 & p_1 \\ & 1 & 0 & 0 & p_2 \\ & & 1 & 0 & p_3 \\ & & & 1 & p_4 \end{pmatrix}, \qquad p_i = \left(\prod_{j=0}^{n-2-i} k_{n-j} \right) x_i. \tag{25}$$

次数 5 の特性多項式が $-\lambda^5 + p_4\lambda^4 + p_3\lambda^3 + p_2\lambda^2 + p_1\lambda + p_0$ として得られる．最高次の係数は “−1” であるが n が偶数の場合は “+1” で，次数 n の特性方程式は次式である[57]．

$$(-1)^n\lambda^n + p_{n-1}\lambda^{n-1} + p_{n-2}\lambda^{n-2} + \cdots + p_0 = 0$$

プログラム例　hesfrb は，$n \times n$ の有理数ヘッセンベルグ行列を与えると，基本変換を繰り返して，X 行列に変換し，さらにフロベニウス標準形に変換することで，多項式を得る[58]．

```
void hesfrb(rational_matrix& a, const int n, rational_vector& p) {
   rational t;
   int i, j, k;
   for (k=1; k <= n-1; ++k) {
      for (i=1; i <= k; ++i) {
         p[i-1] = a[i-1][k-1] / a[k][k-1]; a[i-1][k-1] = rational::ZERO;
      }
      for (j=k+1; j <= n; ++j) {
         for (i=1; i <= k; ++i) a[i-1][j-1] = a[i-1][j-1] - p[i-1] * a[k][j-1];
      }
      for (i=2; i <= k+1; ++i) a[i-1][k] += p[i-2] * a[i-1][i-2];
   }
   t = rational::ONE;
   for (i=n; i > 1; --i) {
      p[n-i+1] = a[i-1][n-1] * t;
      t = t * a[i-1][i-2];
   }
   p[n] = t * a[0][n-1];
   p[0] = -rational::ONE;
```

[56] 式 (25) の p_i は教科書の (52.6) 式とは異なる [6, p. 407].
[57] $(-1)^n\left(\lambda^n - p_{n-1}\lambda^{n-1} - p_{n-2}\lambda^{n-2} - \cdots - p_0\right) = 0$
[58] 使用例は dvsrch2.cpp を参照されたい．

```
   if(n%2 ==0) for (i=0; i <= n; ++i) p[i] = -p[i];
}
```

出力はヘッセンベルグ行列の特性多項式の係数が p に返る[59].

基本変換は raxpy によって RBLAS 化できるが，$X \to F$ 変換で用いる対角行列 D の要素は式 (24) の順序で計算すると依存性があり，逐次計算になる．次のプログラム例では，前半のループでは配列 p は一時配列として利用している．

RBLAS 化前の hesfrb ルーチン

```
void hesfrb(rational_matrix& a, int n, rational_vector& p){
    rational t;
    int i, j, k;
    for (k=1; k <= n-1; ++k) {
        for (i=1; i <= k; ++i) {
            p[i-1] = a[i-1][k-1] / a[k][k-1];
            a[i-1][k-1] = rational::ZERO;
        }
        for (j=k+1; j <= n; ++j) {
            for (i=1; i <= k; ++i) a[i-1][j-1] = a[i-1][j-1] - p[i-1] * a[k][j-1];
        }
        for (i=2; i <= k+1; ++i)  a[i-1][k] += p[i-2] * a[i-1][i-2];
    }
    t = rational::ONE;
    for (i=n; i > 1; --i) {              依存性のあるループ
        p[n-i+1] = a[i-1][n-1] * t;
        t = t * a[i-1][i-2];
    }
    p[n] = t * a[0][n-1];
    p[0] = -rational::ONE;
    if(n%2 ==0){                         偶数次の場合の符号反転
        for (i=0; i <= n; ++i) p[i] = -p[i];
    }
}
```

フロベニウス変換の前半 $H \to X$ では，ヘッセンベルグ行列の副対角項は不変である．これを 1 次元配列 sd にコピーしてから計算を行う．

上三角要素の消去の基本変換は raxpy で RBLAS 化できる．

hesfrb の基本変換

```
   rat::vector<rational> sd(n), tt(n+1);
   for (k=1; k <= n-1; ++k) sd[k]=a[k][k-1];
   sd[0]=rational::ONE;

   for (k=1; k <= n-1; ++k) {
//    for (i=1; i <= k; ++i)  p[i-1] = a[i-1][k-1] / a[k][k-1];
      rational akkm1=rational::ONE/a[k][k-1];
      rat::lvector_section<rational> ps0(p,0,k-1,1);
      rat::raxpy(akkm1,a.column(k-1,0,k-1,1),ps1);
      for (j=k+1; j <= n; ++j) {
//       for (i=1; i <= k; ++i)  a[i-1][j-1] = a[i-1][j-1] - p[i-1] * a[k][j-1];
         rat::lmatrix_column<rational> acol0(a,j-1,0,k-1,1);
         rat::raxpy(-a[k][j-1],p.section(0,k-1,1),acol0);
//       for (i=2; i <= k+1; ++i) a[i-1][k] += p[i-2] * sd[i-1];
         rat::lmatrix_column<rational> acol1(a,k,1,k,1);
         rat::rvma(p.section(0,k-1,1),sd.section(1,k,1),acol1);
```

このあと依存性のあるループ計算を挟み，多項式係数を算出する．

[59]使用例は dvsrch2.cpp を参照されたい．この例題は，対称行列 A の特性多項式（characterestic polynomial）を elmhes と hesfrb によって求めて，さらに特性多項式の因数分解を行い，行列の代数的構造を調べる．

hesfrb の多項式係数の計算

```
    t = rational::ONE;
    for (i=n; i > 1; --i) { tt[i] = t; t = t * a[i-1][i-2]; }
    tt[1] = t;
// for (i=2; i <= n; ++i)  p[n-i+1] = a[i-1][n-1] * tt[i];
    for (i=1; i < n; ++i)   p[i] = rational::ZERO;
    rat::lvector_section<rational> ps1(p,n-1,1,-1);
    rat::rvma(a.column(n-1,1,n-1,1),tt.section(2,n,1),ps1);
```

なお，プリプロセッサに対する変数 SEEMATNUMDGTS を指定することで，フロベニウス変換の過程での行列要素の桁数を表示できる．

5.4 連分数を使用する関数

連分数計算は有理算術演算の特長をよく表す．本節ではまずはじめに，正則連分数の例題を，1000 桁の π の連分数展開と $\sqrt{2}$ を求める関数を紹介する．次に，一般の連分数によって π を求めるプログラムと定数 π をセットする関数 setpi を紹介する．

5.4.1 正則連分数展開

1 より大きな実数 x の連分数展開は，$x_0 = x$ とおき，$i = 0, 1, 2, \cdots$ について「x_i の整数部を a_i とし，小数部を x_{i+1} とする」操作を x_{i+1} が整数になるまで繰り返す．

$$x_0 = a_0 + \frac{1}{x_1} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{x_2}} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{x_3}}} = \cdots \tag{26}$$

場所を節約した $x = [a_0, a_1, a_2, a_3, \cdots, a_{n-1}, x_n]$ の書き方も使用される．

x_{n+1} を無視して a_n までとると**第 n 近似分数**といい，ここでは $k_n = \dfrac{p_n}{q_n}$ と表記する．k_2 を示す．

$$k_2 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2}} = \frac{p_2}{q_2} \tag{27}$$

無理数を連分数展開すると無限に続くが，有理数は有限で止まる．これは分子が 1 の形の正則連分数展開（regular continued fraction expansion）が，2 つの自然数の最大公約数を求める GCD アルゴリズムの変形だからである．

高校の『数学 B』には [7]，自然数の除算 “$a \div b = q \ \cdots r$”，つまり “$a = bq + r$” により $\gcd(a, b) = \gcd(bq + r, b) = \gcd(b, r)$ を繰り返すユークリッドの互除法の BASIC プログラムが掲載されていた[60]．被除数を除数で，除数を剰余で置換えて割るので「互除法」という．“$a = bq + r$” を “$d_{-2} = a$” と “$d_{-1} = b$” と “$d_0 = r$” とおき漸化式

$$d_{i-2} = d_{i-1}q + d_i \tag{28}$$

[60] GCD アルゴリズムは日本語では「ユークリッドの互除法」と呼ばれるが，英語では Euclidian algorithm で,「の互除」は日本か中国の数学者が加えた意訳であろう．ユークリッドは定規とコンパスを用いて幾何学で解いたので，除算はない（除算は位取り記数法が発見された後に現れる）．アルゴリズムの基本は “a が b の倍数のとき $\gcd(a, b) = b$” と “$\gcd(a, b) = \gcd(a - b, b)$” にあって，“$\gcd(a, b) = \gcd(b, r)$” は後者を繰り返すことで得られる．2 進 GCD アルゴリズムは除算を使わない．

に書き直す．$i=0,1,2,\cdots$ について $d_i=0$ になるまで繰り返すと，最後の d_{i-1} が $\gcd(a,b)$ として得られる．**漸化式 (28) の両辺を d_{i-1} で割り，分数 $\frac{d_{i-2}}{d_{i-1}}=x_i$ とおくと連分数展開の漸化式 $x_i=a_i+\frac{1}{x_{i+1}}$ が得られる**（商 q を a_i とした）．

π の 1000 桁の近似値が与えられたとき，その第 n 近似分数まで連分数展開するプログラム CfracPi.cpp を示す[61]．

```
int main(void){
    rational pi,p,r,x,y;
    uint32_t i,n,m,mm;
    std::cout << "Enter n" << std::endl;
    std::cin >> n;
    longint pin("31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679 略 ");
    longint pid("10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 略 ");
    rational pi1000(pin,pid);                  // 1000 桁の \pi の近似値
    rat::vector<rational> a(n+1);
    x=pi1000;
    p = x.rational::floor();                   //  x_0 の整数部を p に
    a[0] = p;
#ifdef LGCD1
    longint::lgcd_p=longint::lgcd1;
#endif
    m=n;
    for(i=1; i<=n; i++){
        y=x-p;                                 //  x_{i-1} の小数部を y に
        if(y==rational::ZERO){m=i-1;break;}    //  y がゼロなら break
        x = y.inv();                           //  y の逆数を x_i に
        p = x.rational::floor();               //  x_i の整数部を p に
        a[i] = p;                              //  a[i] に p を格納
    }
    r=a[m];
    for(i=m-1;i>0;i--)r=a[i]+rational::ONE/r;
    pi=a[0]+rational::ONE/r;
    cout << "pi="<< pi.format(n+10,n)<< endl;
    return 1;
}
```

有理数 x の整数部を `floor` 関数で変数 `p` に得る．前半のループが連分数展開によって式 (26) の a_i を rational 型の配列 `a[i]` に作成する（`x-p` で有理数型変数 `y` に小数部分を得て，その逆数を `inv` 関数で得て[62]，x_i として，その整数部分が連分数の第 i 項になるので `a[i]` に格納する）．`y` が零なら，展開は終わり，ループを出る（break する）．

後半のループは最終項 a_n から降順（$i=n,n-1,\cdots,2,1$）に連分数 a_i から第 n 近似分数 k_n を得る．$\pi-3$ の最良近似分数は $\frac{1}{7}$，$\frac{15}{106}$，$\frac{16}{113}$，$\frac{4687}{33102}$，$\cdots$ となる．

`pi` の表示は `format` 関数に桁数を指定して書くが[63]，ここでは小数点以下 `n` 桁とした．

正則連分数はユークリッドの互除法と同じなので，連続する 2 つの近似分数 k_{i-1} と k_i によって，被近似数を包含（inclusion）できる．証明は教科書 [8, p. 46]，または資料 sighpc2016r.pdf を参照されたい．したがって，この**アルゴリズム自身が解を挟み込む性質によって，自然に区間算術演算が達成される**．

$$\frac{q_2}{p_2}<\frac{q_4}{p_4}<\frac{q_6}{p_6}<\cdots<\pi<\cdots<\frac{q_5}{p_5}<\frac{q_3}{p_3}<\frac{q_1}{p_1}$$

`pi1000` が 10 進数で 1000 桁の精度であるが，連分数展開で得られたこの数値に対する最良近似分数の分母も分子も 104 桁である．なお，π の 1000 桁の近似値は有理数なので，連分数展開は有限回（1997 回）

[61] $\pi\fallingdotseq 3.14159$ とすれば，$3.14159=\frac{314159}{100000}$ と表す．

[62] 逆数は 1 を割ってもできるが，GCD 計算は不要なので inv 関数を使ったほうがよい．

[63] `format` 関数は指定された小数点以下の桁数まで 10 進小数で表示する．

の反復で停止する．第 n 近似分数 $\frac{p_n}{q_n}$ の誤差は，$\frac{1}{q_n^2}$ より小さく，分母の大きさの割に高精度である．これは，$\left|x-\frac{q_n}{p_n}\right| < \left|\frac{q_{n+1}}{p_{n+1}}-\frac{q_n}{p_n}\right|$ の右辺が $\frac{1}{p_n p_{n+1}}$ になることを示せばよい [8, p. 48]．

正則連分数は GCD アルゴリズムと同等なので，変数 y の有理数は既約である [9]．したがって `LGCD1` を指定してコンパイルすると，不必要な GCD 計算を行わないので格段に高速化される．

5.4.2 連分数展開による sqrt2 関数

有理数は分母を 2 のべき乗にすることが多いので，引数が 2 のべき乗の場合は別処理した．アルゴリズムは $\sqrt{n}$ の（正則）連分数展開が

$$\sqrt{n} = [a_0, \dot{a_1}, a_2, \cdots, 2\dot{a_0}].$$

のように循環節（repetend）をもつ（$\sqrt{2} = [1, \dot{2}]$）ことを使用して，

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \ddots}}}} \tag{29}$$

を計算する．正則連分数の第 i 近似分数 $\frac{p_i}{q_i}$ は，(後述する）3 項漸化式 (33) から計算できる．また正則連分数の第 i 近似分数 $\frac{p_i}{q_i}$ の誤差が，$\frac{1}{q_i^2}$ より小さいことを利用して，分母 q_i のビット数から判定することで，rational 型の数の除算 $\frac{p_i}{q_i}$ を避けた[64]．なお，正則連分数展開で得られる近似分数の分母と分子は互いに素であるため，最後の rational 型の数も RRset でセットして GCD 計算は避ける．

ratutil.cpp の sqrt2 関数

```
rational sqrt2(longint X, uint32_t tol){
   longint p0,p1,p,q0,q1,q;
   p0=longint::ONE;   q0=longint::ONE;
   p1=longint::THREE; q1=longint::TWO;
   while(true){
      p=longint::TWO*p1+p0; q=longint::TWO*q1+q0;          // 3 項漸化式
      int nbq=numbits(q);                                  // q_i の 2 進桁数
      if((uint32_t)(2*(double)nbq*0.30103) > tol)  break; // 10 進変換して (q_i)^2 収束判定
      p0=p1; q0=q1; p1=p; q1=q;
   }
   rational r=RRset(p,q); return r;  正則連分数の p_i と q_i は互いに素なので RRset 使用
}
```

[64] 数学的な証明などは sighpc2016r.pdf を参照されたい．

5.4.3 円周率の計算

π を挟む区間の上下限を有理数で求める．$\arctan x$ の連分数展開は次式で表される [8, p. 51].

$$\arctan x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cfrac{b_5}{a_5 + \ddots}}}}} = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{(2x)^2}{5 + \cfrac{(3x)^2}{\ddots \cfrac{}{2n-1 + \cfrac{(nx)^2}{\ddots}}}}}} \tag{30}$$

式 (30) は，べき級数によって定義されるガウス超幾何関数によって表された arctan を連分数展開して得られる[65].

$x = 1$ とおくと $\dfrac{\pi}{4}$ が得られ,

$$\frac{\pi}{4} = \cfrac{1}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{7 + \cfrac{4^2}{\underbrace{9 + \underbrace{\ddots}_{v_5}}_{v_4}}}}}} \tag{31}$$

4 倍すると π が得られる．

式 (30) の $b_i = 1$ の場合を**正則連分数**，そうでない場合を**一般の連分数**という．連分数の第 i 近似分数は $\dfrac{p_i}{q_i}$ であるが，その分母と分子は次の 3 項漸化式で得られる（上が一般の，下が正則連分数）.

$$p_i = a_i p_{i-1} + b_i p_{i-2}, \quad q_i = a_i q_{i-1} + b_i q_{i-2} \tag{32}$$

$$p_i = a_i p_{i-1} + p_{i-2}, \quad q_i = a_i q_{i-1} + q_{i-2} \tag{33}$$

連分数の 3 項漸化式によって π を計算する方法を考える．はじめに末端の分数を v_n と置いて，**降順** $i = n, n-1, \cdots, 2$ **に反復計算**する方法を示す．式 (31) で，$v_5 = \ddots$，$v_4 = \dfrac{4^2}{9 + \ddots}$ とおけば，第 n 近似分数を求める計算は

$$v_{i-1} = \frac{i^2}{(2i+1) + v_i} \tag{34}$$

を i が $n-1$ から 2 まで反復して，最後に 4 倍すればよいことがわかる．

具体的に第 2 近似分数の計算例を示す.

$$k_2 = \cfrac{4}{a_1 + \cfrac{1^2}{a_2 + \cfrac{2^2}{5}}} = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5}}} = \cfrac{4}{1 + \cfrac{1^2 \cdot 5}{3 \cdot 5 + 2^2}} = \frac{4 \cdot 19}{19 + 5} = \frac{76}{24} = 3.16666\cdots$$

[65] 連続する 2 つの近似分数 k_{i-1} と k_i によって，被近似数を包含（inclusion）できることと，ガウス超幾何関数による逆正接関数の連分数展開については，資料 sighpc2016r.pdf『有理算術演算における最大公約数計算について』を参照されたい．資料の付録に「ガウス超幾何関数による逆正接関数の連分数展開」を記した.

第 3 近似分数の計算例を示す.

$$k_3 = \cfrac{4}{a_1 + \cfrac{1^2}{a_2 + \cfrac{2^2}{a_3 + \cfrac{3^2}{7}}}} = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{7}}}} = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2 \cdot 7}{35 + 3^2}}} = \cfrac{4}{1 + \cfrac{1^2 \cdot 44}{3 \cdot 44 + 28}} = \frac{160}{51} = 3.1372549\cdots$$

第 2 近似分数 k_2 は π より大きく，第 3 近似分数 k_3 は π より小さい．$k_0 = 4$, $k_1 = 3$ とおくと，k_i は i が偶数のとき π より大きく，奇数のとき π より小さいことが分かる．なお，第 n 近似分数，例えば $\frac{76}{24}$ も，途中に現れる分数，例えば $\frac{28}{44}$ なども既約でないところが正則連分数と異なる．

3 項漸化式による昇順反復計算　分母と分子を独立に計算すると，それぞれは自然数なので，rational 型の変数を使っても，GCD 計算は行わない.

```
int main() {
   uint32_t n;
   std::cout << "input n=" << std::endl;
   std::cin >> n ;
   rational p,q,p0,q0,p1,q1,u,v,x;
   interval pi;
   long long tstart0=gettime();
#ifdef LGCD1
      longint::lgcd_p = longint::lgcd1;
#endif
   p0=rational::FOUR; q0=rational::ONE;
   p1=rational::TEN+rational::TWO; q1=rational::FOUR;
   for(uint32_t i=2; i<=n; i++){
      x=i;
      p=(rational::TWO*x+rational::ONE)*p1+(x*x)*p0;
      q=(rational::TWO*x+rational::ONE)*q1+(x*x)*q0;
      p0=p1; q0=q1; p1=p;  q1=q;
   }
   u=p0/q0; v=p1/q1;
   if(u<v){
      pi=interval(u,v);
   }else{
      pi=interval(v,u);
   }
   std::cout << " pi="; formatprn(pi);
   return 1;
}
```

`formatprn` 関数によって，$n = 2000$ を与えた場合の結果を示す[66].

2000 項で計算した結果

```
3.14159265358979323 (1504 digits) 236480665088297165
3.14159265358979323 (1504 digits) 236480665569605919
```

$n = 2000$ はすぐに計算が終了するが，10 倍の $n = 20000$ 項で計算すると，パソコンでは反復に 10 秒ほどかかる．`-DLGCD1` を指定してコンパイルすると，反復後の 2 つの除算 `u=p0/q0` と `v=p1/q1` で約分しないので，桁数が大きいままで，その後の formatprn で計算時間を要する[67]．2000 の場合は，p も q も 2^{32} 進数で 677 桁の，合計 1354 桁が，約分された分数では 400 桁に落ちる．20000 の場合は，8825 桁の，

[66] `formatprn` 関数は，interval 型の変数の上限と下限の差から，10 進数での小数点以下の値の違う位置を調べて，その桁以下数桁までを表示する．このとき同じ数は省略し，省略した桁数を表示する．

[67] formatprn では内部で `lgcd_p=lgcd1` を指定しているが，引数の桁数が大きくなっているので遅くなる．

合計 17650 桁が，約分された分数では 4014 桁に落ちる．この桁数の削減をしないで formatprn で 10 進変換すると遅くなるため，反復計算で GCD を省略しても，2 つの除算では GCD 計算は省略しないほうがよい．

降順反復計算　n を与えられたら，第 n 近似分数の初項に最初の反復を行った中間解と，第 $(n-1)$ 近似分数の初項を interval 型の変数に作り，ループ添字 i を $n-2$ から 1 まで降順に反復する．

第 n 近似分数 k_n の初項は $v_n = \dfrac{n^2}{2n+1}$ であるから，反復計算式 (34) から，$v_{n-1} = a_{n-1} + \dfrac{n^2}{2n+1} = 2n-1+\dfrac{n^2}{2n+1}$ から始めて降順に反復する．最初の反復を行うと v_{n-1} は $v_{n-2} = \dfrac{(n-1)^2(2n+1)}{n^2+(2n+1)(2n-1)}$ になる．一方，第 $(n-1)$ 近似分数の初項は $v_{n-1} = \dfrac{(n-1)^2}{2(n-1)+1}$ である．両者を有理数型の変数 u と t に作成したら，大小比較して区間型の変数 v の下限と上限として格納する．ループの添字 i を $(n-2)$ から 1 まで反復し，各反復では「これに $(2i+1)$ を加えてから i^2 を割る」．ループ内の z= $\cdots$ と v= $\cdots$ のステートメントは，interval 型と rational 型の算術演算なので，1 ステートメントで上限と下限に対して演算を行うプログラム PiCfracAtanRint.cpp を示す．

```
        uint32_t n;
        double k;
        std::cout << "input n=" << std::endl;
        std::cin >> n ;
        std::cout << "input k=" << std::endl;
        std::cin >> k ;                      パラメータ k を与える
        rational p,q,t,r,s,u,x,xx,tmp;
        rational pit,piu;
        long long tstart0=gettime();
    #ifdef LGCD1
        std::cout << "LGCD1 defined.  USE lgcd1 for lgcd_p" << std::endl;
        longint::lgcd_p = longint::lgcd1;
    #endif
        p=n*n; q=2*n+1; tmp=2u*(n-1u)+1u; t=p+q*tmp;           //  n^2+(2n+1)*(2n-1)
        tmp=(n-1u)*(n-1u);
        p=q*tmp;         q=t;            t=p/q; // t = (2n+1)*(n-1)^2 / (n^2+(2n+1)*(2n-1))
        r=(n-1u)*(n-1u); s=2u*(n-1u)+1u; u=r/s; // u = (n-1)^2 / 2(n-1)+1
        interval v;                             //  interval 型変数
        if( t >= u){
            v=interval(u,t);
        }else{
            v=interval(t,u);
        }
        rational vlow=v.lower();
        int lastdgts=getdgts(vlow);       //  set number of digits in lastdgts
        for(uint32_t i=n-2; i>0; i--){
            x=i; xx=x*x;
            interval z=v+(rational::TWO*x+rational::ONE);      // +(2i+1)
            interval w=z.inv();
            rational vlow=v.lower();  //  lower number picked for getdgts();
            if((double)getdgts(vlow)/(double)lastdgts >= k){   // check digits increase
                longint::lgcd_p = longint::lgcd2;
                v=w*xx;
                longint::lgcd_p = longint::lgcd1;
                vlow=v.lower(); lastdgts=getdgts(vlow); // set number of digits in lastdgts
            }else{
                v=w*xx;
            }
        }
        std::cout << "loop finish. time=" << (double)(gettime()-tstart0)/1000000.0 << "(sec)" << std::endl;
        //-----------------------------------------------------
        vlow=v.lower(); rational vup=v.upper();
        std::cout << "numdgts lower=" << getdgts(vlow) << " and  upper=" << getdgts(vup) << std::endl;
        //-----------------------------------------------------
```

```
    v=v+rational::ONE;
    interval y=v.inv();
    longint::lgcd_p = longint::lgcd2;  //    FORCE lgcd2 for *4 operation
    y=y*rational::FOUR;
    std::cout << " pi="; formatprn(y);
#ifdef LGCD1
    std::cout << "LGCD1 defined.  USE lgcd1 for lgcd_p" << std::endl;
    longint::lgcd_p = longint::lgcd1;
#endif
    rational vl=y.lower();
    std::cout << " num dgts of pi (after lgcd2) =" << getdgts(vl) << std::endl;
    std::cout << "total time=" << (double)(gettime()-tstart0)/1000000.0 << "(sec)" << std::endl;
    return 1;
}
```

この計算も，毎回 GCD を計算すると遅くなる．連分数から近似分数を求める計算は，桁数の小さな自然数の演算であり，これに対し GCD 計算がはるかに重い演算だからである．

そこで適当なパラメータ k を与えて，桁数の増加がこのパラメータを越えたときだけ lgcd2 関数で変数 v を既約にする．getdgts 関数は，与えられた有理数の分母と分子の 2^{32} 進数での桁数の和を返す．π の近似値は interval 型の変数 v に得られる．その下限を有理数型の変数 vlow に取り出し，桁数を lastdgts に記憶する．反復で桁数は増加していくが，最後に既約にした状態から k 倍になったら約分する．

実測した範囲では $k = 1.8$ の近傍が適当であるようだ．$n = 20000$ の例では，$k = 1.8$ の場合は 20000 回中 21 回 GCD を求めて約分する．

区間幅を指定し昇順に計算　π は無理数なので，π を含む区間の幅を指定して，反復がその区間幅まで狭められたらその区間を返すように実装することもできる．

5.4.4　定数 π を指定幅の区間で返す関数 setpi

本項の関数は，区間の幅を指定されたら，その精度が得られるまでループを昇順に計算するプログラムを longint 型で関数化したものである．setpi 関数は，π を区間幅が 10^{-tol} の interval 型の数で返す．

次のプログラム setpi 関数では，係数 a_i と b_i は longint 型で計算し，分数 $\frac{p_i}{q_i}$ を作るところで rational 型にしている．ここで GCD 計算をすると遅くなるので，GCD 計算は lgcd_p を lgcd1 にすることで省略している．

```
interval setpi(const uint32_t tol){
   uint32_t i=1;
   longint p,q,p0,q0,p1,q1,x;
   rational pp,qq,pq,pq0;
   interval pi;
   longint::lgcd_p = longint::lgcd1;                          GCD 計算の省略指定
   p=longint::TEN; for(int j=1; j<tol; j++) p=p*longint::TEN;
   rational rtol(longint::ONE,p);
   p0=longint::FOUR; q0=longint::ONE;
   p1=longint::TEN+longint::TWO; q1=longint::FOUR;
   while(true){
      i++; x=i;
      p=(longint::TWO*x+longint::ONE)*p1+(x*x)*p0;            3 項漸化式
      q=(longint::TWO*x+longint::ONE)*q1+(x*x)*q0;
      pp=RRset(p,longint::ONE); qq=RRset(q,longint::ONE);
      pq=pp/qq;
      if(i>3){ if((pq0-pq).abs() < rtol) break; }             区間幅から収束判定する
      pq0=pq; p0=p1; q0=q1; p1=p;  q1=q;
   }
   longint::lgcd_p = longint::lgcd2;                          GCD 計算の指定
```

```
    pq0=pq0*rational::ONE; pq=pq*rational::ONE;              約分する
    if(pq0 < pq){
       pi=interval(pq0,pq);
    }else{
       pi=interval(pq,pq0);
    }
    return pi;
}
```

関数の最後で lgcd_p=lgcd2 としているので，lgcd2 以外で GCD を計算する場合は，プログラムの先頭で setpi を使用する必要がある．

5.4.5 最良近似分数 bestappfrac

この関数については，資料 dvsrch.pdf を参照されたい．

5.5 平方根

平方根は無理数なので，有理算術演算でも正確に計算することができない．自然数 a を開平するには，浮動小数点演算ではニュートン法を使用することが多いが，有理算術演算で桁数の多い上下限値で包含した区間を求めるには適さない．この理由は，高精度の初期値の設定が困難で，ニュートン法反復が増えるためである．ここでは，有理数の分母や分子の自然数に対し，(小学生時代からお馴染みの）筆算による開平演算（digit by digit algorithm）を改良した sqrt4g 関数を使用する．SQR 関数は，rational 型の引数 a に対し，$\sqrt{a}$ の近似値 s を rational 型で返し，SQRint 関数は interval 型の区間を返す．どちらも第 2 引数に精度 tol を，10 進数での桁数で指定する．SQR 関数の解 s は，$|\sqrt{a}-s| < 10^{-tol}$ であり，SQRint 関数の解 (s_l, s_u) は，$s_l < \sqrt{a} < s_u$ で，区間幅は 10^{-tol} 以下である．

5.4.1 開平法

SQR 関数が呼出す sqrt4g は，筆算で用いる開平を（10 進数に対してでなく），2^{32} 進数に対して行う．10 進数に対する**開平演算**は

$$(a+b)^2 = a^2 + 2ab + b^2 = a^2 + (2a+b)b \tag{35}$$

に基礎を置くもので，$a^2 + (2a+b)b$ の平方根 $a+b$ を次の形式で求める処理を再帰的に続ける．

$$\begin{array}{rrcrcr}
 & & & a & + & b \\
 & a & & a^2 & + & 2ab+b^2 \\
+ & a & & a^2 & & \\
\hline
 & 2a & + & b & & 2ab+b^2 \\
 & & & b & & 2ab+b^2 \\
\hline
 & & & & & 0
\end{array} \tag{36}$$

実際は位取り表記を行うので，$a = 10$ は 1 と書く．この方法で $\sqrt{2}$ から $1.414\cdots$ を求める手続きを示す．

$$
\begin{array}{cccccccc}
 & & & 1. & 4 & 1 & 4 \\
\cline{4-7}
1 & & & 2. & 00 & 00 & 00 \\
1 & & & 1 & & & \\
\hline
2 & 4 & & 1 & 00 & & \\
 & 4 & & & 96 & & \\
\hline
2 & 8 & 1 & & 4 & 00 & \\
 & & 1 & & 2 & 81 & \\
\hline
2 & 8 & 2 & 4 & 1 & 19 & 00 \\
 & & & 4 & 1 & 12 & 96 \\
\hline
2 & 8 & 2 & 8 & & 6 & 04 \\
 & & & \vdots & & & \vdots
\end{array}
\tag{37}
$$

C++ で，sqrtdec 関数を次のように書いてみる．ここでは根の小数点の上を求めるが，24，281，2824 などは変数 B に，100，400，11900 などは変数 A に，解は変数 R に現れる．関数に渡された X は，10 進数に変換した場合の桁数を求めるループが先頭にあり，次の根を 1 桁ずつ求めるループでは X から 10 進数で 2 桁に変換して取り出すために 10 のべき乗で割る必要がある[68]．

講習会用の SQRdev.cpp の sqrtdec 関数（前半）

```
rational sqrtdec(longint X, uint32_t n){  10 進の開平演算
   longint A,B,J,J1,P,Q,R,S,W,Y,DEN;  // educational purpose only
   uint64_t r; uint32_t i;
   int j, k, l, nd, ndd, ndrootx;
   T = longint::ONE; J = longint::ZERO;                              引数 X の
   nd=1;                                                             10 進数での
   while( T*J < A){ J += longint::ONE; T *= longint::TEN; nd++; }    桁数を調べる
   ndd=nd;                          // nd = ndd : number of decimal digits of X
   Y=X;                             // set initial Y =X
   ndrootx=(nd+1)/2;        // number of digits of sqrt(X), upper than decimal point.
   B = longint::ZERO; R = longint::ZERO;   B は左側の数, R は根 Root(X)
   for(i=ndrootx; i>0 ; i--){              解の i の位について反復する
      k=2;                                 X は 2 桁ずつ処理
      if( (nd % 2)==1 && (i==ndrootx) ) k=1;    X の桁数が奇数のとき，最初の桁は 1 桁処理
      T = longint::ONE;
      for(j=0; j<ndd-k; j++)  T *= longint::TEN;      //  T=10^(ndd-k)
      A = longint::ldivide(Y,T,&Q);              Y の上位 k 桁を A に取り出す
      P=B*longint::TEN;   // p=b*10
      J=longint::ZERO;    // j=0
      while( P*J < A ){ J+=longint::ONE; P+=longint::ONE; }  根の 1 桁を探す
      J-=longint::ONE;               i の位の数字 J
      R=R*longint::TEN+J;            既存の解を 1 桁ずらして J を入れる
      W=(B*longint::TEN+J)*J;        1 桁ずらして J を入れ A も更新する
      A=A-W;
      T = longint::ONE;
      for(l=1; l<i; l++) T *= longint::TEN; // T=10^(i-1)
      Y -= W*T*T;               // y=y-(b*10+j)*j*10^(i-1)*10^(i-1)
      B = B*longint::TEN+J+J;   // b=b*10+j+j, left portion
      ndd=ndd-k;
   }
```

このあと，必要な精度まで，小数点以下の演算を，同様のループ反復によって求める（省略）．

[68] ここでは簡単化のために，X の桁数が要求精度 n よりも多い場合も，小数点までは全桁を計算している．

5.5.2 基数変換した開平法による sqrt4g 関数

筆算では 10 進数で x の 2 桁から $\sqrt{x}$ の 1 桁を求める．2^{32} 進の longint 型の数を 10 進変換するには多くの時間を要するので，入力引数 X の 2^{32} 進数の 2 桁 から平方根の 1 桁を求めれば高速化される．この理由は，

- 入力引数 X の基数変換が不要になる，
- B などに 10 または 10 のべき乗を掛ける演算は，2 のべき乗進数では「語単位のシフト演算」で済む，
- 根の 1 桁を 10 進 から 2^{32} 進数にするので，ループの反復回数を $\frac{10}{2^{32}} \fallingdotseq 10^{-9}$ に減らせる，

からである．

しかし 10 進数の開平演算では，根の 1 桁を 1 から 9 までを試算して見つけられるが，2^{32} 進では 1 から 4G まで試算することになり，この方法をそのまま使用することはできない．そこでこれを「2 分法」と「根の 1 桁を予想する方法」に置き換える．

2 分法　$a = 0$ と $b = 4294967296$ から始めて，中点 $c = \frac{a+b}{2}$ で解がどちら側にあるかを調べる．$b - a$ は 4294967296，2147450880，10737254401，$\cdots$，4，2 と半分，半分になり，32 回目に 1 になって反復は終了する．a，b，c に 64 ビット変数 ja，jb，jc を使用すれば，最内側ループで多桁数の演算は式 (35) の $(2a + b)b$ にあたる算術演算 (P+JC)*JC と A との比較演算だけである．この演算を 32 回行うが，longint 型の変数 P と A を rough 型の変数 rp と ra に置換えて計算する．ただし (rp+jc)*jc と ra の比が 5% 未満の場合は longint 型で計算する．

根の 1 桁の予想　根の 1 桁 J は

$$(P + J) * J < A < (P + J + 1) * (J + 1) \tag{38}$$

を満たす．A と P の桁数が多くなると $P \gg J$ なので，$PJ \fallingdotseq A$ とおき，J の近似値を $\frac{A}{P}$ で得られる．変数 A や P を使用するのではなく，53 ビット精度の ra と rp を用いて計算するので，この予測値が当たっているかどうかを P の桁数で判断している．桁数がまだ少ない間は，2 分法を 32 回反復し，中間の場合は 5 回，十分多い場合は 2 回とした．

プログラムを示す．

ratutil.cpp の sqrt4g 関数（前半）

```
rational sqrt4g(longint X, uint32_t tol){   // called by SQR
   longint R, A, B, J;
   double da, dp;
   int pw, j, nit;
   uint32_t i,jc;
   uint64_t jb,ja,nden=0;
   uint32_t n=(uint32_t)((double)tol/4.816479930623699123419822315591 9)+1; // divide by LOG10(65536)
   if( (n-X.l)%2 == 1) n++;                  //////////// n-X.l even
   rough rcrit=rough::ONE/rough::TWENTY;
   int nxl=n-X.l;
   ////////////////////   A  の初期化  ///////////////
   if( X.l%2 == 1){                    桁数が奇数の場合
         A = longint::lset(X.d[X.l]);          最初の 1 桁
      }else{                          桁数が偶数の場合は 2 桁
         A = longint::lset(X.d[X.l])*longint::FOURGIGA+longint::lset(X.d[X.l-1]);
      }
   int uppoint=1;  // to control before or after the point
```

桁数が奇数か偶数かで最初の 1 桁の扱い方が異なる．また，小数点の上と下で 2 つのループであったもの

を，ループ添字 j でまとめて，j が正のときは小数点の上 j 桁目，負のときは小数点以下 $i+1$ 桁目を作成するように，2 つのループを 1 つにループ融合（loop fusion）した．

sqrt4g 関数の後半のループ部分を示す．

```
for(j=X.l; j>(-nxl); j-=2){      X の 2 桁と sqrt{X} の 1 桁を小数点の上と下について反復
    if(j>0){i=j;}else{i=-j;}     j が正のときは小数点の上，負のときは小数点以下
    if(j==0) uppoint=0;          制御変数
    rough rbj;
    rough ra=numeric_cast<rough>(A);      // A converted to rough number ra
    int pw2=ra.exponent(); double daa=ra.significand();
    if(j==X.l && X.l%2 == 1){     /////  adjust j in case of X.l odd
        ja=0; jb=65536; nit=16; B<<=16;
    }else{
        ja=0; jb=4294967296; nit=32; B<<=32;
    }
    rough rp=numeric_cast<rough>(B);  // B converted to rough number rp
    rough rr;
    if(rp == rzero){                P がゼロ（最初の桁）は sqrt(A) を使う
        if(pw2%2 == 0){
            double sqraa=sqrt(daa); rough rx(sqraa,pw2/2); rr=rx;
        }else{
            double sqra2=sqrt(daa*2); rough ry(sqra2,(pw2-1)/2); rr=ry;
        }
    }else{                           近似値を A/P で求める
        rr=ra/rp;
    }
    double drr=rr.significand()*pow(2.0,rr.exponent());
    uint64_t rjj=(uint64_t)drr;
    if(rp.exponent() <= 80){          P が 80 ビット以下は
        ja=0; jb=4294967296; nit=32;    2 分法反復 32 回
    }else if(rp.exponent() <= 160){   P が 160 ビット以下は
        ja=rjj-16; jb=rjj+16; nit=5;    2 分法反復 5 回
    }else{                            P が 160 ビットより多いときは
        ja=rjj-2; jb=rjj+2; nit=2;      2 分法反復 2 回
    }
    for(int k=0; k<nit; k++){       //////  iteration count is variable
        jc=(ja+jb)/2;
        rough rjc=numeric_cast<rough>(jc); // jc converted to rough number rjc
        rbj=(rp+rjc)*rjc;
        rough rval=rbj/ra;                A と P の比が 5% 以下か
        if(rval.abs() > rcrit){
            if((rp+rjc)*rjc > ra){ jb=jc;   rough 型で判定
            }else{ ja=jc;
            }
        }else{
            longint JC=longint::lset(jc);
            if( (B+JC)*JC > A ){ jb=jc;      longint 型で判定
            }else{ ja=jc;
            }
        }
    }
    J=longint::lset(ja);   J が決まる
    A -= ( B + J ) * J;    A の更新
    B += (J+J);            B の更新
    if(j<=0) nden++;       小数点以下の場合，分母を調整する桁数をインクリメント
    A <<= 64;              A を 2 語シフト
    if(j==X.l && X.l%2 == 1) j++;     最初の反復で桁数が奇数の場合の調整
    if(uppoint==1 && j > 3) A += longint::lset(X.d[j-2])*longint::FOURGIGA+ longint::lset(X.d[j-3]);
}
R=(B>>1);                     解は B / 2
```

```
    longint DEN=longint::ONE; DEN<<=(32*nden);
    rational tmp(R,DEN);     分母をセットして rational 型の解とする，GCD 処理あり
    return tmp;
}
```

5.5.3 SQR 関数

2 べきの数の平方根は，2 進数では最上位の 1 のあとに続く 0 ビットの数が偶数なら，1 のあとにその偶数の半分の数の 0 ビットを付けた数が平方根になる（$\sqrt{4}=2$ は $\sqrt{(100)_2}=(10)_2$，$\sqrt{16}=4$ は $\sqrt{(10000)_2}=(100)_2$，$\cdots$）．奇数の場合は，分母・分子ともに 2 倍して，0 ビットの数を偶数にする．

分母・分子ともに 2 べきでない場合は，要求精度から k を決め，多桁数の除算 $a_n(2^{32})^{2k} \div a_d$ の商を開平し[69]，結果を $(2^{32})^k$ で割る．この操作を「小数化」と呼ぶことにする．いずれの場合も多桁数の開平 1 回で有理数で平方根の近似値 s を指定された精度 $|\sqrt{a}-s|<10^{-tol}$ で得る．

5.5.4 SQRint 関数

SQR 関数で得られる a の平方根 $\sqrt{a}$ と，その近似値 s との大小関係は分らない．区間演算では包含（inclusion）の成立が重要で，このためには大小関係を知る必要がある．s^2 と a の比較をしないでよいように SQRint 関数を用意した．

sqrt4g 関数は解の下を求めるので，分母 2 べきの場合は下，分子が 2 べきの場合は上が返る．分母・分子ともに 2 べきでない場合は，小数化で剰余が 2G= 2^{31} 以上なら商 q が切り上げられるので上，2G よりも小さい場合は切捨てられるので下が返る．

以上の場合分けに従って，包含を満たす区間を得る．分母（分子）が 2 べきの場合は，rational 型の $\sqrt{a_n}$（$\sqrt{a_d}$）の近似値の分子（分母）に 1 を加えた値を分子（分母）とする有理数で包含する．分母・分子ともに 2 べきでない場合は，小数化で商を切捨てることで下を求め，分子に 1 を加えた有理数で包含する．多桁数の開平は 1 回，s^2 と a の比較なしで区間 (s_l, s_u) を指定された精度 $s_u-s_l<10^{-tol}$ で得る．

5.5.5 計測結果

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，メモリは DDR-3 を 32GB 搭載する．ここでは 1 スレッドのみを使用した．OS は Linux version 2.6.32，コンパイラは GNU gcc 4.4.7 である．$\sqrt{1.1}$ の SQR 関数による 10 万から 50 万桁の計算時間（秒）を示す．

処理	10 万桁	20 万桁	30 万桁	50 万桁
B << 32	0.03	0.14	0.31	1.41
2 分法反復	0.00	0.00	0.01	0.01
A, B 更新	0.24	0.98	2.12	7.14
A << 64	0.03	0.14	0.31	1.34
SQR	0.32	1.28	2.76	9.95
2 分法版	(5.83)	(22.5)	(49.5)	(171.6)

最下段に 2 分法版の時間を示した．rough 型を使用して解の予測を行うことで，多桁数 A の更新 A-(B+J)*J を主たる演算にでき，10 倍以上の高速化が実現された．J は 1 桁なので A-(B+J)*J には（1 回の減算は，

[69] 商は，余りによって丸める．

少ないほうの桁数を n として $O(n)$ なので）全体で 32 ビット符号なし演算で $O(n^2)$ の計算量になる[70]．根を 2^{32} 進数で 2 桁，あるいは 4 桁ずつ求めるようにすれば，A の更新回数が減り，さらに速くなる可能性が高い[71]．有理算術演算では，アルゴリズムの選択以上に，rough 型の設定のように，桁数が多く重い計算を回避する手段が重要である．

5.5.6 ガウス・ルジャンドル法による π の計算

GauLeg.cpp は平方根を使用して π を計算して，setpi 関数で得た π の近似値と比較する例題である．この計算方法は，完全楕円積分の第 1 種と第 2 種の公式に，ルジャンドルの関係式を連立させることで，π を未知数に回すことで導かれた．資料 GaussLeg2.pdf に，ガウスの公式とよばれる計算方法の数学の背景を記したので，参照されたい．

アルゴリズム 2 正数 a，b $(0 < b < a)$ をとり，

$$a_1 = \frac{a+b}{2}, \quad b_1 = \sqrt{ab},$$

以下順次

$$a_n = \frac{a_{n-1}+b_{n-1}}{2}, \quad b_n = \sqrt{a_{n-1}b_{n-1}}$$

とおくと，数列 $\{a_n\}$，$\{b_n\}$

$$b < b_1 < b_2 < \cdots < b_{n-1} < b_n < a_n < a_{n-1} < \cdots < a_2 < a_1 < a$$

はともに算術幾何平均 $M(a,b)$ に収束する．これを

$$\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n = M(a,b) = M$$

とおく．$a_0 = 1$，$b_0 = c_0 = \dfrac{1}{\sqrt{2}}$ ，$a_{n+1} = \dfrac{a_n + b_n}{2}$ ，$b_{n+1} = \sqrt{a_n b_n}$ ，$c_{n+1} = \dfrac{a_n - b_n}{2}$ ，$n = 1, 2, \ldots$ とすれば，ガウスの公式より

$$\pi = \frac{2M^2}{1 - \sum_{n=0}^{\infty} 2^n c_n^2} \tag{39}$$

により π を得られる．ただし，$M\left(1, \dfrac{1}{\sqrt{2}}\right) = M$ と書いた．

```
        rational a,b,p,t,x,y,sqrt2,by,zl,zu;
        interval z,Z;
        int i,tol, ttol;
        std::cin >> tol ;
        ttol=tol+tol/2;  // ttol=1.5*tol  used for SQR
        if(tol>99)  ttol=tol+tol/10;// ttol=1.1*tol used for SQR
        if(tol>1000) ttol=tol+tol/100;// ttol=1.01*tol used for SQR
        interval pii=setpi(ttol);
        std::cout << " pii=" << std::endl; formatprn(pii);
        rational pil=pii.lower(); rational piu=pii.upper(); rational pim=(pil+piu)/rational::TWO;
```

[70] A の更新は，$\sqrt{2}$ の 10 進計算の例では，$60400 - 28282 \cdot 2 = 2836$ を求めるので，借り（borrow）が上位桁に及ぶが，B の更新は下位の桁だけで済む．浮動小数点演算は同じ精度の 2 数に対して演算するので，四則演算の回数で計算時間を評価できるが，有理算術演算はオペランドの桁数を考慮して四則演算時間を評価する必要がある．ここでの A の更新と B の更新の時間は全く異なる．

[71] ニュートン法を有理数変数で書くと，各反復で有理数の乗算，除算が必要になり，GCD 計算も必要になるので，反復回数が少なくても，筆算アルゴリズムより時間がかかる．

```
    std::cout << " (double)pim=" << (double)pim << std::endl;
    a=rational::ONE;
    sqrt2=SQR(rational::TWO,ttol);
    b=rational::ONE/sqrt2;              //  b=sqrt2/rational::TWO;  初期値を有理化
    t=RRset(longint::ONE,longint::FOUR);
    x=rational::ONE;
    int nit=(int)log2((double)tol);    反復回数を log_2(tol) とする
    for(int i=0; i<=nit;++i){
       y=a;
       a=(a+b)/rational::TWO;
       by=b*y;
#ifdef SQRRATIONALB
       b=SQR(by,ttol);
#else
       z=SQRint(by,ttol);
       zl=z.lower(); zu=z.upper(); b=(zl+zu)/rational::TWO;
#endif
       t=t-x*(y-a)*(y-a);
       x=x*rational::TWO;
    }
    t=t-x*(y-a)*(y-a);
    p=((a+b)*(a+b))/(rational::FOUR*t);
    if(p>pim){
       interval ppi(pim,p); z=ppi;
       std::cout << " p is upper" << std::endl; formatprn(ppi);
    }else{
       interval ppi(p,pim); z=ppi;
       std::cout << " p is lower" << std::endl; formatprn(ppi);
    }
    int lz=(int)(-log10int(z));   // lz is where the difference begins in interval ab.
    std::cout << "lz=" << lz << std::endl;
```

上記のプログラム GauLeg.cpp について述べる．

- 求める π の精度を tol に 10 進桁数で指定すると，平方根はその精度を 1% 上回る値を ttol にセットする．
- 相乗平均 $b_{n+1} = \sqrt{a_n b_n}$ は a_n を y に入れて by=b*y としてから SQR または SQRint を用いる（SQRint の場合は，区間の中点を b_{n+1} とする）．

コメントに「初期値を有理化」と入れたが，b_0 を $\frac{\sqrt{2}}{2}$ にすると，高速化される．-DSQRRATIONALB を指定するかしないかとの組み合わせで，4 通りを測定した．tol を 2000 にしたときのパソコンでの計算時間（秒）を示す．

b_0	$\frac{1}{\sqrt{2}}$		$\frac{\sqrt{2}}{2}$	
平方根関数	SQR	SQRint	SQR	SQRint
ループ全体	3.48	6.38	0.22	0.22
平方根	.047	.031	.016	.016

有理算術演算では，加算，減算，乗算だけで構成される計算式では,「分母 2 べき」は保存される．分母が 2 のべき乗の有理数は有限桁数の 2 進数である．平方根を SQR や SQRint で求めるなら，計算式に平方根が入っても「分母 2 べき」は保存され，加算，減算，乗算の各演算後の GCD 計算は，2 進の GCD アルゴリズムで高速処理される．このため，初期値 b_0 を $\frac{\sqrt{2}}{2}$ にすると，高速化される．$\frac{1}{\sqrt{2}}$ では，反復で使用する変数の a_n の分母が 2 のべき乗（2 進数）でなくなり，加算（$a_n + b_n$）や乗算（$a_n b_n$）のたびに

既約分数にするための GCD 計算に，長い時間を要する．つまり，無理数を追及するために使用する有理数は，2 進数に丸めると速くなる．

数学では $\frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2}$ であるが，有理算術演算では分母・分子を自然数で保持する．$\sqrt{2}$ は正確な数ではない（丸められる）ので，この等号は成立しない．**分母が 2 のべき乗の有理数は 2 進数であるが，その逆数は 2 進数ではない**[72]．有理算術演算では，逆数の扱いに注意が必要である．

なお，`setpi` による連分数展開よる π の計算は 8 秒以上を要しており，ガウス・ルジャンドル法のほうが速い．

5.6 関数の計算と非線形方程式の求解

関数は有理数を係数にもつ多項式を扱うので，関数の値は正確に計算できる．多項式の零点は非線形方程式の解なので，一般には無理数であり，正確に計算することはできない．2 分法を用いると，分母が 2 のべき乗の有理数（2 進数）で解を追究できる．浮動小数点計算では，反復回数を減らす目的で，挟み撃ち法やニュートン法を用いる．挟み撃ち法（regula falsa）で，2 分法の中点の代わりに，直線近似により直線と x 軸との交点を使うと，分母 2 べきが保存されないため，反復回数が減っても，計算時間は極端に長くなる．ここで重要な高速化は，無理数を追究するとき，分母 2 べきに丸める手法をとることにある．「有理数計算プログラミング環境」では，指定された精度で，与えられた有理数を 2 進数に丸める関数 roundrat を用意した．挟み撃ち法とニュートン法ではこれを使用する．

5.6.1 関数と導関数の値 Horner, Horner3

ホーナー法で多項式の値を求める `Horner` 関数や，1 階導関数，2 階導関数の値も求める `Horner3` 関数を rational クラスに含めた．これらの関数については，資料 dvsrch.pdf を参照されたい．

5.6.2 2 分法 bisect

解の存在する区間を，下限と上限で抑えたら，2 分法による反復で，区間幅を半分，半分にしてゆける．この関数 `bisect` 含めた．なお，bisect は，引数を interval 型にした関数もある．この関数については，資料 dvsrch.pdf を参照されたい．

5.6.3 2 進数丸め関数 roundrat と挟み撃ち法 bisectrf

`roundrat(x,n)` 関数は，有理数 $x = \frac{x_n}{x_d}$ を，分母が 2 のべき乗の 2 つの有理数 y_l と y_u による区間 (y_l, y_u) に挟み，この区間を返す．

$$y_l < x < y_u \tag{41}$$

[72] 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．

$$a = \left(\pm \sum_{i=0}^{52} d_i 2^{-i} \right) \cdot 2^e = A \cdot 2^e \tag{40}$$

$B = A \cdot 2^{52}$ 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 (40) の数は，$a = B \cdot 2^{e-52}$ である．$C = 52 - e$ とおくと $a = B \div 2^C$ であり，分子は B，分母は 2^C と有理数表現される．分子が 2 のべき乗で，分母が 2 のべき乗でない有理数が 2 進数でないことは，有限桁数の 2 進数は正確に 10 進数に変換できることから証明できる．例えば $\frac{9}{8} = 1.125$ であるが，$\frac{8}{9} = \frac{(1000)_2}{(1001)_2} = 0.\dot{8}$ で循環小数になり，正確に 10 進変換できない．つまり有限桁数の 2 進数ではない．

表 2: roundrat 関数に与える有理数と出力される区間

有理数 x	引数 n	ULP	出力される区間
$\frac{1}{3} \fallingdotseq 0.3$	2	$\frac{1}{1{,}024^2}$	$\left(\frac{349{,}525}{1{,}024^2}, \frac{349{,}566}{1{,}024^2}\right) = \left(\frac{349{,}525}{1{,}048{,}576}, \frac{174{,}763}{524{,}288}\right)$
同上	1	$\frac{1}{1024}$	$\left(\frac{341}{1{,}024}, \frac{342}{1{,}024}\right) = \left(\frac{341}{1{,}024}, \frac{171}{512}\right)$
同上	0	1	$(0, 1)$
同上	-1	1024	$(0, 1024)$
$\frac{1{,}000}{3} \fallingdotseq 333.3$	2	$\frac{1}{1024^2}$	$\left(\frac{349{,}525{,}333}{1024^2}, \frac{349{,}525{,}334}{1{,}024^2}\right) = \left(\frac{349{,}525{,}333}{1{,}048{,}576}, \frac{174{,}762{,}667}{524{,}288}\right)$
同上	1	$\frac{1}{1024}$	$\left(\frac{341333}{1024}, \frac{341{,}334}{1{,}024}\right) = \left(\frac{341{,}333}{1{,}024}, \frac{170{,}667}{512}\right)$
同上	0	1	$(333, 334)$
同上	-1	1024	$(0, 1024)$
$\frac{1{,}000{,}000}{3} \fallingdotseq 333333.3$	2	$\frac{1}{1{,}024^2}$	$\left(\frac{349{,}525{,}333{,}333}{1{,}024^2}, \frac{349{,}525{,}333{,}334}{1{,}024^2}\right) = \left(\frac{349{,}525{,}333{,}333}{1{,}048{,}576}, \frac{174{,}762{,}666{,}667}{524{,}288}\right)$
同上	1	$\frac{1}{1{,}024}$	$\left(\frac{341{,}333{,}333}{1{,}024}, \frac{341{,}333{,}334}{1{,}024}\right) = \left(\frac{341{,}333{,}333}{1{,}024}, \frac{170{,}666{,}667}{512}\right)$
同上	0	1	$(333333, 333334)$
同上	-1	1024	$(332800, 333824)$

y_l，y_u の分子 y_n は次式より得る．

$$x = \frac{x_n}{x_d} \fallingdotseq \frac{y_n}{2^k} = y \tag{42}$$

この関数は，無理数を含む区間を縮小反復する場合，**両端点は正確である必要はないのですこしずらして高速化**する．

有理数 $x = \dfrac{x_n}{x_d}$ と分母の 2 のべき k を与えると，y の分子 y_n を区間

$$y_n = \left(\left\lfloor \frac{2^k x_n}{x_d} \right\rfloor, \left\lceil \frac{2^k x_n}{x_d} \right\rceil\right) = (y_{nl}, y_{nu}) \tag{43}$$

で求めて，$y = (y_l, y_u) = \left(\dfrac{y_{nl}}{2^k}, \dfrac{y_{nu}}{2^k}\right)$ を区間（interval 型の変数）で返す．記号 $\lfloor x \rfloor$ は x を，最も近い小さいほうの 2 進数に丸め，記号 $\lceil x \rceil$ は x を，最も近い大きいほうの 2 進数に丸める．

丸めの精度は $\dfrac{1}{1024}$ のべきで隣接する 2 進数の間隔を指定する．これは浮動小数点数の ULP（Unit in the Last Place）に対応する．無理数解を扱う関数で区間を返すものは区間幅を 10 進数で指定し，有理数を返す関数も収束判定は 10 進数を指定する．**区間の上限と下限の差が区間幅であるが，これは `formatprn` 関数で表示したときの差異の現れる小数点以下の桁の位置である場合が多い**．「小数点以下 9 桁目」は $(10^{-3})^3$ であるが，これは $(2^{-10})^3$ に近い．計算機用語では K（キロ）は 1024 であり $2^{10} \fallingdotseq 10^3$ に基づいているが，「有理数計算プログラミング環境」でもこの近似を使用して初等関数の精度を 10 進数で指定する．例えば平方根関数 `SQR(a,n)` は，有理数 a の平方根の近似値の精度を 10 進数で 10^{-n} と指定する．

$x = \dfrac{1}{3}$ を $\mathtt{n} = 2,\ 1,\ 0,\ -1$ の 4 通りで丸めると，ULP は $\dfrac{1}{1024^2}$, $\dfrac{1}{1024}$, 1, 1024 が使用され，出力される区間は，表 2 の上の 4 段になる．表には $x = \dfrac{1000}{3}$ と $\dfrac{1000000}{3}$ の場合も示した．

$x = \dfrac{1}{3}$ は分母と分子の桁数が小さいので，丸める価値がないが，$x = \dfrac{1000(1000\mathtt{digits})000}{3000(1000\mathtt{digits})001}$ のように $x = \dfrac{1}{3}$ にわずかにずれ，しかも分母と分子の桁数が大きな有理数についても，同じ区間が返るので，この場合は高速化の効果がある[73]．

[73]加減算と乗算だけが続く計算では「分母が 2 のべき乗」の状態は継続する．GCD アルゴリズムを 2 進の `lgcd2` を用いると，

高速化の効果がある現実的な数値例を紹介する．挟み撃ち法 $d = \dfrac{af(b) - bf(a)}{f(b) - f(a)}$ で得られる x 切片の座標値 d が，特性多項式 $f(x)$ の係数が 300 桁を超える整数になっているとき，分子 $af(b) - bf(a)$ の桁数は大きく，さらに分母に $f(b) - f(a)$ があるため，分母・分子が 1435 桁になっている．

$$d = -\frac{66361(1425 \text{ digits })59873}{28355(1425 \text{ digits })69856} = -2.340375387850571\cdots \tag{44}$$

この桁数で，しかも分母は 2 のべき乗ではないので，計算速度はおおいに低下する．これを roundrat 関数で丸めて次の区間を得る．

$$\begin{aligned}(d_l, d_u) &= \left(-\frac{30379(24 \text{ digits })84523}{12980(24 \text{ digits })05024}, -\frac{15189(23 \text{ digits })92261}{64903(22 \text{ digits })52512}\right) \\ &= \left(-\frac{30379(24 \text{ digits })84523}{2^{110}}, -\frac{15189(23 \text{ digits })92261}{2^{109}}\right) \\ &= (-2.340375387850571\cdots, -2.340375387850571\cdots)\end{aligned} \tag{45}$$

倍精度の 17 桁では差異は現れず，計算速度は向上する．

ratutil クラスの roundrat 関数

```
interval roundrat(const rational& x, const int n) {
   rational  xl, xu, y, rt=rational::ONE;
   for(int i=0; i<abs(n); i++) rt=rt*rational::KIL; 1024 倍を繰り返してスクリーン幅を得る
   longint xn=x.numerator(); longint xd=x.denominator();
   rational xnr(xn,longint::ONE); rational xdr(xd,longint::ONE);
   if(n > 0){ y=rt*xnr/xdr;
   }else{ y=xnr/(rt*xdr);
   }
   rational y0=y.floor(); rational y1=y.ceil();
   if(n>=0){
      y0=y0/rt; y1=y1/rt;
      if(x>rational::ZERO){ xl=y0; xu=y1;
      }else{ xl=-y1; xu=-y0;
      }
   }else{
      y0=y0*rt; y1=y1*rt;
      if(x>rational::ZERO){ xl=y0; xu=y1;
      }else{ xl=-y1; xu=-y0;
      }
   }
   interval xint=interval(xl,xu); return xint;
}
```

roundrat に続けて細かく求める roundrat2 関数も用意した．

ratutil クラスの roundrat2 関数のプロトタイプ宣言

```
interval roundrat2(const rational& x, const int n, const int m);
```

引数 `n` は roundrat と同じで，1024 倍を n 回の後，2 倍を m 回繰り返して `rt` を設定する．

これらの 2 進数への丸め関数は，後述する平方根関数 SQR や，非線形方程式の解法である bisectrf や newton 関数で使用している．図 8 の左に，区間 (a, b) 間に d が得られたとき，これを含む 2 進数の間隔を 3 通り示した．roundrat2 関数に指定する精度によって，返される区間 (d_l, d_u) も 3 通りになる．粗い区間 (d_l, d_u) に対して，中間の粗さの区間は 2 倍の精度を指定した場合，細かい区間は 4 倍の精度を指定

有理算術演算のたびに行う GCD 計算が「分母 2 べき」によって高速に処理されるので，区間両端を「分母 2 べき」に丸めることは高速化に貢献する．

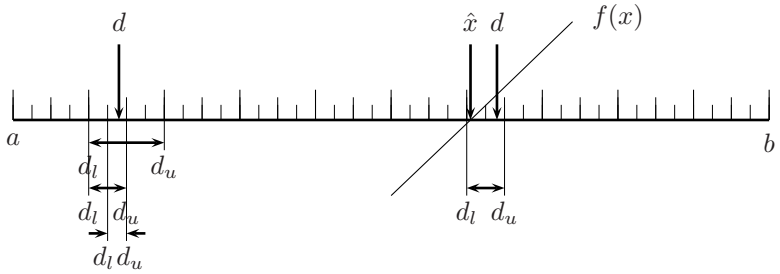


図 8: d を含む精度による 3 通りの区間 (d_l, d_u) と $f(x) = 0$ の求解での 2 進数区間の決定

した場合である．d は任意の有理数をとれるが， d_l, d_u は分母が 2 のべき乗，つまり 2 進数を有理数変換した有理数である[74]．

2 進数間隔はどこまでも細かくしていけるが，**無理数解 $\hat{x}$ とその予測値（近似値）d がともに属する最も狭い 2 進数区間を得る**ことが目的である．図 8 の右に，区間 (a, b) 間に多項式 $f(x)$ の零点 $\hat{x}$ とその近似解 d が含まれる最小の 2 進区間を示した．図には $\hat{x}$ を示したが，実際には真の解は分からない．しかし $f(d_l) \cdot f(d_u) < 0$ の条件から $\hat{x}$ と d が同じ区間に含まれるかどうかは判定できる．図の 2 進数間隔よりももうひとつ細かな区間では，d_l が右に移動するので $f(d_l) \cdot f(d_u) > 0$ になり含まれないと判定できる．この包含の判定は `roundrat` 関数を呼ぶ側のプログラムの仕事である．このようにして，近似解 d と解 $\hat{x}$ を含む区間の両端を 2 進数にできる．これは高速化の重要な技術である．

解の存在する区間を，下限と上限で抑えたら，挟み撃ち法と 2 分法を混在させて区間を縮小し，最後に `roundrat` 関数によって 2 進数に区間を丸める．この関数 `bisectrf` 含めた．なお，bisectrf は，bisect 同様，引数を interval 型にした関数もある．この関数については，資料 dvsrch.pdf を参照されたい．

5.6.4 ニュートン法 newton

newton は，bisect，bisectrf 同様，引数を interval 型にした関数もある．この関数については，資料 dvsrch.pdf を参照されたい．

[74] d が式 (44) の rational 型の数であるとき，これを roundrat と roundrat2 によって式 (45) の interval 型の数に変換するために使用する．

6　例題

本文に説明しなかった例題について説明する．

自然数の逆数の積和は，多桁数の例題「自然数の 7 乗の和」（powersum.cpp）に続けて用意した例題で，longint 型に，rational 型を加えて，有理数を正確に計算する．

π の計算は，有理算術演算による数値計算とは GCD の使用が異なる計算である．

π の計算，対称行列の LDL 分解，シュミットの直交化は，Makefile に含まれる例題である．

このほか，dvsrch.cpp，dvsrch1.cpp，dvsrch2.cpp も Makefile に含まれるが，これについては，資料 dvsrch.pdf を参照されたい．

6.1　自然数の逆数の積和（rational 型変数）

自然数の逆数の積和は，次の公式で与えられる．

$$\sum_{r=1}^{n} \frac{1}{r(r+1)} = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \frac{1}{3\cdot 4} + \cdots + \frac{1}{n\cdot(n+1)} = \frac{n}{n+1}$$

プログラムを示す．

invmulsum.cpp

```
#include <iostream>
#include "rational.h"
#include "longint.h"

using namespace std;

int main(void){
    rational x,y,x1,x12,x12inv;
    int i,n;

    cout << "Enter n" << endl;
    cin >> n;

    x = 0;
    y = 0;

    for(i=1; i<=n; i++){
        x += 1;
        x1=x + rational::ONE;
        x12=x*x1;
        x12inv=rational::ONE/x12;
        y=y+x12inv;
    }
    std::cout << "SUM_{i=1}^" << n << " 1/i*(i+1)=" << y << std::endl;

    return 1;
}
```

インクルードに追加するものは rational.h で，コンパイル時に追加するオブジェクトは rational.o である．“`g++ -O3 invmulsum.cpp rational.o longint.o mempool.o gettimeofday.o`” で稼働する．この例では，実行文の両辺の型を rational 型に統一している．“`x = 0;`” で，rational 型の数をゼロクリアできる．“`x += 1;`” で，rational 型の数に 1 を加えられる．“`x12inv=rational::ONE/x12;`” で rational 型の数の逆数が，rational 型変数 `x12inv` に得られる（rational 型の被除数と除数に対してはオペレータオーバーロードが効く）．

この問題を MPFUN で解く.

```
      program invmulsum
!
!  gfortran -O -c mpfour.f
!  gcc -O -c mpfft1.c
!  gfortran -O invmulsum.f mpfour.o mpfft1.o
!  ./a.exe
      character*1 c
      parameter (mx=2, nx=2**mx, ns=9.5*nx+47, nd=7.225*nx)
      dimension c(nd+50),x(nx+4),y(nx+4),z(nx+4)
      dimension x2(nd+50),x12(nx+4),x12inv(nx+4)
      dimension one(3)
      common /mpcom1/ nw, idb, ldb, ier, mcr, ird, ics, ihs, ims
      common /mpcom3/ s(ns)
      common /mpcom4/ d(12*nx+6)
      common /mpcom5/ u(8*nx)
      nw=nx
      ims=ns
      call mpinix(mx)
c -----
      one(1) = 1.; one(2) = 0.; one(3) = 1.
c -----
      x(1) = 0.; x(2) = 0.
      y(1) = 0.; y(2) = 0.
c -----
      write(*,*) 'n'
      read(*,*) n
      do i=1,n
        call mpadd (x, one, x)
        call mpadd (x, one, x1)
        call mpmul (x,  x1, x12)
        call mpdiv (one,x12,x12inv)
        call mpadd (x12inv, y, y)
      enddo
      call mpout(6,y,nd,c)
c -----
      call mpdiv (x, x1, y)     ! n/(n+1)
      call mpout(6,y,nd,c)
      stop
      end
```

パラメータ mx= 2 にしているが，$n = 5$ での結果は $0.833\cdots334$ と，29 桁に丸められている．パラメータ mx= 3 にすると，結果は $0.833\cdots333$ と，倍の桁数になる．mx を大きくしても，丸められることに変わりはなく，ループ反復して逆数の積を総和する計算結果とは，計算順序が異なるので異なった丸めになる．

6.2 対称行列の LDL 分解と代入計算

LDL.cpp は，7 種類の対称行列に対して，LDL 分解と代入を行う．次数 n またはその平方根を 7 種類の対称行列に対して指定する．

- ./LDL.exe 10 0 とすると，$n = 10$ のフランク行列を生成する（SetFrank）.
- ./LDL.exe 10 1 とすると，$n = 10$ のヒルベルト行列を生成する（SetHilbert）.
- ./LDL.exe 10 2 とすると，$n = 10$ のヒルベルト行列を倍精度で生成し（SetHilbet），cnvmat で有理数の行列に変換する.

- `./LDL.exe 10 3` とすると，$n = 10$ のヒルベルト行列を倍精度で生成し（SetHilbet），`cnvmat` で有理数の行列に変換し，`rat2int` で全要素の分母の LCM を求めて掛けることで整数行列に変換する．
- `./LDL.exe 5 4` とすると，$n = 5$ の擬似乱数による行列を生成する（SetRand）．
- `./LDL.exe 5 5` とすると，$n = 5$ 分母・分子を独立した擬似乱数による行列を生成する（SetRand2）．
- `./LDL.exe 5 6` とすると，$n = 5 \times 5 = 25$ の熱伝導行列を生成する（SetFtherm）．

右辺は解ベクトルの全要素が 1 になるように設定する．

次数 $n = 40$ での計算時間（秒）の比較を示す．

係数行列	時間（秒）
フランク行列	0.19
ヒルベルト行列	0.20
倍精度ヒルベルト行列	1.29
LCM 倍した倍精度ヒルベルト行列	1.43
分子乱数・分母 $2^{31} - 1$	1.43
分子乱数・分母乱数	251.40

計算機はノートパソコンで，1 スレッドのみを使用した．

また，例題には対角項を表示させる `SeeMatDiag` を入れてある．その表示例を，ヒルベルト行列について示す．

`./LDL.exe 6 1` による SeeMatDiag の出力

```
see diagonal terms before compute determinant
d_{0}=        1.00000000000000000000 NumDgts N/D=1 / 1  =\frac{1}{1}
d_{1}=        0.08333333333333333333 NumDgts N/D=1 / 1  =\frac{1}{12}
d_{2}=        0.00555555555555555556 NumDgts N/D=1 / 1  =\frac{1}{180}
d_{3}=        0.00035714285714285714 NumDgts N/D=1 / 1  =\frac{1}{2800}
d_{4}=        0.00002267573696145125 NumDgts N/D=1 / 1  =\frac{1}{44100}
d_{5}=        0.00000143154905059667 NumDgts N/D=1 / 1  =\frac{1}{698544}
determinant time=0(sec)
determinant=                  0.0000000000000000537 1/186313420339200000
```

最後に対角項の総積を計算して行列式を求める．

6.3 シュミットの直交化

LLSschmidt.cpp は，線形最小 2 乗解を求める．$A\boldsymbol{x} \cong \boldsymbol{b}$ の問題を解く．MGS，CGS，スケール化した行列に対して CGS，スケール付き CGS の 4 つの方法の比較を行う．詳しくは論文を参照されたい [10]．

参考文献

[1] P. van der Linden. *Just JAVA2.* Sun Microsystems, Inc., 1999. 中田秀基訳,Just JAVA2, アスキー, 2000.

[2] Steve Oualline. *Practical C++ Programming.* O'Reilly and Associates, Inc., 1995. 望月康司　訳：C++ 実践プログラミング，オライリー・ジャパン，1996.

[3] 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.

[4] 寒川 光, 藤野清次, 長嶋利夫, and 高橋大介. *HPC* **プログラミング***—ITText* **シリーズ**. オーム社, 2009.

[5] W. H. Press, Teukolsky S. A., Vetterling W. T., and B. P. Flannery. *Numerical Recipes in Fortran, second edition.* Cambridge University Press, 1992.

[6] J. H. Wilkinson. *Algebraic Eigenvalue Problem.* Oxford University Press, 1965.

[7] 宮西正宣（他）. **高等学校**–**数学** *B.* 新興出版社啓林館, 2007.

[8] 小林昭七. **円の数学**. 裳華房, 1999.

[9] 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.

[10] 寒川 光. 有理数線形代数計算における有理数 blas の提案. In *HPCS2014* **論文集**, pages 57–64, 2014.

索 引