# 連立１次方程式 I,

— 計算量と直接法 —

桂田 祐史

2015 年 12 月 16 日

# 目 次

# 第1章　序にかえてブツブツ

この文書の大部分は、20 世紀に書いたもので、何か今読むと辛いところが多い (2011/11/6)。今から書き直す余力はないので、放置しておくけれど、読む方はうまくピックアップして読まないといけないだろうな。まあ、定理とかプログラムに間違いは (あまり) ないはずだけど。

**目標**　この文書では連立 1 次方程式の直接法を説明する。桂田研の卒研のために用意したものである。

**現時点での言い訳**　足りないところが多い。

- 理論的な結果をもっと補充すべきだ。
- LINPACK, LAPACK, MATLAB 等のソフトウェアの紹介も欠かせない。
- 歴史についても記述すべきだ。

**蛇足または脱線**　書き方について。工夫が必要であると思う。もともと、このノートのシリーズは、数値解析の分野で「全面的に頼って」よいような本がないことに音をあげて (か弱い学生に安心して勧められる本って少ない)、また数少ない定評のある書物 (複数) の入手がかなり難しい現状に腹を立てて、書き出したものだ。

実際に書いてみると、色々な困難につきあたる。まず書く前から分かっていたことだが、正確な定式化、厳密な証明を探すのに骨が折れることが多い。誤解を恐れずに言えば、(いわゆる) 数学者が書いていないのが一番の原因であろう。例えば某 I 先生の書いた本は本当に素晴らしいと思うが、それでも時々何を主張しているのか良く分からないことがある。(証明の行間が読めないというのではない。何を主張しているのかが分からないということである。)

…もっとも、この辺はむしろ世代間ギャップかもしれないとも思うこともある。工学系の著者の場合も、若い場合は非常に明晰に主張を書く人が多いと感じている。そういえば、通常の数学書の世界でも、古い時代に書かれた本は、よほど著者がまともでないと霞がかかったようなのが結構あるから。

元に戻って、いわゆる普通の数学書とも、工学書とも異なる、自分の納得の行く書き方を作って (発見して) 書き上げたい。

例えば数学者の S 先生の本は、数値解析にかなりシンパシーのある書き方をしていると思うが、どこか詰めが甘い感じがする。個人的には使いでのある本で面白いと感じているが、普通の数学屋にはそれが良く分からないだろうし、工学の人にとって有益な本であるかどうかは分からない。

一方、例えば I 先生の本は、かなり徹底してアルゴリズミックな書き方であり、使う立場で紐解いた時に参考になるところは多いが、最初に通読するには辛いところがある。

少し話は飛んで。このノートでは定理にはなるべく証明を書こうとしているが、時々二つ以上の証明を与えることがある。それは、応用を考えているからには、構成的な (アルゴリズ

ミックな) 証明が良いのは確かだが、それが分かりやすいかどうかはまた別の問題だ、と感じているせいである[1]。

[1] しばらく前の雑誌『数学』に載っていたグラフ理論の教科書の書評で読んだように。

# 第2章　「発展系の数値解析」から

## (6) ガウス (Gauss) の消去法のアルゴリズム

連立 1 次方程式の解法として、線形代数の教科書には **クラーメル (Cramer) の公式**や**掃き出し法** (Jordan の消去法ともいう) が説明されていることが多いが、ガウスの消去法は、掃き出し法を改良したものである。

例として次の方程式を取りあげて説明しよう。

$$\left\{\begin{array}{rcrcrl} 2x_1 & + & 3x_2 & - & x_3 & = 5 \\ 4x_1 & + & 4x_2 & - & 3x_3 & = 3 \\ -2x_1 & + & 3x_2 & - & x_3 & = 1. \end{array}\right. \tag{2.1}$$

掃き出し法では係数行列と右辺のベクトルを並べた行列を作り、それに

1. ある行に 0 でない定数をかける。
2. 二つの行を入れ換える。
3. ある行に別の行を加える。

のような操作 — **行に関する基本変形**と呼ぶ — をほどこして、連立方程式の係数行列に相当する部分を単位行列にするのであった。

$$\begin{pmatrix} 2 & 3 & -1 & 5 \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 0 & -2 & -1 & -7 \\ 0 & 6 & -2 & 6 \end{pmatrix} \to$$

$$\to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 3 & -1 & 3 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & -\frac{5}{4} & -\frac{11}{4} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 0 & -\frac{5}{2} & -\frac{15}{2} \end{pmatrix} \to \begin{pmatrix} 1 & 0 & -\frac{5}{4} & -\frac{11}{4} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 0 & 1 & 3 \end{pmatrix} \to$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix}, \quad \text{ゆえに} \begin{pmatrix} x_2 \\ x_3 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

ガウスの消去法も、前半の段階はこの方法に似ていて、同様の変形を用いて掃き出しを行なうのだが、以下のように対角線の下側だけを 0 にする。

$$\begin{pmatrix} 2 & 3 & -1 & 5 \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 6 & -2 & 6 \end{pmatrix} \to \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 0 & -5 & -15 \end{pmatrix}.$$

最後の行列は

$$2x_1 + 3x_2 - x_3 = 5, \quad -2x_2 - x_3 = -7, \quad -5x_3 = -15$$

ということを表しているので、後の方から順に

$$x_3 = \frac{-15}{-5} = 3, \quad x_2 = \frac{-7 + x_3}{-2} = 2, \quad x_1 = \frac{5 - 3x_2 + x_3}{2} = \frac{5 - 3 \times 2 + 3}{2} = 1$$

と解くことが出来る。前半の対角線の下側を 0 にする掃き出しの操作を**前進消去** (forward elimination)、後半の代入により解の値を求める操作を**後退代入** (backward substitution) と呼ぶ。

以下簡単に 3 つの方法の比較をしよう。

クラーメルの公式を適用するには $n+1$ 個の行列式を求める必要があるため、計算の手間がかかる (大きな計算量が必要になる、という)。実際、行列式を一つ計算するための手間は、連立方程式を一つ解くための手間と本質的に同等であることが分かっているので、クラーメルの公式を使うことに固執すると、本来必要な計算量の $n$ 倍程度の計算をする羽目になり、大変な損をすることになる (差分法に応用する場合には、$n$ が非常に大きな数になることに注意しよう)。そのため、実際の数値計算では、ごく特殊な例を除いて、クラーメルの公式が利用されることはない。クラーメルの公式は、理論的な問題を扱う場合に、真価が発揮されるものである。

このクラーメルの方法に比べれば、掃き出し法は、かなりの実用性を持っているが、空間1次元の熱方程式を差分法で解く場合に現れる連立一次方程式のように[1]、係数行列が三重対角行列の場合には、ガウスの消去法の方が断然有利である。それは、ガウスの消去法を採用すると、掃き出しの途中に現れる行列が三重対角のままであることから、計算量が少なくてすむためである。

[1]例えば『発展系の数値解析』http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat-fdm-0.pdf を見よ。

# 第3章　計算量

## 3.1　計算量とは

同じものを計算で求める場合でも、うまくやるとやらないとでは、計算の手間がずいぶん異なるということは、経験上知っていることと思う。このことを科学的に考えよう、というのが**計算量**の理論である。

計算量には、**時間計算量**と**領域計算量**という二つのものがあるが、ここでは時間計算量のみを考える。(簡単に言って、領域計算量というのは、計算に必要なメモリーの量のことである。)

計算の手間を測るための素朴な尺度としては、コンピューターで実際に計算に要した時間を測るということが考えられるが、これは以下の理由からあまり良くない。

- 実際のコンピューター・システムを用いて実験しないと、何も分らない。
- コンピューター・システムによって、計算時間は大きく異なり、異なるコンピューター上で測定した結果を比較するのが難しい。
- コンピューター・システムには、計算の得手不得手があるので、あるシステムでは方法 A が良くても、別の system では方法 B が良いという結果が出やすく、一般的な話がしづらい。
- 同一のコンピューター・システムにおいても、搭載メモリー、コンパイラー・オプション、バックグラウンド・ジョブの差などによって、結果が異なる。

そこで、ある種の基本的かつ主要な演算に着目して、その回数を勘定することで、計算量を測るという考え方を採用する。この方法では上にあげた欠点のほぼすべてを一応は解決している。例えば、アルゴリズム入門にしばしば登場するソーティングでは、比較の回数や、要素の交換の回数を評価することが行われる。

以下では、連立1次方程式の解法などのいわゆる線形演算の計算量を考えるので、四則演算の回数を数えることにする。なお、しばしば乗除算の回数のみを数える。これは

- 多くの場合、乗除算と加減算の回数は大体同じになるので、片方だけ調べるだけで、大ざっぱな比較は出来る。
- 実際のコンピューターでは、乗算、除算に要する時間はほぼ同じ程度で、加減算よりも長い場合が多い。

という理由に基づくものである (その正当性について、ここで議論することはしない)。

## 3.2　基本的線形演算の計算量

### 3.2.1　ベクトルの和、スカラー乗法、内積

以下はいずれも明らかであろう。

**和**　$a, b$ が $\mathbf{R}^n$ の要素であるとき、$c = a + b$ を計算するには、ちょうど $n$ 回の実数の加算で十分である。

$$c_i = a_i + b_i \quad (i = 1, 2, \cdots, n).$$

C 言語風のプログラム断片を示せば以下のようになる:

```
for (i = 1; i <= n; i++)
  c[i] = a[i] + b[i];
```

**スカラー倍**　$a$ が $\mathbf{R}^n$ の要素、$\lambda \in \mathbf{R}$ であるとき、$c = \lambda a$ を計算するには、ちょうど $n$ 回の実数の乗算で十分である。

$$c_i = \lambda a_i \quad (i = 1, 2, \cdots, n).$$

```
for (i = 1; i <= n; i++)
  c[i] = lambda * a[i];
```

**内積**　$a, b$ が $\mathbf{R}^n$ の要素であるとき、$c = a \cdot b$ を計算するには、ちょうど $n$ 回の実数の乗算と $n - 1$ 回の加算で十分である。

$$c = \sum_{i=1}^{n} a_i b_i.$$

```
c = a[1] * b[1];
for (i = 2; i <= n; i++)
  c += a[i] * b[i];
```

### 3.2.2　行列×ベクトル、行列×行列

**和**　$A$ が $n$ 次正方行列、$b \in \mathbf{R}^n$ であるとき、$c = Ab$ を計算するには、$n^2$ 回の乗算と $n(n-1)$ 回の加算で十分である。

$$c_i = \sum_{j=1}^{n} a_{ij} b_j \quad (i = 1, 2, \cdots, n).$$

```
for (i = 1; i <= n; i++) {
  c[i] = a[i][1] * b[1];
  for (j = 2; j <= n; j++)
    c[i] += a[i][j] * b[j];
}
```

**念のため確認:**　加算の回数は $\displaystyle\sum_{i=1}^{n}\sum_{j=2}^{n} 1 = n(n-1)$, 乗算の回数は $\displaystyle\sum_{i=1}^{n}\left(1 + \sum_{j=2}^{n} 1\right) = n^2$.

**積**　$A$, $B$ が $n$ 次正方行列であるとき、$C = AB$ を計算するには、$n^3$ 回の乗算と $n^2(n-1)$ 回の加算で十分である。

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad (i, j = 1, 2, \cdots, n).$$

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++) {
    c[i][j] = a[i][1] * b[1][j];
    for (k = 2; k <= n; k++)
      c[i][j] += a[i][k] * b[k][j];
  }
```

**念のため確認:**　加算の回数は $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=2}^{n} 1 = n^2(n-1)$, 乗算の回数は $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n}\left(1 + \sum_{k=2}^{n} 1\right) = n^3$.

**問**　(各自必ずやってみること) $A$, $B \in M(n; \mathbf{R})$, $c \in \mathbf{R}^n$ とするとき、$ABc$ を計算するのに $(AB)c$ と結合して計算するのと、$A(Bc)$ と結合して計算するのと、いずれが得か調べよ。

### 3.2.3　連立 1 次方程式に対する Gauss の消去法

連立 1 次方程式を解くためのアルゴリズムに、**Gauss の消去法**というのがあるが (次の章で詳しく論じる)、以下に示すように $n$ 元連立 1 次方程式を解くのに、$n^3/3 + O(n^2)$ 回の乗除算で十分である。

行列が対称であったりすると $n^3/6 + O(n^2)$ にできる (これについてはここでは省略する)。

次に掲げるのは、C 言語で記述した Gauss の消去法のプログラムである。

```
void gauss(int n, matrix a, vector x)
{
    int i, j, k;
    double q, s;

    /* 前進消去 */
    for (k = 1; k <= n-1; k++)
        for (i = k + 1; i <= n; i++) {
            q = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] -= q * a[k][j];
            x[i] -= q * x[k];
        }
    /* 後退代入 */
    x[n] /= a[n][n];
    for (k = n - 1; k >= 1; k--) {
      s = x[k];
      for (j = k + 1; j <= n; j++)
        s -= a[k][j] * x[j];
      x[k] = s / a[k][k];
    }
}
```

乗算の回数は、まず前進消去で

$$\begin{aligned}\sum_{k=1}^{n-1}\sum_{i=k+1}^{n}\left(\sum_{j=k+1}^{n}1+1\right) &= \sum_{k=1}^{n-1}\sum_{i=k+1}^{n}(n-k+1)=\sum_{k=1}^{n-1}(n-k)(n-k+1)=\sum_{\ell=1}^{n-1}\ell(\ell+1)\\ &= \frac{n(n-1)(2n-1)}{6}+\frac{n(n-1)}{2}=\frac{n(n-1)}{6}(2n-1+3)\\ &= \frac{n(n-1)(n+1)}{3}.\end{aligned}$$

後退代入で

$$\sum_{k=1}^{n-1}\sum_{j=k+1}^{n}1=\sum_{k=1}^{n-1}(n-k)=\sum_{\ell=1}^{n-1}\ell=\frac{n(n-1)}{2}.$$

合計で

$$\frac{n(n-1)(n+1)}{3}+\frac{n(n-1)}{2}=\frac{n(n-1)}{6}(2(n+1)+3)=\frac{n(n-1)(2n+5)}{6}.$$

除算の回数は、まず前進消去で

$$\sum_{k=1}^{n-1}\sum_{i=k+1}^{n}1=\sum_{k=1}^{n-1}(n-k)=\sum_{\ell=1}^{n-1}\ell=\frac{n(n-1)}{2}.$$

後退代入で

$$1+\sum_{k=1}^{n-1}1=1+(n-1)=n.$$

合計で

$$\frac{n(n-1)}{2}+n=\frac{n(n+1)}{2}.$$

乗除算の回数は

$$\begin{aligned}\frac{n(n-1)(2n+5)}{6}+\frac{n(n+1)}{2} &= \frac{n}{6}\left((n-1)(2n+5)+3(n+1)\right)\\ &= \frac{n}{6}\left(2n^2+3n-5+3n+3\right)\\ &= \frac{n(n^2+3n-1)}{3}.\end{aligned}$$

### 3.2.4 Strassen の算法

前項まで「△回の○算で十分」と書いてきたが、「△回の○算が必要」ということは示せるだろうか？ (つまり、それ以上少ない計算量では計算できない、ということが分かるか、ということ) 簡単な場合にはそれが証明できるが、一般には難しい。行列の掛け算などでもそれほど簡単ではない…実は意外なことが成り立つ。先に示した素朴な方法では約 $n^3$ 回の乗算が必要となったが、もっと少ない乗算で済む方法があるのである。

V.Strassen は、二つの $n$ 次正方行列の積を計算するのに、$O(n^{\log_2 7})$ の演算回数で十分であることを示した ($\log_2 7 = 2.81\cdots$)。実際に、この算法が素朴な方法よりも効率的になるのは、$n$ が数十以上の密行列に対してであるということが、理論的、経験的に分かったため、実用にされることはないようである。オーダーだけで言えば、1981 年の段階で $n^{2.51\cdots}$ 程度にまで改良されているらしい (伊理・藤野 [3] の第 13 章)。

(**桂田独り言:** この後の歴史がどうなっているか、誰か調べてくれると嬉しい。実は $n$ がそれほど大きな場合でなくても、素朴な $O(n^3)$ のアルゴリズムよりも速く計算できることがある、という話を読んだ気もする。知っている人はどうか教えてください。)

### 3.2.5 逆行列、連立 1 次方程式、行列式

行列の乗算、逆行列の計算、行列の LU 分解、連立 1 次方程式の求解、行列式の計算の計算量は、いずれも本質的に同じオーダーであることが分っている。詳しく言うと、ある問題に対して、計算量が少なくて済む、うまい解法が見つかった場合、それを利用して他の問題を同じオーダーの計算量で解くことができる、ということである。特に Gauss 消去法に基づくアルゴリズムでは、いずれも $O(n^3)$ の乗除算で問題が解ける。

### 3.2.6 行列の形を考慮したアルゴリズム

前項までの議論は、一般の行列を対象にしている。実際の問題に現われる行列は事前に色々な性質が分っていて、一般的な問題に適用できるのとは異なったアルゴリズムが利用できることがある。

例えば、三角行列[1] を係数行列とする $n$ 元連立 1 次方程式 $(Ly = b, Ux = y)$ は、$n^2$ 回程度の乗除算で解くことができる。

例えば、3 項方程式 (三重対角行列を係数行列とする連立 1 次方程式) は、$6n$ 回程度の乗除算で解くことができる。実際、次章で掲載するプログラム `trid-lu.c` 内の関数 `trilu()` では $2(n-1)$ 回、関数 `trisol()` では $4n-3$ 回の乗除算をしている。

`trilu()` では、減算、乗算、除算の回数はいずれも

$$\sum_{i=0}^{n-2} 1 = n - 1.$$

`trisol()` では、減算、乗算の回数はいずれも

$$\sum_{i=0}^{n-2} 1 + \sum_{i=0}^{n-2} 1 = 2(n-1)$$

で、除算の回数は $2n - 1$.

### 3.2.7 逆行列シンドローム — もっと LU 分解を

ここでは、伊理&藤野 [3] の第 5 章「逆行列よさようなら」の内容を紹介する。

$Ax = b$ を解く、すなわち $x = A^{-1}b$ を計算するために、$A$ の逆行列 $A^{-1}$ を求めて、それを $b$ にかけるという手順が考えられるが、実はこれは (ほとんどすべての場合に) 大変まずいやり方である。

**よほどのことがない限り、逆行列など求めない**

のが本当である。ここは一つ

**桂田研では、数値計算で逆行列を求めたら破門**

と言っておこう。

同じ係数行列を持つ連立 1 次方程式を大量に解く場合も、逆行列を用いないでより賢く計算する方法がある。

[1] 三角行列については次の章で解説する。

1. 連立1次方程式の右辺が事前に全部分っている (それらを $b_1, b_2, \cdots, b_m$ とする) 場合は、

$$(Ab_1b_2\cdots b_m)$$

という行列を基本変形で

$$(Ix_1x_2\cdots x_m)$$

と変形すると、$x_j$ が $Ax = b_j$ の解になる。

2. 連立1次方程式の右辺が事前には分っていない場合は、最初に $A$ の LU 分解 ($A = LU$) を求めておいて、

$$Ax = b$$

を解くため、

$$Ly = b$$

とそれに続いて

$$Ux = y$$

を解けば良い。三角行列を係数に持つ連立1次方程式は効率的に解ける (次章で解説)。

## 3.3 数学の本にのっていない常識

- 実際の計算では、データの移動・コピーにかなりの時間がかかり、それが馬鹿にならない (CPU 上のデータ同士の演算よりも、CPU と メモリー間のデータ転送の方が時間がかかることも稀ではない)。高級プログラム言語の上では、代入に時間がかかる、ということである。

- 比較演算などは、ほとんど減算を行っていることに相当する。つまり比較にもコストがかかる。

- 普通 (あくまでも「普通」で、例外は多い)、コンピューターは加算と減算をほぼ同じ計算時間で遂行するが、乗算に要する時間はそれより長めで、除算はそれよりさらに長い時間を要する。さらに平方根計算、三角関数計算…と続く。

- 加減算と桁ずらしを基本命令として持つ計算機においては、乗除算と初等関数 (指数・対数関数、三角関数) の計算時間はせいぜい定数倍程度しか違わない。初等関数計算用に **CORDIC** (=coordinate rotation digital computation), **STL** (=sequential table lookup) などのすぐれたアルゴリズムがある (森・名取・鳥井 [?] などを見よ)。

- もっとも、普通は乗算は乗算器というハードウェアに任せ、初等関数は何らかの近似式 (Taylor 展開では ない) を利用して四則演算の組合わせで計算することが多い。

- 計算に要する時間は、演算精度とも関係する。もちろん高精度の演算の方が長い計算時間を要するが、精度と計算時間は比例の関係にあるというわけでもない。

- 浮動小数点演算と整数・論理型演算は明確に区別すべきである。普通は浮動小数点演算の方が整数演算よりも計算に要する時間が長いが例外もある。スーパーコンピューターなどの速さは、一秒間に何回の浮動小数点演算を実行できるか (FLOPS = floating operations per second) で測られることが多い。かつて新聞などを賑わしたパイプライン方式のスーパーコンピューターで数 GFLOPS であったが、現在ではワークステーションやパソコンでもその程度速度は出る。現在のスーパーコンピューターは超並列マシンが主流で、

地球シミュレータ (NEC SX6) は 35.86 TFLOPS を記録している (2004 年 6 月現在)。なお、http://www.top500.org/ を見よ。

- このテキストでは述べないが、数値解析を学ぶ者は最低限のことはマスターすべき話題として、高速 Fourier 変換 (fast Fourier transform, FFT) という魔術的アルゴリズムがある。

# 第4章　LU 分解を理解するための数学的準備

(LU 分解について初めて学ぶ人には

「『発展系の数値解析』に書き加えること」
http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat-fdm-0-add.pdf

の解説が分かりやすいかも知れない。この内容はそのうちに整理して、この文書に取り込む予定であるが、なかなか暇が見つからないので、当分は実現しないと思われる。)

## 4.1　行列の基本変形

### 4.1.1　行列単位

**定義 4.1.1 (行列単位)** $(i,j)$ 成分が 1 で、他のすべての成分が 0 である行列を $E_{ij}$ で表わし、行列単位と呼ぶ。

**補題 4.1.2 (行列単位の積)**

$$E_{ij}E_{k\ell} = \delta_{jk}E_{i\ell}.$$

### 4.1.2 基本行列

**定義 4.1.3 (基本行列)** $n$ 次の**基本行列** (elementary matrix) とは、次の 3 種類の行列の総称である。

1) (**互換行列** (interchange matrix)) $i \neq j$ とするとき、

$$P_n(i,j) \stackrel{\text{def.}}{=} \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & 0 & \cdots & \cdots & \cdots & 1 & & \cdots & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & 1 & \cdots & \cdots & \cdots & 0 & & \cdots & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix}.$$

(単位行列の第 $i$ 行と第 $j$ 行を入れ替えたもの。)

2) $c \neq 0$ とするとき、

$$Q_n(i;c) \stackrel{\text{def.}}{=} \operatorname{diag}(1 \ \ 1 \ \ \cdots \ \ 1 \ \ \underset{\substack{i \\ \downarrow}}{c} \ \ 1 \ \ \cdots \ \ 1) = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}.$$

(単位行列の $(i,i)$ 成分を $c$ にしたもの。)

3) $i \neq j$ とするとき、

$$R_n(i,j;c) \stackrel{\text{def.}}{=} I + cE_{ij} \quad (\text{ここで } E_{ij} \text{ は行列単位})$$

(単位行列の $(i,j)$ 成分を $c$ にしたもの。)

**命題 4.1.4 (基本行列の行列単位による表現)** (一部定義中の式と重複するが)

$$\begin{aligned} P_n(i,j) &= E_{ij} + E_{ji} + \sum_{k \neq i,j} E_{kk} = I - E_{ii} - E_{jj} + E_{ij} + E_{ji} \quad (i \neq j), \\ Q_n(i;c) &= I + (c-1)E_{ii}, \\ R_n(i,j;c) &= I + cE_{ij} \quad (i \neq j). \end{aligned}$$

**命題 4.1.5 (基本行列の掛け算)** $A$ を $(m,n)$ 型行列とする。

(1) $A$ に左から $P_m(i,j)$ をかけると、$A$ の第 $i$ 行と第 $j$ 行が交換される。

(2) $A$ に右から $P_n(i,j)$ をかけると、$A$ の第 $i$ 列と第 $j$ 列が交換される。

(3) $A$ に左から $Q_m(i;c)$ をかけると、$A$ の第 $i$ 行が $c$ 倍される。

(4) $A$ に右から $Q_n(i;c)$ をかけると、$A$ の第 $i$ 列が $c$ 倍される。

(5) $A$ に左から $R_m(i,j;c)$ をかけると、$A$ の第 $i$ 行に第 $j$ 行の $c$ 倍が加わる。

(6) $A$ に右から $R_n(i,j;c)$ をかけると、$A$ の第 $i$ 列に第 $j$ 列の $c$ 倍が加わる。

**定義 4.1.6 (行列の基本変形)** 命題 4.1.5 にある変形を**基本変形** (elementary transformation) と総称する。そのうち左から掛け算するものを**行に関する基本変形** (基本行演算, elementary row operation)、右から掛け算するものを**列に関する基本変形** (基本列演算, elementary column operation) と呼ぶ。

**命題 4.1.7 (基本行列の逆行列)** 基本行列は正則である。

$$P_n(i,j)^{-1} = P_n(i,j) \quad (i \neq j).$$

$$Q_n(i;c)^{-1} = Q_n(i;c^{-1}) \quad (c \neq 0).$$

$$R_n(i,j;c)^{-1} = R_n(i,j;-c) \quad (i \neq j).$$

**系 4.1.8** 基本変形は「可逆」である。すなわち、ある行列 $A$ に基本変形を施して行列 $B$ が得られた場合、$B$ に基本変形を施して $A$ に戻すことができる。

### 4.1.3 便利な命題をいくつか

次の二つの命題は伊理 [2] に載っていたものだが、さすがにかゆいところに手が届く感じである。

**命題 4.1.9 (基本行列同士の積)** (以下、下つき添字 $_n$ は省略して書く。)

(1) $i_0 \neq j_0$ とするとき、

$$P(i_0, j_0)Q(k; c) = Q(\ell; c)P(i_0, j_0),$$

ただし

$$\ell = \begin{cases} k & (k \neq i_0 \text{ かつ } k \neq j_0 \text{ のとき}) \\ j_0 & (k = i_0 \text{ のとき}) \\ i_0 & (k = j_0 \text{ のとき}). \end{cases}$$

(2) $i_0 \neq j_0, k \neq \ell$ とするとき、

$$P(i_0, j_0)R(k, \ell; c) = R(k', \ell'; c)P(i_0, j_0),$$

ただし

(i) $\{k, \ell\} \cap \{i_0, j_0\} = \emptyset$ のとき、$k' = k$, $\ell' = \ell$.

(ii) $k = i_0$ かつ $\ell \neq j_0$ のとき $k' = j_0$, $\ell' = \ell$.

(iii) $k = j_0$ かつ $\ell \neq i_0$ のとき $k' = i_0$, $\ell' = \ell$.

(iv) $k \neq i_0$ かつ $\ell = j_0$ のとき $k' = k$, $\ell' = i_0$.

(v) $k \neq j_0$ かつ $\ell = i_0$ のとき $k' = k$, $\ell' = j_0$.

(vi) $k = i_0$ かつ $\ell = j_0$ のとき $k' = \ell$, $\ell' = k$.

(vii) $k = j_0$ かつ $\ell = i_0$ のとき $k' = \ell$, $\ell' = k$.

(3) $i_0 \neq j_0$ とするとき、

$$R(i_0, j_0; c)Q(k; d) = Q(k; d)R(i_0, j_0; c'),$$

ただし

$$c' = \begin{cases} c & (k \neq i_0 \text{ かつ } k \neq j_0 \text{ のとき}) \\ cd & (k = j_0 \text{ のとき}) \\ c/d & (k = i_0 \text{ のとき}). \end{cases}$$

**命題 4.1.10 (互換行列を他の基本行列の積で表わす)** $i_0 \neq j_0$ とするとき、

$$P(i_0, j_0) = Q(j_0; -1)R(i_0, j_0; 1)R(j_0, i_0; -1)R(i_0, j_0; 1).$$

この命題を基本変形の言葉に翻訳すると次の系が得られる (こちらの命題は齋藤 [7] でも採り上げられていた)。

**系 4.1.11 (行の交換は「なくてもよい」)** 行の交換は、他の二つの行に関する基本変形の組合せで実現できる。例えば、$i$ 行と $j$ 行を交換するには、次の 4 つの基本変形を順に行なえば良い。

1) $i$ 行に第 $j$ 行を加える

2) $j$ 行から第 $i$ 行を引く

3) $i$ 行に第 $j$ 行を加える

4) $j$ 行に $-1$ をかける

(つまり、四則演算で swap ができるという奴ですね。)

**定義 4.1.12 (置換行列)** 正方行列 $P$ の要素がどれも 0 か 1 に等しく、しかもどの行にも 1 がちょうど 1 個、どの列にも 1 がちょうど 1 個あるとき、$P$ を**置換行列** (permutation matrix) とよぶ。

この定義から、置換行列 $P$ は実直交行列であり、したがって、

$$P^{-1} = P^T$$

が成り立つことが分かる。また $P^T$ も置換行列であることも定義から明らかである。これから置換行列全体は群をなすことが分かる ($n$ 次の置換全体の作る群と同型になることは明らかなので、それからも分かる)。

$m$ 次の置換行列 $P = (p_{ij})$ の第 $i$ 行の $\sigma(i)$ 列目に 1 があるとする $(i = 1, 2, \cdots, m)$、すなわち

$$p_{ij} = \begin{cases} 1 & (j = \sigma(i)) \\ 0 & (j \neq \sigma(i)). \end{cases}$$

このとき $\sigma$ は $m$ 次の置換に他ならない。置換 $\sigma$ に対応する置換行列を $P_\sigma$ と書くことにする。

置換行列の全体と $m$ 次対称群は同型であることが容易に分かる。$P_\sigma \mapsto \sigma$ がその対応を与える。

よく知られているように、任意の置換は互換の積で表わせるから、任意の置換行列は互換行列の積で表わせる。

置換行列 $P$ の逆行列は置換行列であることは既に説明したが、このことは、

$$P \text{ が互換行列} \quad \Longrightarrow \quad P^{-1} = P^T = P$$

という事実を用いて以下のように示すことも出来る (これはこれで何かの参考になりそうなので書いておく)。置換行列 $P$ を互換行列の積

$$P = P_1 P_2 \cdots P_k$$

に分解すると

$$P^{-1} = (P_k)^{-1}(P_{k-1})^{-1} \cdots (P_2)^{-1}(P_1)^{-1} = (P_k)^T (P_{k-1})^T \cdots (P_2)^T (P_1)^T = (P_1 P_2 \cdots P_k)^T = P^T.$$

$(m, n)$ 型行列

$$A = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

に左から $m$ 次置換行列 $P_\sigma$ をかけたもの $A' = P_\sigma A$ の行集合は $A$ の行集合に置換 $\sigma$ を施したものになる:

$$P_\sigma \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} a_{\sigma(1)} \\ a_{\sigma(2)} \\ \vdots \\ a_{\sigma(m)} \end{pmatrix}.$$

同様に右から $n$ 次置換行列 $P_\tau$ をかけたもの $A'' = AP_\tau$ の列集合は $A$ の列集合に置換 $\tau$ を施したものになる:

$$(\vec{a}_1\ \vec{a}_2 \cdots \vec{a}_n)P_\tau = (\vec{a}_{\tau(1)}\ \vec{a}_{\tau(2)} \cdots \vec{a}_{\tau(n)}).$$

おまけ: **Octave における LU 分解**

MATLAB 互換のソフト Octave の関数 `lu()` で、与えられた行列 $A$ の LU 分解 $PA = LU$ (これは $A = P^T LU$ と同値) を求めることが出来る。

```
yurichan% octave
GNU Octave, version 2.0.16 (i386-unknown-freebsd3.4).
Copyright (C) 1996, 1997, 1998, 1999, 2000 John W. Eaton.
This is free software with ABSOLUTELY NO WARRANTY.
For details, type 'warranty'.

octave:1> n=10;a=rand(n,n);[l u p]=lu(a);norm(p'*l*u-a)
ans =  3.4269e-16
octave:2> p
p =

  0  1  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  1
  0  0  0  0  1  0  0  0  0  0
  0  0  0  0  0  1  0  0  0  0
  0  0  0  0  0  0  0  0  1  0
  1  0  0  0  0  0  0  0  0  0
  0  0  1  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  1  0  0
  0  0  0  0  0  0  1  0  0  0
  0  0  0  1  0  0  0  0  0  0

octave:3> p*p'
ans =

  1  0  0  0  0  0  0  0  0  0
  0  1  0  0  0  0  0  0  0  0
  0  0  1  0  0  0  0  0  0  0
  0  0  0  1  0  0  0  0  0  0
  0  0  0  0  1  0  0  0  0  0
  0  0  0  0  0  1  0  0  0  0
  0  0  0  0  0  0  1  0  0  0
  0  0  0  0  0  0  0  1  0  0
  0  0  0  0  0  0  0  0  1  0
  0  0  0  0  0  0  0  0  0  1

octave:4> b=rand(n);x=u\(l\(p*b));norm(a*x-b)
ans =  1.7516e-15
octave:5>
```

$A^{-1} = U^{-1}L^{-1}(P^T)^{-1} = U^{-1}L^{-1}P$ であるから、$Ax = b$ の解は `U\(L\(P*b))` で計算できるわけである。

## 4.2 三角行列

**補題 4.2.1** 正方行列 $A$ の対称な区分け (symmetric partitioning)

$$A = \begin{pmatrix} A_{11} & A_{12} \\ O & A_{22} \end{pmatrix}$$

において、

$$A \text{ が正則} \quad \Leftrightarrow \quad A_{11} \text{ と } A_{22} \text{ が正則}$$

であり、

$$A^{-1} = \begin{pmatrix} {A_{11}}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ O & {A_{22}}^{-1} \end{pmatrix}.$$

**証明** 線型代数の教科書の例や演習問題に取り上げられていることが多いが、例えば齋藤 [7] を見よ。 ■

**命題 4.2.2** (1) $A = (a_{ij})$, $B = (b_{ij})$ が $n$ 次上三角行列ならば、$A + B$ も上三角行列である。

(2) $A = (a_{ij})$, $B = (b_{ij})$ が $n$ 次上三角行列ならば、$AB$ も上三角行列であり、その対角線分は、$A$, $B$ の対応する対角成分の積に等しい:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ 0 & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & & & * \\ 0 & a_{22}b_{22} & & \\ \vdots & \vdots & \ddots & \\ 0 & 0 & \cdots & a_{nn}b_{nn} \end{pmatrix}.$$

(3) $n$ 次上三角行列 $A = (a_{ij})$ について、

$$A \text{ が正則} \quad \Leftrightarrow \quad a_{ii} \neq 0 \quad (i = 1, 2, \cdots, n)$$

であり、$A$ が正則のとき、$A^{-1}$ は上三角行列である。

(4) $A$, $B$ がともに冪零上三角行列ならば、$AB$ は冪零上三角行列であり、その対角線のすぐ上の斜め線上の成分もすべて 0 になる。言い換えると $AB = (c_{ij})$ とするとき、

$$c_{i,i+1} = 0 \quad (1 \leq i \leq n-1).$$

(5) $A$ が $n$ 次の冪零上三角行列ならば、$A^n = O$.

**証明** 例えば齋藤 [7] を見よ。ただし、(3) の後半の「上三角行列の逆行列が上三角行列である」ことは載っていないので、以下に証明を与える[1]。$n$ 次の上三角行列 $A = (a_{ij})$ で、$a_{ii} \neq 0$ $(1 \leq i \leq n)$ を満たすものが与えられたとせよ。このとき上三角行列 $B = (b_{ij})$ で、$AB = I$ を満たすものが存在することを示す。$A$, $B$ が上三角であることは

$$i > j \quad \Longrightarrow \quad a_{ij} = b_{ij} = 0 \tag{4.1}$$

[1]補題 4.2.1 を用いた証明も簡単であるが、ここでは算法を与える証明を採用した。

と表わされる。さて、条件 $AB = I$ は

$$\sum_{k=1}^{n} a_{ik} b_{kj} = \delta_{ij} \quad (1 \leq i \leq n,\, 1 \leq j \leq n)$$

と書けるが、(4.1) より

$$\sum_{i \leq k \leq j} a_{ik} b_{kj} = \delta_{ij} \quad (1 \leq i \leq n,\, 1 \leq j \leq n)$$

と書直すことができる。この条件のうち、$i > j$ なる $i, j$ についてのものが成り立つことは明らかであるから [2]、以下は $i \leq j$ なる $i, j$ について考えればよい。$i = j$ の場合は、

$$a_{jj} b_{jj} = 1$$

であるから、

$$b_{jj} = \frac{1}{a_{jj}}. \tag{4.2}$$

一方 $i < j$ の場合は

$$0 = \sum_{k=i}^{j} a_{ik} b_{kj} = a_{ii} b_{ij} + \sum_{k=i+1}^{j} a_{ik} b_{kj}$$

より

$$b_{ij} = \frac{-1}{a_{ii}} \sum_{k=i+1}^{j} a_{ik} b_{kj}. \tag{4.3}$$

以下の手順で計算すれば (4.2), (4.3) を満たす $\{a_{ij}\}$, $\{b_{ij}\}$ が求まることは明らかである。 ■

**上三角行列の逆行列の計算**

```
(* 変数 b_ij の下三角部分は計算しない。*)
for j := 1 to n do begin
  b_jj := 1 / a_jj;
  for i := j - 1 downto 1 do begin
    s := 0;
    for k := i + 1 to j do s := s + a_ik * b_kj;
    b_ij := - s / a_ii;
  end;
end;
```

同様にして下三角行列の計算法を得る。

[2] $i \leq k \leq j$ を満たす $k$ は存在しないので、左辺は 0 に等しく、右辺 $\delta_{ij}$ も当然 0 に等しい。

**下三角行列の逆行列の計算**

```
(* 変数 b_ij の上三角部分は計算しない。*)
for i := 1 to n do begin
  b_ii := 1 / a_ii;
  for j := i - 1 downto 1 do begin
    s := 0;
    for k := j + 1 to i do s := s + b_ik * a_kj;
    b_ij := - s / a_jj;
  end;
end;
```

念のため、実際に動作実績のあるプログラム例をあげる (C++ で書かれている)。

```
MATRIX inv_u(const MATRIX &a)
{
  int n = RowDimension(a);
  MATRIX b = zeros(n,n);
  for (int j = 1; j <= n; j++) {
    b(j,j) = 1 / a(j,j);
    for (int i = j - 1; i >= 1; i--) {
      REAL s = 0;
      for (int k = i + 1; k <= j; k++)
        s = s + a(i,k) * b(k,j);
      b(i,j) = - s / a(i,i);
    }
  }
  return b;
}
```

```
MATRIX inv_l(const MATRIX &a)
{
  int n = RowDimension(a);
  MATRIX b = zeros(n,n);
  for (int i = 1; i <= n; i++) {
    b(i,i) = 1 / a(i,i);
    for (int j = i - 1; j >= 1; j--) {
      REAL s = 0;
      for (int k = j + 1; k <= i; k++)
        s = s + b(i,k) * a(k,j);
      b(i,j) = - s / a(j,j);
    }
  }
  return b;
}
```

簡単のため、対角成分がすべて 1 である下三角行列を単位下三角行列、対角成分がすべて 1 である上三角行列を単位上三角行列と呼ぶことにするが、それぞれ乗法について群をなす。

**系 4.2.3 (単位三角行列全体は乗法について群をなす)** $n$ 次の単位上三角行列は正則で、逆行列も単位上三角行列である。

## 4.3 主座小行列式が 0 にならない場合の LU 分解

すべての主座小行列式が 0 にならない場合 (代表例に正値対称行列がある)、LU 分解は簡単 (ピボットの選択 — 行や列の交換 — が必要ない、一意性もある) である。まずはこの場合をまとめておこう。

**命題 4.3.1 (Dolittle タイプの LU 分解)** $n$ 次正則行列 $A = (a_{ij})$ の主座小行列 $A_1$, $A_2$, $\cdots$, $A_n = A$ がすべて正則ならば、$A$ は正則な下三角行列 $L$ と対角成分がすべて 1 であるような上三角行列 $U$ との積 $LU$ の形に一意的に表わされる。

**証明**　齋藤 [7] (分解が可能なこと) $n$ に関する帰納法を用いる。$n = 1$ ならば明らか。

$$A = \begin{pmatrix} A_{n-1} & \boldsymbol{b} \\ \boldsymbol{c}^T & d \end{pmatrix}$$

と区分すると

$$\begin{pmatrix} A_{n-1} & \boldsymbol{b} \\ \boldsymbol{c}^T & d \end{pmatrix} = \begin{pmatrix} A_{n-1} & \boldsymbol{0} \\ \boldsymbol{c}^T & d - c^T {A_{n-1}}^{-1}\boldsymbol{b} \end{pmatrix} \begin{pmatrix} E_{n-1} & {A_{n-1}}^{-1}\boldsymbol{b} \\ \boldsymbol{0}^T & 1 \end{pmatrix}.$$

帰納法の仮定により $A_{n-1} = L_{n-1}U_{n-1}$ と書ける ($L_{n-1}$, $U_{n-1}$ はそれぞれ単位下三角行列、正則上三角行列)。

$$L = \begin{pmatrix} L_{n-1} & \boldsymbol{0} \\ \boldsymbol{c}^T (U_{n-1})^{-1} & d - \boldsymbol{c}^T {A_{n-1}}^{-1}\boldsymbol{b} \end{pmatrix}, \quad U = \begin{pmatrix} U_{n-1} & U_{n-1}{A_{n-1}}^{-1}\boldsymbol{b} \\ \boldsymbol{0}^T & 1 \end{pmatrix}$$

とおくと $L, U$ はそれぞれ単位下三角行列、正則上三角行列で $A = LU$ を満たす。

(一意性) $A = LU = L'U'$ となったとする。${L'}^{-1}L = U'U^{-1}$ で、左辺は単位下三角行列、右辺は上三角行列なので実は単位行列 $I$ に等しい。これから $L = L'$, $U = U'$. ■

**証明の楽屋裏**　条件を満たす $L, U$ があったとすると、$A = LU$ は

$$\begin{pmatrix} L_{n-1} & \boldsymbol{0} \\ \boldsymbol{q}^T & r \end{pmatrix} \begin{pmatrix} U_{n-1} & \boldsymbol{p} \\ \boldsymbol{0}^T & 1 \end{pmatrix} = \begin{pmatrix} A_{n-1} & \boldsymbol{b} \\ \boldsymbol{c}^T & a_{nn} \end{pmatrix}$$

と書ける。この条件は

$$L_{n-1}U_{n-1} = A_{n-1}, \tag{4.4}$$
$$L_{n-1}\boldsymbol{p} = \boldsymbol{b}, \tag{4.5}$$
$$\boldsymbol{q}^T U_{n-1} = \boldsymbol{c}^T, \tag{4.6}$$
$$\boldsymbol{q}^T\boldsymbol{p} + r = a_{nn} \tag{4.7}$$

と書ける。帰納法の仮定から、(4.4) を満たす正則下三角行列 $L_{n-1}$, 単位上三角行列 $U_{n-1}$ の存在が分かる。すると、(4.5), (4.6), (4.7) は

$$\begin{aligned} \boldsymbol{p} &= (L_{n-1})^{-1}\boldsymbol{b}, \quad \boldsymbol{q}^T = \boldsymbol{c}^T (U_{n-1})^{-1}, \\ r &= a_{nn} - \boldsymbol{q}^T\boldsymbol{p} = a_{nn} - \boldsymbol{c}^T (U_{n-1})^{-1}(L_{n-1})^{-1}\boldsymbol{b} = a_{nn} - {A_{n-1}}^{-1}\boldsymbol{b} \end{aligned}$$

と解ける。■

**系 4.3.2 (Crout タイプの LU 分解)** $n$ 次正則行列 $A=(a_{ij})$ の主座小行列 $A_1, A_2, \cdots, A_n = A$ がすべて正則ならば、$A$ は対角成分がすべて 1 であるような下三角行列 $L$ と正則な上三角行列 $U$ との積 $LU$ の形に一意的に表わされる。

**証明**　$A^T$ について命題 4.3.1 を適用すればよい。■

**系 4.3.3 (LDU 分解)** $n$ 次正則行列 $A=(a_{ij})$ の主座小行列 $A_1, A_2, \cdots, A_n = A$ がすべて正則ならば、$A$ は

$$A = LDU$$

の形に一意的に表わされる。ただし、$L$ は対角成分がすべて 1 であるような下三角行列、$D$ は対角行列、$U$ は対角成分がすべて 1 であるような上三角行列である。

**証明**　まず命題 4.3.1 により $A=LU$ と分解しておき、$u_{ij}$ の代りに $u_{ij}/u_{ii}$ を要素とする行列を改めて $U$ とおき、

$$D = \mathrm{diag}(u_{11}, u_{22}, \cdots, u_{nn})$$

とおけばよい。■

**系 4.3.4 (Cholesky 分解)** $n$ 次対称行列 $A=(a_{ij})$ の主座小行列 $A_1, A_2, \cdots, A_n = A$ がすべて正則ならば、$A$ は

$$A = LDL^T$$

の形に一意的に表わされる。ただし、$L$ は対角成分がすべて 1 であるような下三角行列、$D$ は対角行列である。

**証明**　まず $A=LDU$ と LDU 分解する。$A$ が対称行列であるから、

$$A = A^T = (LDU)^T = U^T D L^T$$

であり、$A=LDU$ と比べて (LDU 分解の一意性により対応するところが一致する)

$$L^T = U, \quad U^T = L.$$

ゆえに

$$A = LDL^T.$$■

## 4.4　一般的な場合の LU 分解

一般には与えられた行列 $A$ を LU 分解できる (下三角行列と上三角行列の積に書ける) とは限らない。$A$ の行あるいは列 (あるいはその両方) を置換した行列は LU 分解できる。

この節の内容は伊理 [2] による。証明が必要になったら参照せよ。

$m \le n$ のとき、$(m,n)$ 型行列 $A$ を

$$A = LU$$

と表わすことを考える ($m > n$ の場合は転置して考えればよい)。ただし $L=(\ell_{ij})$ は $m$ 次正方行列で

$$\ell_{ii} = 1 \quad (i = 1, 2, \cdots, m), \quad \ell_{ij} = 0 \quad (i < j)$$

を満たす行列、$U=(u_{ij})$ は $(m,n)$ 型行列で

$$u_{ij} = 0 \quad (i > j)$$

を満たす行列とする。

いわゆる Gauss の消去法アルゴリズムで現れる中間結果の行列の対角成分が 0 にならなければ (正方行列の場合、これは主座小行列式が 0 にならないことと同値？)、$A$ は LU 分解することができ、その分解は一意である。

$A$ がフルランク、すなわち $\operatorname{rank} A = m$ であるならば、$A$ の代りに適当に列の置換を施した行列 $A' = AP$ ($P$ は置換行列) が LU 分解できる:

$$AP = LU.$$

$A$ が正則行列の場合は (転置行列を考えることによって)

$$PA = LU$$

と書ける、と言ってもよいわけだ。

# 第5章　Gauss の消去法と LU 分解

(繰り返しになるが、LU 分解について初めて学ぶ人はまず

「『発展系の数値解析』に書き加えること」
http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat-fdm-0-add.pdf

を読むこと。)

## 5.1　なぜ必要なのか

微分方程式を数値的に解く場合に、非常にしばしば連立 1 次方程式

$$Ax = b$$

を解く必要が生じる。連立 1 次方程式の解は係数行列の逆行列 $A^{-1}$ を用いて

$$x = A^{-1}b$$

と記述されるが、実際に連立 1 次方程式を解くには、逆行列を求めて利用するよりも、Gauss の消去法を用いる方が効率上有利である (既に「発展系の数値解析」でそう述べてあるが、ここでは計算量の概念を用いて詳しく解析する)。これは一般に言えることであるが、特に微分方程式を解く場合に非常にしばしばそうなるような、疎行列を係数行列に持つ連立 1 次方程式については、Gauss の消去法の方が逆行列法よりも断然有利である。

ところで、熱伝導方程式を差分法で解く場合、同じ係数行列を持つ連立 1 次方程式を非常に多数解く必要が生じる。このような場合、毎回 Gauss の消去法を行うのは計算量が膨大になってしまい、逆行列を用いた方が有利になりうるように思われるかもしれないが、実は Gauss の消去法には、LU 分解という補助テクニックがあって、そうでない (結局、逆行列法は Gauss の消去法に勝てない) ことが分かる。

連立 1 次方程式の解法はよく研究されていて、様々な優れた方法があるが、Gauss の消去法＋ LU 分解がベストである場合は結構多い。特に 1 次元熱伝導方程式を差分法で解く場合に現われるような 3 項方程式 (係数行列が三重対角行列である連立 1 次方程式) を解く場合は、Gauss の消去法＋ LU 分解は決定版とも言える解法である。

## 5.2　Gauss の消去法の例による説明

連立 1 次方程式の解法として、線形代数の教科書には **Cramer (クラーメル) の公式**や**掃き出し法**が説明されていることが多いが、Gauss の消去法は、高等学校で学ぶ消去法である [1]。

[1] そういう意味では、Gauss の消去法は、もっとも基本的な解法であると言えるが、以下このテキストで説明するように、実は数値計算法の観点からはかなり優秀な方法である。数学の課程で線型代数の初歩を学ぶと掃き出し法 (Gauss-Jordan の消去法) という解法が「標準的解法」として推奨され、習得させられることになるが、それよりも Gauss の消去法の方が優秀な解法であるのは、少し皮肉なことではある。

例として次の方程式を取りあげて説明しよう。

$$\begin{cases} 2x_1 + 3x_2 - x_3 = 5 \\ 4x_1 + 4x_2 - 3x_3 = 3 \\ -2x_1 + 3x_2 - x_3 = 1. \end{cases}$$

掃き出し法では係数行列と右辺のベクトルを並べた行列を作り、それに

1. ある行に 0 でない定数をかける。
2. 二つの行を入れ換える。
3. ある行に別の行を加える。

のような操作 — **行に関する基本変形**と呼ぶ — をほどこして、連立方程式の係数行列に相当する部分を単位行列にするのであった。

$$\begin{pmatrix} 2 & 3 & -1 & 5 \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 0 & -2 & -1 & -7 \\ 0 & 6 & -2 & 6 \end{pmatrix} \to$$

$$\to \begin{pmatrix} 1 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 3 & -1 & 3 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & -\frac{5}{4} & -\frac{11}{4} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 0 & -\frac{5}{2} & -\frac{15}{2} \end{pmatrix} \to \begin{pmatrix} 1 & 0 & -\frac{5}{4} & -\frac{11}{4} \\ 0 & 1 & \frac{1}{2} & \frac{7}{2} \\ 0 & 0 & 1 & 3 \end{pmatrix} \to$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix}, \quad \text{ゆえに} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

Gauss の消去法も、前半の段階はこの方法に似ていて、同様の変形を用いて掃き出しを行なうのだが、以下のように対角線の下側だけを 0 にする。

$$\begin{pmatrix} 2 & 3 & -1 & 5 \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix} \to \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 6 & -2 & 6 \end{pmatrix} \to \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 0 & -5 & -15 \end{pmatrix}.$$

最後の行列は

$$2x_1 + 3x_2 - x_3 = 5, \quad -2x_2 - x_3 = -7, \quad -5x_3 = -15$$

ということを表しているので、後の方から順に

$$x_3 = \frac{-15}{-5} = 3, \quad x_2 = \frac{-7 + x_3}{-2} = 2, \quad x_1 = \frac{5 - 3x_2 + x_3}{2} = \frac{5 - 3 \times 2 + 3}{2} = 1$$

と解くことが出来る。前半の対角線の下側を 0 にする掃き出しの操作を**前進消去** (**forward elimination**)、後半の代入により解の値を求める操作を**後退代入** (**backward substitution**) と呼ぶ。

## 5.3 LU 分解とは何か

### 5.3.1 上の例の基本行列による解釈

$$A = \begin{pmatrix} 2 & 3 & -1 & 5 \\ 4 & 4 & -3 & 3 \\ -2 & 3 & -1 & 1 \end{pmatrix}$$

に左から順に

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \quad L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

をかけたことになる。

$$L_1A = \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ -2 & 3 & -1 & 1 \end{pmatrix}, \quad L_2L_1A = \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 6 & -2 & 6 \end{pmatrix}, \quad L_3L_2L_1A = \begin{pmatrix} 2 & 3 & -1 & 5 \\ 0 & -2 & -1 & -7 \\ 0 & 0 & -5 & -15 \end{pmatrix}$$

$$\mathcal{L} := L_3L_2L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -5 & 3 & 1 \end{pmatrix}$$

$$L := \mathcal{L}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -3 & 1 \end{pmatrix}$$

$$U := \begin{pmatrix} 2 & 3 & -1 \\ 0 & -2 & -1 \\ 0 & 0 & -5 \end{pmatrix}$$

$$LU = A$$

### 5.3.2 三角行列

$n$ 次正方行列 $L = (\ell_{ij})$ が**下三角行列** (lower triangular matrix) であるとは、対角線よりも上にある成分がすべて 0 であること:

$$\ell_{ij} = 0 \quad (1 \leq i < j \leq n)$$

である。つまり

$$L = \begin{pmatrix} \ell_{11} & & & 0 \\ \ell_{21} & \ell_{22} & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & \ell_{nn} \end{pmatrix}.$$

特に対角成分がすべて 1 である、つまり

$$L = \begin{pmatrix} 1 & & & 0 \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}.$$

となっている下三角行列を**単位下三角行列**と呼ぶことにする。

$n$ 次正方行列 $U = (U_i^n)$ が**上三角行列** (upper triangular matrix) であるとは、対角線よりも下にある成分がすべて 0 であること:

$$u_{ij} = 0 \quad (i > j)$$

である。つまり

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & u_{nn} \end{pmatrix}.$$

特に対角成分がすべて 1 である、つまり

$$U = \begin{pmatrix} 1 & u_{12} & \cdots & u_{1n} \\ & 1 & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & 1 \end{pmatrix}.$$

となっている上三角行列を**単位上三角行列**と呼ぶことにする。

**問**　上三角行列 (もちろん下三角行列) は和、差、積、(正則な場合は) 逆行列を作る操作について閉じていることを示せ。

### 5.3.3　LU 分解

行列 $A$ を下三角行列 $L$ と上三角行列 $U$ の積に分解する、つまり

$$\begin{aligned} A &= LU, \\ L &= \begin{pmatrix} \ell_{11} & & & 0 \\ \ell_{21} & \ell_{22} & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & \ell_{nn} \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & u_{nn} \end{pmatrix} \end{aligned}$$

(LU 分解).

の形に表わすことを、$A$ を **LU 分解** (LU decomposition) するという。特に $L$ が単位下三角行列である場合を **Crout タイプの LU 分解**と呼ぶ:

$$\begin{aligned} A &= LU, \\ L &= \begin{pmatrix} 1 & & & 0 \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & u_{nn} \end{pmatrix} \end{aligned}$$

(Crout タイプの LU 分解).

その反対に $U$ が単位上三角行列である場合を **Dolittle タイプの LU 分解**と呼ぶ:

$$\begin{aligned} A &= LU, \\ L &= \begin{pmatrix} \ell_{11} & & & 0 \\ \ell_{21} & \ell_{22} & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & \ell_{nn} \end{pmatrix}, \quad U = \begin{pmatrix} 1 & u_{12} & \cdots & u_{1n} \\ & 1 & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & 1 \end{pmatrix} \end{aligned}$$

(Dolittle タイプの LU 分解).

### 5.3.4 LDU 分解

$A$ を

$$A = \mathcal{L}D\mathcal{U},$$

$$\mathcal{L} = \begin{pmatrix} 1 & & & 0 \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}, \quad D = \begin{pmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{pmatrix}, \quad \mathcal{U} = \begin{pmatrix} 1 & u_{12} & \cdots & u_{1n} \\ & 1 & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ 0 & & & 1 \end{pmatrix}$$

のように単位下三角行列 $\mathcal{L}$, 対角行列 $D$, 単位上三角行列 U の積に分解することを **LDU 分解**とよぶ。

**命題 5.3.1** 行列の LDU 分解 $A = \mathcal{L}D\mathcal{U}$ があるとき、$A = \mathcal{L}(D\mathcal{U})$ は Crout タイプの LU 分解、$A = (\mathcal{L}D)\mathcal{U}$ は Dolittle タイプの LU 分解となる。

**証明** 本当に簡単なので省略する。 ■

その反対に Crout タイプの LU 分解や Dolittle タイプの LU 分解から LDU 分解を求めることができる。例えば Crout タイプの LU 分解

$$A = LU,$$

$$L = \begin{pmatrix} 1 & & & 0 \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{N1} & \cdots & \ell_{N,N-1} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1N} \\ & u_{22} & \ddots & \vdots \\ & & \ddots & u_{N-1,N} \\ 0 & & & u_{NN} \end{pmatrix}$$

があるとき、

$$\widetilde{u}_{kj} \stackrel{\text{def.}}{=} \frac{u_{kj}}{u_{kk}} \quad (k = 1, 2, \cdots, N;\ j = k+1, k+2, \cdots, N),$$

$$D \stackrel{\text{def.}}{=} \begin{pmatrix} u_{11} & & & 0 \\ & u_{22} & & \\ & & \ddots & \\ 0 & & & u_{NN} \end{pmatrix}, \quad \widetilde{U} = \begin{pmatrix} 1 & \widetilde{u}_{12} & \cdots & \widetilde{u}_{1N} \\ & 1 & \ddots & \vdots \\ & & \ddots & \widetilde{u}_{N-1,N} \\ 0 & & & 1 \end{pmatrix}$$

とおくと、

$$A = LD\widetilde{U}.$$

### 5.3.5 意義

$A$ が $A = LU$ と LU 分解されているとき、連立 1 次方程式

$$Ax = b$$

は少ない計算量で解くことが出来る。以下、このことを説明する (早い話、$A^{-1} = (LU)^{-1} = U^{-1}L^{-1}$ であり、三角行列である $L, U$ の逆行列が効率良く計算できるということである)。

$$LUx = b$$

は

$$Ly = b, \quad Ux = y$$

という二つの問題に分解される。

$Ly = b$ は

$$\begin{array}{lllll} \ell_{11}y_1 & & & & = b_1 \\ \ell_{21}y_1 & +\ell_{22}y_2 & & & = b_2 \\ \ell_{31}y_1 & +\ell_{32}y_2 & +\ell_{33}y_3 & & = b_3 \\ \vdots & & & & \vdots \\ \ell_{n1}y_1 & +\ell_{n2}y_2 & +\ell_{n3}y_3 & \cdots \quad +\ell_{nn}y_n & = b_n \end{array}$$

ということであり、これは上から順に

$$\begin{aligned} y_1 &= b_1/\ell_{11}, \\ y_2 &= (b_2 - \ell_{21}y_1)/\ell_{22}, \\ y_3 &= (b_3 - \ell_{31}y_1 - \ell_{32}y_2)/\ell_{33}, \\ &\cdots \\ y_i &= \left(b_i - \sum_{j=1}^{i-1} \ell_{ij}y_j\right)/\ell_{ii}, \\ &\cdots \\ y_n &= \left(b_n - \sum_{j=1}^{n-1} \ell_{nj}y_j\right)/\ell_{nn} \end{aligned}$$

と解くことが出来る。

これを計算するには、$1 + 2 + \cdots + n = n(n+1)/2$ 回の乗除算で十分である。

同様に $Ux = y$ は

$$\begin{array}{llllll} u_{11}x_1 & \cdots & +u_{1,n-2}x_{n-2} & +u_{1,n-1}x_{n-1} & +u_{1n}x_n & = y_1, \\ \cdots & & & & & \\ & & u_{n-2,n-2}x_{n-2} & +u_{n-2,n-1}x_{n-1} & +u_{n-2,n}x_n & = y_{n-2}, \\ & & & u_{n-1,n-1}x_{n-1} & +u_{n-1,n}x_n & = y_{n-1}, \\ & & & & u_{n,n}x_n & = y_n \end{array}$$

なので、下から順に

$$\begin{aligned} x_n &= y_n/u_{nn}, \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}, \\ x_{n-2} &= (y_{n-2} - u_{n-2,n-1}x_{n-1} - u_{n-2,n}x_n)/u_{n-2,n-2}, \\ &\cdots \\ x_i &= \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)/u_{ii}, \\ &\cdots \\ x_1 &= \left(y_1 - \sum_{j=2}^{n} u_{1j}x_j\right)/u_{11} \end{aligned}$$

と解くことができる。

これも計算するには、$1 + 2 + \cdots + n = n(n+1)/2$ 回の乗除算で十分である。

まとめると、$n(n+1)/2 + n(n+1)/2 = n(n+1)$ 回の乗除算で連立 1 次方程式が解けることになる。

普通 Gauss の消去法で $n$ 元連立 1 次方程式を解くには、$n^3/3$ 回程度の乗除算が必要であったことを思い出すと、LU 分解があると、いかに効率的に連立 1 次方程式が解けるかが分かる。

逆行列 $A^{-1}$ が得られている場合に、$A^{-1}b$ を計算するのに普通は $n^2$ 回の乗算が必要であるから、LU 分解が得られていると、逆行列をかけて解を得るのと、ほぼ同等の効率で解が得られることが分かる。

そこで (LU 分解がそれほど具合いの良いものであるならば)、正則行列 $A$ が与えられたとき、

- LU 分解は存在するか
- LU 分解が存在するとしたら、どのような方法で求められるか
- LU 分解を求めるにはどれくらいの計算量が必要か

が問題となる。実は任意の正則行列 $A$ の LU 分解が存在するとは限らない (存在するための条件については後述する)。しかし $A$ の行を適当に入れ替えた行列 (置換行列 $P$ を用いて $PA$ と書ける行列) の LU 分解はつねに存在する (これも後述する)。実は Gauss の消去法における係数行列の前進消去の部分は、LU 分解のアルゴリズムとなっている (すぐ後で説明する)。したがって $n^3/3 + O(n^2)$ の乗除算で十分である。

こうしてみると、$A$ の逆行列を求めるための計算量がどれくらいか気になるが、線形代数の講義で学ぶ掃き出し法では、約 $n^3$ 回の乗除算が必要になる (それをここで示しはしないが、興味があれば自分でやってみると良い)。

**注意 5.3.2 (実際の応用ではもっと差がつく場合が多い)** 工学における実際の応用では、行列 $A$ は疎であることが多く (特に微分方程式の数値シミュレーションをする場合に現われる行列はほとんどすべてそうであると言って良い)、その場合に、逆行列は疎であることが期待できないが、$L, U$ は疎となるので、逆行列を求めるよりも LU 分解を利用する方が断然効率的になる。

**注意 5.3.3 (少しでも逆行列に負けるのはしゃくなので)** LU 分解では、$L$ または $U$ の対角成分をすべて 1 であるようにできるので、乗除算の回数は $n(n+1)/2 + n(n-1)/2 = n^2$ と、逆行列を用いる場合と、まったく同等に出来る。つまり、たくさんのベクトルに $A^{-1}$ をかける必要がある場合に、$A^{-1}$ を求める必要はなく、$A$ の LU 分解があればよい。

**連立 1 次方程式を解くためには逆行列は無用の長物である**

## 5.4 Gauss の消去法と LU 分解の関係

以下 Gauss の消去法と LU 分解の関係を述べることを目標にする (結論を先走って述べると、Gauss の消去法は LU 分解のアルゴリズムともなる、つまり LU 分解を求めるには、Gauss の消去法を実行すればよい)。そのため Gauss の消去法を復習しよう。 $Ax = b$ ($b \in \mathbf{R}^n$, $A \in M(n; \mathbf{R})$) を解くために、最初に $A, b$ を並べた行列 $(A\ b)$ を作り、

1. 前進消去
2. 後退代入

を施すのであった。

### 5.4.1 前進消去

$k = 1, 2, \cdots, n-1$ の順に、第 $k$ 行で第 $i$ 行 $(i = k+1, \cdots, n)$ を掃き出すという操作を続けて、上三角行列に変形する。

$$(A\ b) = (A^{(1)}\ b^{(1)}),$$

$$(A^{(1)}\ b^{(1)}) \to (A^{(2)}\ b^{(2)}) \cdots \to (A^{(k)}\ b^{(k)}) \to (A^{(k+1)}\ b^{(k+1)}) \cdots \to (A^{(n)}\ b^{(n)}),$$

$$A^{(n)} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ 0 & & & u_{nn} \end{pmatrix} \stackrel{\text{def.}}{=} U, \quad b^{(n)} = y.$$

$A^{(k)} = (a_{ij}^{(k)})$, $b^{(k)} = (b_i^{(k)})$ とおくと、次のようにまとめられる。

**前進消去の手順**

(1)

$$a_{ij}^{(1)} \stackrel{\text{def.}}{=} a_{ij}, \quad b_i^{(1)} \stackrel{\text{def.}}{=} b_i \quad (i, j = 1, 2, \cdots, n)$$

(2) 以下 $A^{(k)} = (a_{ij}^{(k)})$, $b^{(k)} = (b_i^{(k)})$ を漸化式で定める。すなわち $k = 1, 2, \cdots, n-1$ の順に

(5.1)
$$a_{ij}^{(k+1)} \stackrel{\text{def.}}{=} \begin{cases} a_{ij}^{(k)} & (1 \le i \le k;\ 1 \le j \le n) \\ a_{ij}^{(k)} - \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)} & (k+1 \le i \le n;\ 1 \le j \le n), \end{cases}$$

(5.2)
$$b_i^{(k+1)} \stackrel{\text{def.}}{=} \begin{cases} b_i^{(k)} & (1 \le i \le k) \\ b_i^{(k)} - \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)} & (k+1 \le i \le n), \end{cases}$$

(3) $U = (u_{ij})$, $y = (y_i)$ とおくと、

$$u_{ij} = a_{ij}^{(n)}, \quad y_i^* = b_i^{(n)}.$$

プログラミングでは、行列を表す一つ変数を用意し、$A^{(k)} = (a_{ij}^{(k)})$ $(k = 1, 2, \cdots, n)$ を順番に上書きして計算していけばよい。C 言語風に表すと次のようになる。

**前進消去のプログラム**

```
for (k = 1; k < n; k++)
  for (i = k + 1; i <= n; i++) {
    q = a[i][k] / a[k][k];
    for (j = 1; j <= n; j++)}
      a[i][j] = a[i][j] - q * a[k][j];
    b[i] = b[i] - q * b[k];
  }
```

### 5.4.2 後退代入

**後退代入の手順**

$$x_n = \frac{b_n^*}{a_{nn}^*}, \tag{5.3}$$

$$x_k = \frac{1}{a_{kk}^*}\left(b_k^* - \sum_{j=k+1}^{n} a_{kj}^* x_j\right) \quad (k = n-1, n-2, \cdots, 2, 1 \text{ の順に計算}). \tag{5.4}$$

これも C 言語風プログラムで表わすと以下のようになる。

**後退代入のプログラム**

```
x[n] = b[n] / a[n][n];
for (k = n - 1; k >= 1; k--) {
  s = b[k];
  for (j = k + 1; j <= n; j++)
    s = s - a[k][j] * x[j];
  x[k] = s / a[k][k];
}
```

### 5.4.3 Gauss の消去法のプログラム

以上をまとめると、次のような Gauss の消去法のプログラムができる。

```
/*
 * gauss-ver1.c
 */

#include <matrix.h>

void gauss(int n, matrix a, vector x)
{
    int i, j, k;
    double q;
    /* 前進消去 */
    for (k = 1; k < n; k++)
        for (i = k + 1; i <= n; i++) {
            q = a[i][k] / a[k][k];
#ifdef  ORIGINAL
            for (j = 1; j <= n; j++)
#else
            for (j = k + 1; j <= n; j++)
#endif
                a[i][j] -= q * a[k][j];
            x[i] -= q * x[k];
        }
    /* 後退代入 */
    x[n] /= a[n][n];
    for (k = n - 1; k >= 1; k--) {
        for (j = k + 1; j <= n; j++)
            x[k] -= a[k][j] * x[j];
        x[k] /= a[k][k];
    }
}
```

掃き出しで 0 になると分かり切っているところの計算は省略できることに注意すると、前進消去の j についてのループを、1 でなく k+1 から始めても構わないことが分かる。

### 5.4.4 Gauss の消去法は LU 分解をしていること

$$(A^{(k)}\ b^{(k)}) \to (A^{(k+1)}\ b^{(k+1)})$$

における基本変形を考えると、第 $k$ 行で、第 $i$ 行 $(k+1 \le i \le n)$ を掃き出している。これは、左から基本行列をかけることになる。

$$A^{(k+1)} = L_n^{(k)} L_{n-1}^{(k)} \cdots L_{k+1}^{(k)} A^{(k)}, \quad b^{(k+1)} = L_n^{(k)} L_{n-1}^{(k)} \cdots L_{k+1}^{(k)} b^{(k)},$$

ここで $L_i^{(k)}$ は、第 $k$ 行で第 $i$ 行を掃き出す基本変形を表す行列である。

$$L_i^{(k)} = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & -q_i^{(k)} & & 1 & \\ & & & & & \ddots \\ & & & & & & 1 \end{pmatrix} = I - q_i^{(k)} E_{ik}, \quad q_i^{(k)} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}},$$

$$E_{ik} = \text{第 } (i,k) \text{ 成分が 1 に等しく、他の成分は 0 である行列.}$$

この $L_i^{(k)}$ を用いて

$$L^{(k)} \stackrel{\text{def.}}{=} L_n^{(k)} L_{n-1}^{(k)} \cdots L_{k+1}^{(k)}$$

とおくと、$L^{(k)}$ は第 $k$ 行での掃き出し操作を表す下三角行列である:

$$A^{(k+1)} = L^{(k)} A^{(k)}, \quad b^{(k+1)} = L^{(k)} b^{(k)}.$$

さらに

$$\mathcal{L} \stackrel{\text{def.}}{=} L^{(n-1)} L^{(n-2)} \cdots L^{(2)} L^{(1)}$$

とおくと、

$$A^{(n)} = \mathcal{L} A^{(1)}, \quad b^{(n)} = \mathcal{L} b^{(1)}.$$

言い替えると

$$U = \mathcal{L} A, \quad y = \mathcal{L} b.$$

これから

$$L \stackrel{\text{def.}}{=} \mathcal{L}^{-1}$$

とおけば

$$A = LU$$

となるが、実は

$$L = \begin{pmatrix} 1 & & & & \\ q_2^{(1)} & 1 & & & \\ q_3^{(1)} & q_3^{(2)} & & & \\ \vdots & \vdots & \ddots & & \\ & & & 1 & \\ q_n^{(1)} & q_n^{(2)} & \cdots & q_n^{(n-1)} & 1 \end{pmatrix} \tag{5.5}$$

となり、$L$ は下三角行列である。つまり $A$ の LU 分解が得られた。

**(5.5) の証明**　まず $\mathcal{L} = L^{(n-1)}L^{(n-2)}\cdots L^{(2)}L^{(1)}$ であるから、

$$L = \mathcal{L}^{-1} = (L^{(1)})^{-1}(L^{(2)})^{-1}\cdots(L^{(n-2)})^{-1}(L^{(n-1)})^{-1}.$$

また $L^{(k)} = L_n^{(k)}L_{n-1}^{(k)}\cdots L_{k+1}^{(k)}$ から

$$(L^{(k)})^{-1} = (L_{k+1}^{(k)})^{-1}\cdots(L_{n-1}^{(k)})^{-1}(L_n^{(k)})^{-1}$$

であるが、

$$(L_i^{(k)})^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & \ddots & & & \\ & & q_i^{(k)} & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

となる。これは計算しても分かるが、基本行列の意味を考えれば明らかである。同様に基本行列の意味を考えることにより、

$$(L^{(k)})^{-1} = (L_{k+1}^{(k)})^{-1}\cdots(L_{n-1}^{(k)})^{-1}(L_n^{(k)})^{-1} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & q_{k+1}^{(k)} & \ddots & \\ & & \vdots & & \ddots \\ & & q_n^{(k)} & & 1 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & & & & \\ q_2^{(1)} & 1 & & & \\ q_3^{(1)} & q_3^{(2)} & & & \\ \vdots & \vdots & \ddots & & \\ & & & 1 & \\ q_n^{(1)} & q_n^{(2)} & \cdots & q_n^{(n-1)} & 1 \end{pmatrix}$$

が分かる。 ■

上の計算を振り替えると、$U$ と

$$q_i^{(k)} \overset{\text{def.}}{=} \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \quad (k = 1, 2, \cdots, n-1;\ i = k+1, \cdots, n)$$

さえ記憶しておけば、新しい $b$ が与えられたときに、$Ax = b$ を解くことが出来ることが分かる。つまり

> **Gauss の消去法の前進消去過程のうちで、元の連立 1 次方程式の係数行列 $A$ の部分と方程式の右辺 $b$ の部分は同時に計算する必要はなく、係数行列の前進消去部分だけ先に済ませておくことができる。それが実は LU 分解に他ならない。**

### 5.4.5 LU 分解のプログラムと実験

**素朴なプログラム** lu-ver1.c

```
/*
 * lu-ver1.c
 */

#include <matrix.h>
#include "lu-ver1.h"

void lu(int n, matrix a, matrix L, matrix u)
{
    int i, j, k;
    double q;

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            u[i][j] = a[i][j];
            L[i][j] = 0.0;
        }
        L[i][i] = 1.0;
    }
    for (k = 1; k < n; k++)
        for (i = k + 1; i <= n; i++) {
            q = u[i][k] / u[k][k];
            /* 次の j を 1 からとするのは、定義式通りだが、
               実は j を k から取っても OK (k+1 とは出来ない！) */
#ifdef ORIGINAL
            for (j = 1; j <= n; j++)
#else
            for (j = k; j <= n; j++)
#endif
                u[i][j] = u[i][j] - q * u[k][j];
            L[i][k] = q;
        }
}

/* 行列 A の LU 分解を用いて A x = b を解く */
void solve(int n, matrix L, matrix U, vector b, vector x)
{
  /* 以下の計算手順はプリント 4.3.4 を見よ */
  int i, j;
  double sum;
  vector y = new_vector(n + 1);
  /* 前進消去 */
  for (i = 1; i <= n; i++) {
    sum = b[i];
    for (j = 1; j < i; j++) sum -= L[i][j] * y[j];
    y[i] = sum / L[i][i];
  }
  /* 後退代入 */
  for (i = n; i >= 1; i--) {
    sum = y[i];
    for (j = i + 1; j <= n; j++) sum -= U[i][j] * x[j];
    x[i] = sum / U[i][i];
  }
  /* 不要になったメモリーを再利用可能なように解放する */
  free_vector(y);
}
```

このようにして $A = LU$ と LU 分解したとき、その結果は 2 つの行列 $L, U$ であるが、

- $L$ の対角線とその上部にある成分は覚える必要がない (対角線上の成分は 1, 対角線より上にあるものは 0)
- $U$ の対角線の下部にある成分は覚える必要がない (すべて 0 である)

から、実際には自由度は $n^2$ 個しかない (言い換えると、本当に記憶していなければならないのは、$L$ の対角線の下側と、$U$ の対角成分と対角線の上側の成分である)。それを

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ q_2^{(1)} & u_{22} & u_{23} & \cdots & u_{2n} \\ q_3^{(1)} & q_3^{(2)} & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ q_n^{(1)} & q_n^{(2)} & \cdots & q_n^{(n-1)} & u_{nn} \end{pmatrix}$$

のように一つの行列に詰め込んで記憶することがよく行われる。

matrix a の内容を保存しなくて良いならば、次のようにして、$L, U$ のうちの記憶すべき内容を a に上書きするコードが作れる。(j についてのループを k+1 から始めているのがミソである。)

**詰め込みをしたプログラム** lu-ver2.c

```
/*
 * lu-ver2.c
 */

#include <matrix.h>
#include "lu-ver2.h"

void lu(int n, matrix a)
{
    int i, j, k;
    double q;

    for (k = 1; k < n; k++)
        for (i = k + 1; i <= n; i++) {
            q = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - q * a[k][j];
            a[i][k] = q;
        }
}

/* 行列 A の LU 分解を用いて A x = b を解く */
void solve(int n, matrix a, vector b)
{
  /* 以下の計算手順はプリント 4.3.4 を見よ */
  int i, j;

  /* 前進消去 */
  for (i = 1; i <= n; i++)
    for (j = 1; j < i; j++)
      b[i] -= a[i][j] * b[j];
  /* 注意: L_{i,i}=1 なので割り算は必要ない。 */
  /* 後退代入 */
  for (i = n; i >= 1; i--) {
    for (j = i + 1; j <= n; j++)
      b[i] -= a[i][j] * b[j];
    b[i] /= a[i][i];
  }
}
```

**問**　プログラム lu-ver2.c に採用されたアルゴリズムの計算量を求めよ。

**実験** (lu-ver1.c)

util.c

```
/*
 * util.c
 */

#include <stdio.h>
#include <matrix.h>
#include "util.h"

/* n 次正方行列 A を表示する */
void print_matrix(int n, matrix A)
{
  int i, j;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++)
      printf("%10g ", A[i][j]);
    printf("\n");
  }
}

/* n 次元ベクトル x を表示する */
void print_vector(int n, vector x)
{
  int i;
  for (i = 1; i <= n; i++)
    printf("%10g ", x[i]);
  printf("\n");
}

/* n 次正方行列 A, B の積を計算し、AB に代入する */
void mul_matrix(int n, matrix AB, matrix A, matrix B)
{
  /* 各自 */
  int i, j, k;
  double sum;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++) {
      sum = 0.0;
      for (k = 1; k <= n; k++)
        sum += A[i][k] * B[k][j];
      AB[i][j] = sum;
    }
}
```

test-lu-ver1.c

```
/*
 * test-lu-ver1.c --- 行列の LU 分解
 *  gcc -I/usr/local/include -c util.c
 *  gcc -I/usr/local/include -c lu-ver1.c
 *  ccmg test-lu-ver1.c util.o lu-ver1.o
 */

#include <stdio.h>
#include <matrix.h>
#include "util.h"
#include "lu-ver1.h"

int main()
{
  int n;
  matrix a, L, u, Lu;
  vector b, x;
  n = 3;

  /* n 次正方行列 a, L, u, Lu を準備 */
  a = new_matrix(n + 1, n + 1);
  L = new_matrix(n + 1, n + 1);
  u = new_matrix(n + 1, n + 1);
  Lu = new_matrix(n + 1, n + 1);
  /* n 次元ベクトル b, x を準備 */
  b = new_vector(n + 1);
  x = new_vector(n + 1);

  /* 行列 a に値を設定（プリント p.27 を見よ）*/
  a[1][1] =  2; a[1][2] = 3; a[1][3] = -1;
  a[2][1] =  4; a[2][2] = 4; a[2][3] = -3;
  a[3][1] = -2; a[3][2] = 3; a[3][3] = -1;

  /* LU 分解する */
  lu(n, a, L, u);

  /* a, L, u の内容を表示する */
  printf("A=\n");  print_matrix(n, a);
  printf("L=\n");  print_matrix(n, L);
  printf("U=\n");  print_matrix(n, u);

  /* 確認用に L と u の積を計算して表示する */
  mul_matrix(n, Lu, L, u);
  printf("L U=\n"); print_matrix(n, Lu);

  /* 連立 1 次方程式 (p.27) を解く */
  b[1] = 5; b[2] = 3; b[3] = 1;
  solve(n, L, u, b, x);
  printf("x=\n");  print_vector(n, x);

  return 0;
}
```

**コンパイル、実行**

```
oyabun% gcc -I/usr/local/include -c util.c
oyabun% gcc -I/usr/local/include -c lu-ver1.c
oyabun% ccmg test-lu-ver1.c util.o lu-ver1.o
oyabun% ./test-lu-ver1
A=
         2          3         -1
         4          4         -3
        -2          3         -1
L=
         1          0          0
         2          1          0
        -1         -3          1
U=
         2          3         -1
         0         -2         -1
         0          0         -5
L U=
         2          3         -1
         4          4         -3
        -2          3         -1
x=
         1          2          3
oyabun%
```

確かに $LU = A$ となっていて、解も求まっていることが分かる。

### 実験 (lu-ver2.c)

test-lu-ver2.c

```
/*
 * test-lu-ver2.c --- 行列の LU 分解
 *  gcc -I/usr/local/include -c util.c
 *  gcc -I/usr/local/include -c lu-ver2.c
 *  ccmg test-lu-ver2.c util.o lu-ver2.o
 */

#include <stdio.h>
#include <matrix.h>
#include "util.h"
#include "lu-ver2.h"

int main()
{
  int n;
  matrix a;
  vector b;
  n = 3;

  /* n 次正方行列 a を準備 */
  a = new_matrix(n + 1, n + 1);
  /* n 次元ベクトル b, x を準備 */
  b = new_vector(n + 1);

  /* 行列 a に値を設定（プリント p.27 を見よ）*/
  a[1][1] =  2; a[1][2] = 3; a[1][3] = -1;
  a[2][1] =  4; a[2][2] = 4; a[2][3] = -3;
  a[3][1] = -2; a[3][2] = 3; a[3][3] = -1;

  /* a の内容を表示する */
  printf("A=\n");  print_matrix(n, a);
  /* b に値を設定して、表示する */
  b[1] = 5; b[2] = 3; b[3] = 1;
  printf("b=\n");  print_vector(n, b);

  /* LU 分解する */
  lu(n, a);
  /* a の内容を表示する */
  printf("LU 分解後の A=\n");  print_matrix(n, a);

  /* 連立 1 次方程式 (p.27) を解く */
  solve(n, a, b);
  printf("x=\n");  print_vector(n, b);

  return 0;
}
```

**コンパイル、実行**

```
oyabun% gcc -I/usr/local/include -c util.c
oyabun% gcc -I/usr/local/include -c lu-ver2.c
oyabun% ccmg test-lu-ver2.c util.o lu-ver2.o
oyabun% ./test-lui--ver2
A=
         2          3         -1
         4          4         -3
        -2          3         -1
b=
         5          3          1
LU 分解後の A=
         2          3         -1
         2         -2         -1
        -1         -3         -5
x=
         1          2          3
oyabun%
```

## 5.5 部分枢軸選択つきの LU 分解

### 5.5.1 lu-ver4.c

```
/*
 * lu-ver4.c --- 部分枢軸選択をした Gauss 消去法に基づく LU 分解
 *
 * 行列 A が与えられたとき、部分枢軸選択をした Gauss 消去法のアルゴリズムで
 *
 *   P A = L U, (P は置換行列， L は下三角行列，U は上三角行列)
 *
 * を満たす P, L, U を求める。
 *
 */

#include <math.h>
#include <matrix.h>
#include "lu-ver4.h"

void ipvt2perm(int n, ivector perm, ivector ipvt)
{
  int i, m, perm_i;
  for (i = 1; i <= n; i++)
    perm[i] = i;
  for (i = 1; i < n; i++) {
    m = ipvt[i];
    if (i != m) {
      perm_i = perm[i]; perm[i] = perm[m]; perm[m] = perm_i;
    }
  }
}

/* 部分枢軸選択つきの Gauss の消去法に基づく LU 分解 */
void lu(int n, matrix a, ivector ipvt)
{
  int i, j, k, m;
  double q, t, maxpvt;

  /* 置換の符号 */
  ipvt[n] = 1;
  /* 前進消去 */
```

```
  for (k = 1; k < n; k++) {
    /* a[i][k] (i=k,k+1,...,n) の中で絶対値最大のものを見い出す */
    maxpvt = fabs(a[k][k]); m = k;
    for (i = k + 1; i <= n; i++)
      if (fabs(a[i][k]) > maxpvt) {
        maxpvt = fabs(a[i][k]); m = i;
      }
    ipvt[k] = m;
    /* 第 m 行に最大のものがある */
    if (k != m) {
      /* 第 k 行と第 m 行を交換する */
      for (j = 1; j <= n; j++) {
        t = a[k][j]; a[k][j] = a[m][j]; a[m][j] = t;
      }
      /* 互換を施すので、置換の符号が変わる */
      ipvt[n] = - ipvt[n];
    }
    /* 枢軸を用いて消去 */
    for (i = k + 1; i <= n; i++) {
      q = a[i][k] / a[k][k];
      for (j = k + 1; j <= n; j++)
        a[i][j] = a[i][j] - q * a[k][j];
      a[i][k] = q;
    }
  }
}

/* 行列 A の LU 分解 (lu() で得られたもの) を用いて A x = b を解く */
void solve(int n, matrix a, vector b, ivector ipvt)
{
  int i, j;

  for (i = 1; i < n; i++) {
    int m;
    double t;
    m = ipvt[i];
    if (i != m) {
      t = b[m]; b[m] = b[i]; b[i] = t;
    }
  }
  /* 前進消去 */
  for (i = 1; i <= n; i++)
    for (j = 1; j < i; j++)
      b[i] -= a[i][j] * b[j];
  /* 注意: L_{i,i}=1 なので割り算は必要ない。 */
  /* 後退代入 */
  for (i = n; i >= 1; i--) {
    for (j = i + 1; j <= n; j++)
      b[i] -= a[i][j] * b[j];
    b[i] /= a[i][i];
  }
}

/* ipvt[] で与えられた置換行列 P を行列 A に左からかける */
void permute_matrix(int n, matrix pa, matrix a, ivector ipvt)
{
  ivector perm = new_ivector(n + 1);
  ipvt2perm(n, perm, ipvt);
  permute_matrix0(n, pa, a, perm);
  free_ivector(perm);
}

/* perm[] で与えられた置換行列 P を行列 A に左からかける */
```

```
void permute_matrix0(int n, matrix pa, matrix a, ivector perm)
{
  int i, j, m;
  for (i = 1; i <= n; i++) {
    m = perm[i];
    for (j = 1; j <= n; j++)
      pa[i][j] = a[m][j];
  }
}
```

### 5.5.2 数値実験 (1)

#### test-lu-ver4.c

```
/*
 * test-lu-ver4.c --- 行列の LU 分解
 *  gcc -I/usr/local/include -c util.c
 *  gcc -I/usr/local/include -c lu-ver4.c
 *  ccmg test-lu-ver4.c util.o lu-ver4.o
 */

#include <stdio.h>
#include <matrix.h>
#include "util.h"
#include "lu-ver4.h"

int main()
{
  int n;
  matrix a;
  vector b;
  ivector ipvt;
  n = 3;

  /* n 次正方行列 a を準備 */
  a = new_matrix(n + 1, n + 1);
  /* n 次元ベクトル b, x を準備 */
  b = new_vector(n + 1);
  /* 置換を記憶するためのベクトルを準備 */
  ipvt = new_ivector(n + 1);

  /* 行列 a に値を設定 (プリント p.27 を見よ) */
  a[1][1] =  2; a[1][2] = 3; a[1][3] = -1;
  a[2][1] =  4; a[2][2] = 4; a[2][3] = -3;
  a[3][1] = -2; a[3][2] = 3; a[3][3] = -1;

  /* a の内容を表示する */
  printf("A=\n");  print_matrix(n, a);
  /* b に値を設定して、表示する */
  b[1] = 5; b[2] = 3; b[3] = 1;
  printf("b=\n");  print_vector(n, b);

  /* LU 分解する */
  lu(n, a, ipvt);
  /* a の内容を表示する */
  printf("LU 分解後の A=\n");  print_matrix(n, a);

  /* 連立 1 次方程式 (p.27) を解く */
  solve(n, a, b, ipvt);
  printf("x=\n");  print_vector(n, b);

  return 0;
```

```
}
```

test-lu-ver4 の実行結果

```
oyabun% ./test-lu-ver4
A=
          2          3         -1
          4          4         -3
         -2          3         -1
b=
          5          3          1
LU 分解後の A=
          4          4         -3
       -0.5          5       -2.5
        0.5        0.2          1
x=
          1          2          3
oyabun%
```

### 5.5.3 数値実験 (2)

**test-lu4.c**

```
/*
 * test-lu4.c --- LU 分解のチェック
 *
 *  コンパイルするには、
 *   例: gcc -o test-lu4 test-lu4.o lu-ver4.o matrix-base1.o
 *          -L/usr/local/lib -lmatrix -lm
 */

#include <stdio.h>  /* 標準入出力 standard input & output */
#include <math.h>   /* 数学関数など */

#include <matrix.h> /* 桂田が書いた matrix 関係の型、関数の宣言 */
#include "lu-ver4.h"
#include "matrix-base1.h"

int main()
{
    /* 変数宣言文 */
    int n;
    matrix a, LandU, Lu, diff, pa;
    ivector perm, ipvt;
    double matrix_norm_2();

    printf("n 次の Hilbert 行列の LU 分解をする。\n");
    printf("n="); scanf("%d", &n);
    /* 記憶のための場所を確保する (メモリーをアロケートする) */
    a = new_matrix(n + 1, n + 1);
    LandU = new_matrix(n + 1, n + 1);
    Lu = new_matrix(n + 1, n + 1);
    diff = new_matrix(n + 1, n + 1);
    pa = new_matrix(n + 1, n + 1);
    perm = new_ivector(n + 1);
    ipvt = new_ivector(n + 1);

    /* この後は配列のように使える */
#ifdef OLD
    {
      int i, j;
      for (i = 1; i <= n; i++)
```

```
        for (j = 1; j <= n; j++) {
            printf("a[%d][%d]=", i, j);
            scanf("%lf", &a[i][j]);
        }
    }
#else
    hilbert(n, a);
#endif

    printf("A =\n");
    display_matrix(n, a, "%f ");

    /* LU 分解する */
    copy_matrix(n, LandU, a);
    lu(n, LandU, ipvt);
    printf("L, U を詰め込んだ行列\n");
    display_matrix(n, LandU, "%f ");

    /* L と U の積を計算して表示する */
    multiply_LUmatrix(n, Lu, LandU);
    printf("L U=\n");
    display_matrix(n, Lu, "%f ");

    /* P と A の積を計算して表示する */
    permute_matrix(n, pa, a, ipvt);
    printf("P A=\n");
    display_matrix(n, pa, "%f ");

    /* P A と L U が等しいことの確認 */
    subtract_matrix(n, diff, pa, Lu);
    printf("P A - L U=\n");
    display_matrix(n, diff, "%g ");
    printf("||P A - L U||=%g\n", matrix_norm_2(n, diff));

    return 0;
}
```

test-lu4 の実行結果

```
oyabun% ./test-lu4
n 次の Hilbert 行列の LU 分解をする。
n=3
A =
1.000000 0.500000 0.333333
0.500000 0.333333 0.250000
0.333333 0.250000 0.200000
L, U を詰め込んだ行列
1.000000 0.500000 0.333333
0.333333 0.083333 0.088889
0.500000 1.000000 -0.005556
L U=
1.000000 0.500000 0.333333
0.333333 0.250000 0.200000
0.500000 0.333333 0.250000
P A=
1.000000 0.500000 0.333333
0.333333 0.250000 0.200000
0.500000 0.333333 0.250000
P A - L U=
0 0 0
0 0 0
0 0 0
||P A - L U||=0
oyabun%
```

## 5.6 Gauss の消去法の優秀性

以下簡単に

1. Cramer の公式
2. 掃き出し法 (Jordan の消去法)
3. Gauss の消去法

という連立 1 次方程式の 3 つの解法の比較をしよう。

Cramer の公式を適用するには $n+1$ 個の行列式を求める必要があるため、計算の手間がかかる (大きな計算量が必要になる)。実際、行列式を一つ計算するための手間は、連立方程式を一つ解くための手間と本質的に同等であることが分かっているので、Cramer の公式を使うことに固執すると、本来必要な計算量の $n$ 倍程度の計算をする羽目になり、大変な損をすることになる (差分法などの計算に現れる連立 1 次方程式では、$n$ が非常に大きな数になることに注意しよう)。そのため、実際の数値計算では、ごく特殊な例を除いて、Cramer の公式が利用されることはない。Cramer の公式は、理論的な問題を扱う場合に真価が発揮されるものである [2]。

この Cramer の方法に比べれば、掃き出し法 (通常の消去法) は、かなりの実用性を持っているが、空間 1 次元の熱伝導方程式に対する差分方程式のように、係数行列が三重対角行列の場合には、Gauss の消去法の方が断然有利である。それは、Gauss の消去法を採用すると、掃き出しの途中に現れる行列が三重対角のままであることから、計算量が少なくてすむためである。

[2] いくつか初等数学に現われる例をあげておくと、整数を成分にもつ行列の行列式が $\pm 1$ であれば、逆行列の成分もすべて整数であるとか、逆関数定理における逆関数の微分可能性とか。

## 5.7　3 項方程式を解くためのプログラム

三重対角行列を係数とする連立 1 次方程式を 3 項方程式と呼ぶのであった。熱方程式の初期値境界値問題を $\theta$ 法で解く際に現れる方程式が典型例であるが、これを解くためのプログラムを以下に示そう。

### 5.7.1　3 項方程式を解くアルゴリズム

やはり Gauss の消去法で LU 分解するのが良い。(一部では、それを Thomas の方法と呼ぶらしいが[3]、わざわざそういう名前にするのは何故なんだろう？)

三重対角行列を LU 分解するとき、ピボットの交換なしで済めば、帯幅は増えないので、結果として得られる下三角行列、上三角行列は対角線とその一つ隣だけが非零成分を含むようになる。このことは簡単に証明できるが、実例で見てみよう。

**三重対角行列を LU 分解すると三重対角行列**

```
mathpc00% make test-lu-trid
gcc -O -pipe -c test-lu-trid.c
gcc -O -pipe -c lu-ver2.c
gcc -O -pipe -c util.c
gcc -o test-lu-trid test-lu-trid.o lu-ver2.o util.o -L/usr/local/lib -lmatrix -lm
mathpc00% ./test-lu-trid
n=7
A=
          2       -0.5          0          0          0          0          0
       -0.5          2       -0.5          0          0          0          0
          0       -0.5          2       -0.5          0          0          0
          0          0       -0.5          2       -0.5          0          0
          0          0          0       -0.5          2       -0.5          0
          0          0          0          0       -0.5          2       -0.5
          0          0          0          0          0       -0.5          2
LU 分解後の A=
          2       -0.5          0          0          0          0          0
      -0.25      1.875       -0.5          0          0          0          0
          0  -0.266667    1.86667       -0.5          0          0          0
          0          0  -0.267857    1.86607       -0.5          0          0
          0          0          0  -0.267943    1.86603       -0.5          0
          0          0          0          0  -0.267949    1.86603       -0.5
          0          0          0          0          0  -0.267949    1.86603
mathpc00%
```

### 5.7.2　ピボットの選択について

以下に 3 項方程式を解くためのプログラム (菊地・山本 [1] にある解説と Fortran プログラムを参考にした) を紹介するが、ここでは**枢軸選択** (**ピボットの選択**、**pivoting**) を行わない。

これが正当である理由は… (準備中。ピボットが 0 にならないことが、例えば Smith [8] の本に載っている。ところで、古い本 (Wilkinson とか) を見ると、正値対称行列に対して消去法を施す場合は、ピボット選択が不要だと書いてあることが多いが、丸め誤差の観点からはそんなに簡単に割りきれるものではなく、やはりきちんとピボット選択をすべき場合も多いのだとか言う人もいる (一松先生など)。というわけで、きちんとしたことを書くには時間がかかりそうなので、ここは知らんぷりしておく。)

[3] Smith [8] などを見よ。

### 5.7.3 FORTRAN 77 によるプログラム

ここでは菊地・山本 [1] を少し書き換えたプログラムを示す。

```
      subroutine trid(a,b,c,f,n,id)
      integer n,id
      real a(*),b(*),c(*),f(*)
      integer i
c     前進消去 (forward elimination)
      if (id .eq. 0) then
        do i=1,n-1
          b(i+1)=b(i+1)-c(i)*a(i+1)/b(i)
        end do
      endif
      do i=1,n-1
        f(i+1)=f(i+1)-f(i)*a(i+1)/b(i)
      end do
c     後退代入 (backward substitution)
      f(n)=f(n)/b(n)
      do i=n-1,1,-1
        f(i)=(f(i)-c(i)*f(i+1))/b(i)
      end do
      end
```

### 5.7.4 C によるプログラム (1)

添字が 0 から始まる版。

```
/* trid-lu.c -- 3 項方程式を Gauss の消去法で解く */

#include "trid-lu.h"

/* 3 項方程式 (係数行列が三重対角である連立 1 次方程式のこと) Ax=b を解く
 *
 *   入力
 *     n: 未知数の個数
 *     al,ad,au: 連立 1 次方程式の係数行列
 *       (al: 対角線の下側 i.e. 下三角部分 (lower part)
 *        ad: 対角線       i.e. 対角部分 (diagonal part)
 *        au: 対角線の上側 i.e. 上三角部分 (upper part)
 *        つまり
 *
 *         ad[0] au[0]   0   ................. 0
 *         al[1] ad[1] au[1]   0  ............. 0
 *           0   al[2] ad[2] au[2]  0 ......... 0
 *                         ...................
 *                         al[n-2] ad[n-2] au[n-2]
 *                           0     al[n-1] ad[n-1]

 *        al[i] = A_{i,i-1}, ad[i] = A_{i,i}, au[i] = A_{i,i+1},
 *        al[0], au[n-1] は意味がない)
 *
 *     b: 連立 1 次方程式の右辺の既知ベクトル
 *        (添字は 0 から。i.e. b[0],b[1],...,b[n-1] にデータが入っている。)
 *   出力
 *     al,ad,au: 入力した係数行列を LU 分解したもの
 *     b: 連立 1 次方程式の解
 *   能書き
 *     一度 call すると係数行列を LU 分解したものが返されるので、
 *     以後は同じ係数行列に関する連立 1 次方程式を解くために、
 *     関数 trisol() が使える。
 *   注意
```

```
 *      Gauss の消去法を用いているが、ピボットの選択等はしていな
 *      いので、ピボットの選択をしていないので、係数行列が正定値である
 *      などの適切な条件がない場合は結果が保証できない。
 */

void trid(int n, double *al, double *ad, double *au, double *b)
{
  trilu(n,al,ad,au);
  trisol(n,al,ad,au,b);
}

/* 三重対角行列の LU 分解 (pivoting なし) */
void trilu(int n, double *al, double *ad, double *au)
{
  int i, nm1 = n - 1;
  /* 前進消去 (forward elimination) */
  for (i = 0; i < nm1; i++) {
    al[i + 1] /= ad[i];
    ad[i + 1] -= au[i] * al[i + 1];
  }
}

/* LU 分解済みの三重対角行列を係数に持つ 3 項方程式を解く */
void trisol(int n, double *al, double *ad, double *au, double *b)
{
  int i, nm1 = n - 1;
  /* 前進消去 (forward elimination) */
  for (i = 0; i < nm1; i++) b[i + 1] -= b[i] * al[i + 1];
  /* 後退代入 (backward substitution) */
  b[nm1] /= ad[nm1];
  for (i = n - 2; i >= 0; i--) b[i] = (b[i] - au[i] * b[i + 1]) / ad[i];
}
```

**メモ** 2000 年 11 月 4 日、C 版を書き換えた。もともと trisol() にあった計算の一部を trilu() に移した。同じ係数行列を持つ連立 1 次方程式を何度も解く場合には、全体の計算量を少なくすることができる。そういうわけで、現在では Fortran 版と C 版では関数の仕様そのものが異なっている (いわゆる LU 分解をしているのは C 版の方である)。

### 5.7.5 C によるプログラム (2)

添字が 1 から始まる版。

```
/* trid-lu1.c -- 3 項方程式を Gauss の消去法で解く */

#include "trid-lu1.h"

/* 3 項方程式 (係数行列が三重対角である連立 1 次方程式のこと) Ax=b を解く
 *
 *   入力
 *     n: 未知数の個数
 *     al,ad,au: 連立 1 次方程式の係数行列
 *       (al: 対角線の下側 i.e. 下三角部分 (lower part)
 *        ad: 対角線       i.e. 対角部分 (diagonal part)
 *        au: 対角線の上側 i.e. 上三角部分 (upper part)
 *        つまり
 *
 *         ad[1] au[1]   0   ................. 0
 *         al[2] ad[2] au[2]   0  ............ 0
 *           0   al[3] ad[3] au[3]  0 ........ 0
 *                         ...................
```

```
 *                          al[n-1] ad[n-1] au[n-1]
 *                            0     al[n]   ad[n]

 *        al[i] = A_{i,i-1}, ad[i] = A_{i,i}, au[i] = A_{i,i+1},
 *        al[1], au[n] は意味がない)
 *
 *     b: 連立 1 次方程式の右辺の既知ベクトル
 *        (添字は 1 から。i.e. b[1],b[2],...,b[n] にデータが入っている。)
 *   出力
 *     al,ad,au: 入力した係数行列を LU 分解したもの
 *     b: 連立 1 次方程式の解
 *   能書き
 *     一度 call すると係数行列を LU 分解したものが返されるので、
 *     以後は同じ係数行列に関する連立 1 次方程式を解くために、
 *     関数 trisol1() が使える。
 *   注意
 *     Gauss の消去法を用いているが、ピボットの選択等はしていな
 *     いので、ピボットの選択をしていないので、係数行列が正定値である
 *     などの適切な条件がない場合は結果が保証できない。
 */

void trid1(int n, double *al, double *ad, double *au, double *b)
{
  trilu1(n,al,ad,au);
  trisol1(n,al,ad,au,b);
}

/* 三重対角行列の LU 分解 (pivoting なし) */
void trilu1(int n, double *al, double *ad, double *au)
{
  int i;
  /* 前進消去 (forward elimination) */
  for (i = 1; i < n; i++) {
    al[i + 1] /= ad[i];
    ad[i + 1] -= au[i] * al[i + 1];
  }
}

/* LU 分解済みの三重対角行列を係数に持つ 3 項方程式を解く */
void trisol1(int n, double *al, double *ad, double *au, double *b)
{
  int i;
  /* 前進消去 (forward elimination) */
  for (i = 1; i < n; i++) b[i + 1] -= b[i] * al[i + 1];
  /* 後退代入 (backward substitution) */
  b[n] /= ad[n];
  for (i = n - 1; i >= 1; i--) b[i] = (b[i] - au[i] * b[i + 1]) / ad[i];
}
```

実は trisol(), trilu(), trid() があれば、trisol1(), trilu1(), trid1() は不要である (以下に示すように前者を呼び出すだけですんでしまうので)。

```
void trisol1(int n, double *al, double *ad, double *au, double *b)
{
  trisol(n, al+1, ad+1, au+1, b+1);
}

void trilu1(int n, double *al, double *ad, double *au)
{
  trilu(n, al+1, ad+1, au+1);
}

void trid1(int n, double *al, double *ad, double *au, double *b)
{
  trilu(n, al+1, ad+1, au+1, b+1);
}
```

## 5.8 帯行列を係数行列とする場合

### 5.8.1 半バンド幅, 帯行列

$N$ 次正方行列 $A = (a_{i,j})$ が半バンド幅 $m$ であるとは、

$$|i-j| > m \quad \Longrightarrow \quad a_{i,j} = 0$$

を満たすことと定義する。この条件の対偶を取ると

$$a_{i,j} \neq 0 \quad \Longrightarrow \quad |i-j| \leq m.$$

(非零成分は、行番号と列番号の差が $m$ 以下のところにしかない。)

**注意 5.8.1** この定義によると、与えられた行列に対して、半バンド幅は一意的に定まるものではない。$A$ が半バンド幅 3 であれば、$A$ は半バンド幅 4 でもある。この意味では用語が少し変だと思う。あるとき、ゼミで某学生が覚えていなくて、とっさに

$$\text{(新) 半バンド幅} = \max_{a_{ij}\neq 0} |i-j|$$

という定義をひねり出した。このやり方は零行列を特別扱いしないといけないが、割と筋が通っているように思う (半バンド幅が一意的に定まる)。ただ応用上は正確な半バンド幅を求めるのはしばしば面倒であり (例えば実際に行列の成分を全部計算し終わってみないと分からないとか)、事前の簡単な考察で分かる半バンド幅の評価を使って計算を進めることが多いので、最初に述べた (通常流布している) 定義で問題ないのであろう。 ■

対角行列は半バンド幅 0, 三重対角行列は半バンド幅 1 である。
$m \ll N$ であるとき、行列 $A$ は帯行列であるという [4]。

### 5.8.2 帯行列の詰め込み

C 言語でプログラムを書くことを念頭に、成分の添字は 0 から始めることにする。

$A = (a_{ij})_{0\leq i,j\leq N-1}$ が半バンド幅 $m$ の $N$ 次正方行列であるとき、$N$ 行 $2m+1$ 列の行列 $\tilde{A} = (\tilde{a}_{ik})_{\substack{0\leq i\leq N-1 \\ 0\leq k\leq 2m}}$ に詰め込むことが出来る。

例えば次のように対応させればよい。

[4]「背の高い人達の集まりは集合ではない」というのと同様の理由で、これは数学的な定義とはいいかねる。

$a_{ij}$ **を** $\widetilde{a}_{ik}$ **で表す**　$0 \le i, j \le N-1$ なる $i, j$ に対して、

$$a_{ij} = \begin{cases} \widetilde{a}_{i,j-i+m} & (|i-j| \le m) \\ 0 & (|i-j| > m). \end{cases}$$

$\widetilde{a}_{ik}$ **を** $a_{ij}$ **で表す**　$0 \le i \le N-1, 0 \le k \le 2m$ なる $i, k$ に対して、

$$\widetilde{a}_{ik} = \begin{cases} a_{i,i+k-m} & (0 \le i+k-m \le N-1) \\ \text{未使用} & (\text{それ以外}). \end{cases}$$

### 5.8.3　対称帯行列の詰め込み

C 言語でプログラムを書くことを念頭に、成分の添字は 0 から始めることにする。

$A = (a_{ij})_{0 \le i,j \le N-1}$ が半バンド幅 $m$ の $N$ 次対称行列であるとき、対角線の上側だけを記憶すれば良いので、$N$ 行 $m+1$ 列の行列 $\tilde{A} = (\tilde{a}_{ik})_{\substack{0 \le i \le N-1 \\ 0 \le k \le m}}$ に詰め込むことが出来る。

$a_{ij}$ **を** $\widetilde{a}_{ik}$ **で表す**　$0 \le i, j \le N-1$ なる $i, j$ に対して、

$$a_{ij} = \begin{cases} 0 & (|i-j| > m) \\ \widetilde{a}_{i,j-i} & (i \le j \le i+m) \\ \widetilde{a}_{j,i-j} & (i-m \le j < i). \end{cases}$$

$\widetilde{a}_{ik}$ **を** $a_{ik}$ **で表す**　$0 \le i \le N-1, 0 \le k \le m$ なる $i, k$ に対して、

$$\widetilde{a}_{ik} = \begin{cases} a_{i,i+k} & (i+k \le N-1) \\ \text{未使用} & (\text{それ以外}). \end{cases}$$

### 5.8.4　対称行列の LU 分解

#### 下三角行列は要らない

まず、対称行列 $A$ の LU 分解 $A = LU$ においては、上三角行列 $U$ さえ記憶しておけば、下三角行列 $U$ は (以下に紹介する公式 (5.6) で) 簡単に求まることを説明しよう。

$A$ が $\det A(1:k, 1:k) \neq 0$ $(k = 1, 2, \cdots, n)$ を満たす行列ならば (例えば $A$ が正値対称であればこの条件は満たされる)、$A$ は

$$A = LU, \quad L = \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ * & & 1 \end{pmatrix}, \quad U = \begin{pmatrix} * & & * \\ & \ddots & \\ 0 & & * \end{pmatrix}$$

のように単位下三角行列と上三角行列の積に分解されるのであった。

以下に示すように、$U$ から $L$ を簡単に計算することができるので、$U$ だけ覚えておけばよい、という結論が得られる。

まず上三角行列 $U$ は容易に、

$$U = DU', \quad D = \begin{pmatrix} * & & 0 \\ & \ddots & \\ 0 & & * \end{pmatrix}, \quad U' = \begin{pmatrix} 1 & & * \\ & \ddots & \\ 0 & & 1 \end{pmatrix}.$$

のように対角行列と単位上三角行列の積に分解され、$A = LDU'$ といういわゆる LDU 分解が得られる。LDU 分解の一意性、すなわち「与えられた行列を単位下三角行列 × 対角行列 × 上三角行列の形に表す仕方は一意的である」という定理を思いだそう。この場合、$A$ が対称であることから

$$A = A^T = (LDU')^T = (U')^T D L^T.$$

これも $A$ の LDU 分解であるが、一意性から $U' = L^T$. ゆえに $U = DU' = DL^T$.

$$\begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ 0 & & & u_{nn} \end{pmatrix} = \begin{pmatrix} d_1 & & & 0 \\ & d_2 & & \\ & & \ddots & \\ 0 & & & d_n \end{pmatrix} \begin{pmatrix} 1 & \ell_{21} & \ell_{31} & \cdots & \ell_{n1} \\ & 1 & \ell_{32} & \cdots & \ell_{n2} \\ & & \ddots & \ddots & \vdots \\ & & & 1 & \ell_{n,n-1} \\ 0 & & & & 1 \end{pmatrix}.$$

ゆえに

$$u_{ii} = d_i \quad (i = 1, \ldots, n), \quad u_{ij} = d_i \ell ji \quad (1 \le i < j \le n).$$

これから、$U$ から $L$ を求める公式

$$\ell_{ji} = u_{ij}/d_i = u_{ij}/u_{ii} \quad (1 \le i < j \le n) \tag{5.6}$$

が得られる。

**コーディングの時間です**

コードを考えて見よう。普通の LU 分解は例えば次のように書ける。

```
/* 普通の LU 分解 */
void lu1(matrix a, int n)
{
  int i, j, k;
  double q;
  for (i = 1; i < n; i++) {
    for (j = i + 1; j <= n; j++) {
      q = a[j][i] / a[i][i];
      for (k = i + 1; k <= n; k++)
        a[j][k] -= q * a[i][k];
      a[j][i] = q;
    }
  }
}

void solve1(matrix a, vector b, int n)
{
  int i, j;
  for (i = 1; i < n; i++) {
    for (j = i + 1; j <= n; j++)
      b[j] -= a[j][i] * b[i];
  }
  b[n] /= a[n][n];
  for (i = n - 1; i >= 1; i--) {
    for (j = i + 1; j <= n; j++)
      b[i] -= a[i][j] * b[j];
    b[i] /= a[i][i];
  }
}
```

$L$ に相当する部分の計算をサボると次のようになる (実はまだまだ甘い)。

```
/* 対称行列用の LU 分解 (暫定バージョン) */
void lu2(matrix a, int n)
{
  int i, j, k;
  double q;
  for (i = 1; i < n; i++) {
    for (j = i + 1; j <= n; j++) {
      q = a[j][i] / a[i][i];
      for (k = i + 1; k <= n; k++)
        a[j][k] -= q * a[i][k];
      /* lu1() では L[j][i] を a[j][i] に記憶していたが、それをサボる。*/
    }
  }
  /* もしも a[i][j] (i<j) が欲しければ、ここからでも次にようにすればよい。
   * for (i=1; i<=n; i++) for (j=1; j<i; j++) a[i][j] = a[j][i] / a[j][j]; */
}

void solve2(matrix a, vector b, int n)
{
  int i, j;
  for (i = 1; i < n; i++) {
    for (j = i + 1; j <= n; j++) {
      /* L[j][i] は a[i][j] / a[i][i] に等しい */
      b[j] -= (a[i][j] / a[i][i]) * b[i];
    }
  }
  b[n] /= a[n][n];
  for (i = n - 1; i >= 1; i--) {
    for (j = i + 1; j <= n; j++)
      b[i] -= a[i][j] * b[j];
    b[i] /= a[i][i];
  }
}
```

よく見ると、この lu2() では、まだ a[][] の対角線の下側を読み書きしている (a[$i$][$j$] ($i > j$) を参照し、また代入している)。

```
void lu3(matrix a, int n)
{
  int i, j, k;
  double q;
  for (i = 1; i < n; i++) {
    for (j = i + 1; j <= n; j++) {
      /* 次の命令で a[j][i] を参照していたが、対称性から a[i][j] にしてよい */
      q = a[i][j] / a[i][i];
      /* 次のループで k は i + 1 からだったけれど、実は k は j からで十分 */
      for (k = j; k <= n; k++)
        a[j][k] -= q * a[i][k];
    }
  }
}

/* solve2() は修正すべき点はない */
```

この段階まで来ると、a[$i$][$j$] ($i > j$) は一切アクセスしなくなっている。ゆえに最初から記憶しておく必要すらないということになる。

### 5.8.5 プログラム例

**luband.c**

以下に掲げるのは卒研の学生 (石川君) が書いたプログラムである。

```
/*  ------ 帯行列の形をした連立方程式を Gauss の消去法で解く -----
 *
 * 係数行列 A が、半バンド幅 m の帯行列である場合、
 * それを利用して連立方程式 Ax=b を解くのを高速化+省メモリ化する。
 * つまり、係数行列の帯の部分を取り出して計算していく。
 * 元の係数行列を (a_{i,j}) とすると、取り出した行列 am との関係は,
 *
 *   am[i][j] = a_{i,i+j-m}   (m ≦ i+j < m+n)
 *            = 0             (i+j < m, m+n ≦ i+j)
 * または、
 *   a_{i,j} = am[i][j-i+m]   (0 ≦ j-i+m < 2m+1)
 */

#include <matrix.h>

/* LU 分解 */
void bandlu(int m, int n, matrix am)
{
    int i, j, k, m1, mm;
    double aa;

    m1 = m - 1;
    for (i = 0; i < m1; i++) {
        mm = i + n + 1;
        if (mm > m) mm = m;
        for (j = i + 1; j < mm; j++) {
            /* aa = a[j][i]/a[i][i]; */
            aa = am[j][i - j + n] / am[i][n];
            for (k = i; k < mm; k++)
                /* a[j][k] -= aa * a[i][k]; */
                am[j][k - j + n] -= aa * am[i][k - i + n];
            am[j][i - j + n] = aa;
        }
    }
}

/* LU 分解したものを使って方程式を解く */
void bandsol(int m, int n, matrix am, vector f)
{
    int i, j, jj, m1, mm;

    m1 = m - 1;
    for (i = 1; i < m; i++) {
        jj = i - n;
        if (jj < 0) jj = 0;
        for (j = jj; j < i; j++)
            f[i] -= f[j] * am[i][j - i + n];
    }

    f[m1] /= am[m1][n];
    for (i = m1 - 1; i >= 0; i--) {
        mm = i + n + 1;
```

```
        if (mm > m) mm = m;
        for (j = i + 1; j < mm; j++)
            f[i] -= f[j] * am[i][j - i + n];
        f[i] /= am[i][n];
    }
}
```

### symbandlu.c

菊地 [4] に対称帯行列用の LU 分解プログラムがある (ただし Fortran で書いてある)。これを C で書き換えたものが次のプログラムである。

```
/*
 * symbandlu.c --- 菊地文雄『有限要素法概説』サイエンス社
 */

#include <matrix.h>

/* the Gauss elimination method for symmetric band matrices */
/* N は未知数の個数、 m は半バンド幅+1 */

void symbandlu(matrix at, int N, int m)
{
    int N1, i, j, k, mm;
    double aa;
    /* forward elimination; */
    N1 = N - 1;
    for (i = 0; i < N1; i++) {
        /* mm = min(i+m,N) */
        mm = i + m; if (mm > N) mm = N;
        for (j = i + 1; j < mm; j++) {
            aa = at[i][j-i] / at[i][0]; /* aa = a_{i,j} / a_{i,i} */
            for (k = j; k < mm; k++)
                at[j][k-j] -= aa * at[i][k-i]; /* a_{j,k} -= aa * a_{i,k} */
        }
    }
}

void symbandsolve(matrix at, vector f, int N, int m)
{
    int N1, i, j, k, mm;
    double aa;
    /* forward elimination; */
    N1 = N - 1;
    for (i = 0; i < N1; i++) {
        mm = i + m; if (mm > N) mm = N;
        for (j = i + 1; j < mm; j++) {
            aa = at[i][j-i] / at[i][0]; /* aa = a_{i,j} / a_{i,i} */
            f[j] -= aa * f[i];
        }
    }
    /* backward substitution */
    f[N-1] /= at[N-1][0]; /* f_{N-1} /= a_{N-1,N-1} */
    for (i = N - 2; i >= 0; i--) {
        mm = i + m;
        if (mm > N) mm = N;
         for (j = i + 1; j < mm; j++)
            f[i] -= at[i][j-i] * f[j]; /* f_i -= at_{i,j} * f_j */
         f[i] /= at[i][0]; /* f_i /= a_{i,i} */
    }
}
```

この中で include されている `symbandlu.h` は以下のようなファイルである。

symbandlu.h

```
/*
 * symbandlu.h
 */

#include <matrix.h>

void band(matrix at, vector f, int N, int m);
void symbandlu(matrix at, int N, int m);
void symbandsolve(matrix at, vector f, int N, int m);
```

一方、線形演算用のサブルーチン・ライブラリィとして有名な LAPACK には、この種のプログラムがたくさんあるので、性能 (効率、信頼性) の高いプログラムが作りたければ、そこから適当なものを選んで利用すべきであろう。(ただし C から LAPACK を利用するのは少し面倒である。ここにある bandlu.c, symbandlu.c のようなプログラムにも存在価値があると思う。LAPACK++ ではどうなるのだろうか…)

ところで symbandlu.c を 1 から書き直したもので置き換えようかと計画している。とりあえず作ったサンプル・プログラムを載せておく。

```
/*
 * bandtest5.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrix.h"

void lu4(matrix, int, int);
void solve4(matrix, vector, int, int);
void testmatrix(int, int, matrix, vector, vector);
void disp(int, int, matrix, vector, char *, char *);

#define FORMAT " %6.4f "

void testmatrix(int n, int m, matrix a, vector x, vector b)
{
  int i, j, mm;
  /* テスト用の A を作る */
  /* テスト用の A を作る */
  for (i = 0; i <= n; i++)
    for (j = 0; j <= n; j++)
      a[i][j] = 0;
  for (i = 1; i <= n; i++) {
    mm = i + m; if (mm > n) mm = n;
    for (j = i + 1; j <= mm; j++) {
      a[i][j-i] = rand() / (double) RAND_MAX;
    }
    a[i][0] = i + 1;
  }
  /* x=(1,2,...,n)^T として b:=A x を計算 */
  for (i = 1; i <= n; i++)
    x[i] = i;
  for (i = 1; i <= n; i++) {
    b[i] = 0;
    mm = i - m; if (mm < 1) mm = 1;
    for (j = mm; j < i; j++)
      b[i] += a[j][i-j] * x[j];
```

```
    mm = i + m; if (mm > n) mm = n;
    for (j = i; j <= mm; j++)
      b[i] += a[i][j-i] * x[j];
  }
}

void disp(int n, int m, matrix a, vector x, char *s1, char *s2)
{
  int i, j;
  printf(s1);
  for (i = 1; i <= n; i++) {
    for (j = 0; j <= m; j++)
      printf(FORMAT, a[i][j]);
    printf("\n");
  }
  printf(s2);
  for (i = 1; i <= n; i++)
    printf(FORMAT, x[i]);
  printf("\n");

}

int main()
{
  int i, j, k, n, mm, m;
  matrix a, a0;
  vector x, b, b0;
  n = 8;
  m = 3;
  a  = new_matrix(n + 1, m + 1);
  b  = new_vector(n + 1);
  x = new_vector(n + 1);

  testmatrix(n, m, a, x, b);
  disp(n, m, a, b, "A=\n", "b=\n");

  /* LU 分解して解く */
  lu4(a, n, m);
  solve4(a, b, n, m);
  disp(n, m, a, b, "LU=\n", "x=\n");
}

void lu4(matrix a, int n, int m)
{
  int i, j, k, mm;
  double q;
  for (i = 1; i < n; i++) {
    mm = i + m; if (mm > n) mm = n;
    for (j = i + 1; j <= mm; j++) {
      /* 本来 q = a_{ji} / a_{ii} であるが a_{ji} の代りに a_{ij} を読む */
      q = a[i][j-i] / a[i][0];
      /* for (k = i + 1; k <= mm; k++) だったけれど対角線の下は何もしない */
      for (k = j; k <= mm; k++)
        a[j][k-j] -= q * a[i][k-i];
    }
  }
}

void solve4(matrix a, vector b, int n, int m)
{
  int i, j, mm;
  for (i = 1; i < n; i++) {
```

```
    mm = i + m; if (mm > n) mm = n;
    for (j = i + 1; j <= mm; j++) {
      /* b[j] -= L[j][i] * b[i]; としたい */
      b[j] -= (a[i][j-i] / a[i][0]) * b[i];
    }
  }
  b[n] /= a[n][0];
  for (i = n - 1; i >= 1; i--) {
    mm = i + m; if (mm > n) mm = n;
    for (j = i + 1; j <= mm; j++)
      b[i] -= a[i][j-i] * b[j];
    b[i] /= a[i][0];
  }
}
```

```
oyabun% ccmg bandtest5.c
oyabun% ./bandtest5
A=
 2.0000  0.5139  0.1757  0.3086
 3.0000  0.5345  0.9476  0.1717
 4.0000  0.7022  0.2264  0.4948
 5.0000  0.1247  0.0839  0.3896
 6.0000  0.2772  0.3681  0.9835
 7.0000  0.5354  0.7657  0.0000
 8.0000  0.6465  0.0000  0.0000
 9.0000  0.0000  0.0000  0.0000
b=
 4.7895  12.7666  20.1544  28.1649  43.6289  55.0792  67.7829  86.0367
LU=
 2.0000  0.5139  0.1757  0.3086
 2.8680  0.4894  0.8683  0.1717
 3.9011  0.5269  0.1971  0.4948
 4.6183  0.0461  0.0171  0.3896
 5.9793  0.2521  0.3642  0.9835
 6.9266  0.5186  0.7242  0.0000
 7.9061  0.5324  0.0000  0.0000
 8.7267  0.0000  0.0000  0.0000
x=
 1.0000  2.0000  3.0000  4.0000  5.0000  6.0000  7.0000  8.0000
oyabun%
```

### 5.8.6 帯行列用プログラムを利用する意義について

正方形領域における Poisson 方程式の Dirichlet 境界値問題を差分法で離散化した時に現れる連立 1 次方程式を解く場合を例に取って考えよう。

各辺を $N$ 等分することで格子を作ると、未知数の個数は約 $n := N^2$ 個となり、係数行列は $n$ 次正値対称行列で、半バンド幅 $m$ は $N$ 程度である (正確にどうなるかは方程式の立て方による)。

何の工夫もしない Gauss の消去法を用いて解く場合、四則演算の回数は $n^3 = N^6$ の程度となる [5]。$N = 100 = 10^2$ の場合に、未知数の個数は $n = N^2 = 10^4$, 行列の要素数は $n^2 = N^4 = (10^2)^4 = 10^8 = 100\mathrm{M}$. パソコン、ワークステーション上の C 言語処理系で実数を double で表現すると、1 要素 8B 必要なので、800MB のメモリーが必要になる。一方、四

[5] 乗除算が $n^3/3$ 回、加減算が $n^3/3$ 回程度。

則演算の回数は約 $N^6 = (10^2)^6 = 10^{12} = 1\mathrm{T} = 1000\mathrm{G}$ であるので、1GFLOPS のコンピューターで 1000 秒、10MFLOPS のコンピューターで $1000 \times 100$ 秒 $\fallingdotseq 28$ 時間かかる。$N$ を 2 倍にすると、メモリーは $2^4 = 16$ 倍、計算時間は $2^6 = 64$ 倍必要になる。

一方、行列が対称帯行列であることを利用して解く場合、四則演算の回数は $nm^2$ の程度となる。対角線の上側の帯の部分にある要素数は $nm = N^3$ で、演算回数は $nm^2 = N^4$ 程度である。上と同様にパソコン、ワークステーションで計算する場合、必要となるメモリーは $8 \times nm\mathrm{B} = 8N^3\mathrm{B} = 8(10^2)^3\mathrm{B} = 8 \times 10^6\mathrm{B} = 8\mathrm{MB}$. 演算回数は $nm^2 = N^4 = (10^2)^4 = 10^8 = 100\mathrm{M}$ なので、10MFLOPS のコンピューターで 10 秒程度。$N$ を 2 倍にすると、メモリーは $2^3 = 8$ 倍、計算時間は $2^4 = 16$ 倍必要になる。

## 5.9 実際の数値シミュレーションでは

後で述べるように、大規模数値シミュレーションに現われる連立 1 次方程式の解法には、反復法が採用されることが多くなって来たため、Gauss の消去法が利用されることは段々と少なくなりつつある (と思っているのだけど本当かな？)。

Gauss の消去法系のアルゴリズムを採用するのが適当である場合は、LAPACK などの線形計算ライブラリィの利用を考えるのが良い。「可能な限りプロの書いたプログラムを使え」が基本方針である。

# 第6章　Gauss の消去法と LU 分解 — がらくた箱 —

(この章は前章に含めてまとめるつもりで、まだ出来ていない文章を置いてある。)

## 6.1　Gauss の消去法

### 6.1.1　計算手順の記述

この小節ではピボット (以下の記号で $a_{kk}^{(k)}$) が 0 にならないと仮定する。
$A = (a_{ij}) \in M(n; \mathbf{R})$, $b \in \mathbf{R}^n$ とするとき、

$$Ax = b \tag{6.1}$$

を解く。

### 6.1.2　前進消去

(6.1) を成分を使って書くと

$$\begin{array}{ccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\
a_{21}^{(1)}x_1 & + & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\
a_{31}^{(1)}x_1 & + & a_{32}^{(1)}x_2 & + & \cdots & + & a_{3n}^{(1)}x_n & = & b_3^{(1)} \\
\cdots & & \cdots & & \cdots & & \cdots & & \vdots \\
a_{n1}^{(1)}x_1 & + & a_{n2}^{(1)}x_2 & + & \cdots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)}
\end{array}$$

となる。ただし第一段であることを強調する意味で右肩に $^{(1)}$ をつけた。
第 1 行で第 $2, \cdots, n$ 行を掃き出すと

$$\begin{array}{ccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\
 & & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\
 & & a_{32}^{(2)}x_2 & + & \cdots & + & a_{3n}^{(2)}x_n & = & b_3^{(2)} \\
 & & \cdots & & \cdots & & \cdots & & \vdots \\
 & & a_{n2}^{(2)}x_2 & + & \cdots & + & a_{nn}^{(2)}x_n & = & b_n^{(2)}
\end{array}$$

を得る。具体的には以下のように計算した。

$$
\begin{aligned}
&q_{21}=a_{21}^{(1)}/a_{11}^{(1)},\quad q_{31}=a_{31}^{(1)}/a_{11}^{(1)},\quad \cdots,\quad q_{n1}=a_{n1}^{(1)}/a_{11}^{(1)},\\
&a_{22}^{(2)}=a_{22}^{(1)}-q_{21}a_{12}^{(1)},\quad a_{23}^{(2)}=a_{23}^{(1)}-q_{21}a_{13}^{(1)},\quad \cdots,\quad a_{2n}^{(2)}=a_{2n}^{(1)}-q_{21}a_{1n}^{(1)},\quad b_2^{(2)}=b_2^{(1)},\\
&a_{32}^{(2)}=a_{32}^{(1)}-q_{31}a_{12}^{(1)},\quad a_{33}^{(2)}=a_{33}^{(1)}-q_{31}a_{13}^{(1)},\quad \cdots,\quad a_{3n}^{(2)}=a_{3n}^{(1)}-q_{31}a_{1n}^{(1)},\quad b_3^{(2)}=b_3^{(1)},
\end{aligned}
$$

$$
a_{i2}^{(2)}=a_{i2}^{(1)}-q_{i1}a_{12}^{(1)},\quad a_{i3}^{(2)}=a_{i3}^{(1)}-q_{i1}a_{13}^{(1)},\quad \cdots,\quad a_{in}^{(2)}=a_{in}^{(1)}-q_{i1}a_{1n}^{(1)},\quad b_i^{(2)}=b_i^{(1)},
$$

$$
a_{n2}^{(2)}=a_{n2}^{(1)}-q_{n1}a_{12}^{(1)},\quad a_{n3}^{(2)}=a_{n3}^{(1)}-q_{n1}a_{13}^{(1)},\quad \cdots,\quad a_{nn}^{(2)}=a_{nn}^{(1)}-q_{n1}a_{1n}^{(1)},\quad b_n^{(2)}=b_n^{(1)}.
$$

一般に第 $k$ 段目では

$$
\begin{array}{ccccccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1k}^{(1)} & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)}\\
 & & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2k}^{(2)} & + & \cdots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)}\\
 & & & & \ddots & & & & & & & & \\
 & & & & & & a_{kk}^{(k)}a_k & + & \cdots & + & a_{3n}^{(k)}x_n & = & b_3^{(k)}\\
 & & & & & & \vdots & & \vdots & & \vdots & & \vdots\\
 & & & & & & a_{nk}^{(k)}x_k & + & \cdots & + & a_{nn}^{(k)}x_n & = & b_n^{(k)}
\end{array}
$$

から、第 $k$ 行で第 $k+1,\cdots,n$ 行を掃き出すことで、

$$
\begin{array}{ccccccccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1k}^{(1)}x_k & + & a_{1,k+1}^{(1)}x_{k+1} & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)}\\
 & & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2k}^{(2)}x_k & + & a_{2,k+1}^{(2)}x_{k+1} & + & \cdots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)}\\
 & & & & \ddots & & & & & & & & & & \\
 & & & & & & a_{kk}^{(k)}x_k & + & a_{k,k+1}^{(k)}x_{k+1} & + & \cdots & + & a_{3n}^{(k)}x_n & = & b_k^{(k)}\\
 & & & & & & & & a_{k+1,k+1}^{(k+1)}x_{k+1} & + & \cdots & + & a_{3n}^{(k+1)}x_n & = & b_{k+1}^{(k+1)}\\
 & & & & & & & & \vdots & & \vdots & & \vdots & & \vdots\\
 & & & & & & & & a_{k+1,k+1}^{(k+1)}x_{k+1} & + & \cdots & + & a_{3n}^{(k+1)}x_n & = & b_{k+1}^{(k+1)}
\end{array}
$$

を得る。具体的には次のように計算する。

$$
\begin{aligned}
q_{ik} &= a_{ik}^{(k)}/a_{kk}^{(k)} \quad (i=k+1,k+2,\cdots,n),\\
a_{ij}^{(k+1)} &= a_{ij}^{(k)}-q_{ik}a_{kj}^{(k)} \quad (i=k+1,k+2,\cdots,n;\ j=k+1,k+2,\cdots,n),\\
b_i^{(k+1)} &= b_i^{(k)}-q_{ik}b_k^{(k)} \quad (i=k+1,k+2,\cdots,n).
\end{aligned}
$$

第 $n-1$ 段の計算が終わると次のような形になっている。

$$
\left\{
\begin{array}{ccccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & a_{1,n-1}^{(1)}x_{n-1} & + & a_{1,n}^{(1)}x_n & = & b_1^{(1)},\\
 & & a_{22}^{(2)}x_2 & + & \cdots & a_{2,n-1}^{(2)}x_{n-1} & + & a_{2,n}^{(2)}x_n & = & b_2^{(2)},\\
 & & & & \ddots & \vdots & \vdots & \vdots & & \\
 & & & & & a_{n-1,n-1}^{(n-1)}x_{n-1} & + & a_{n-1,n}^{(n-1)}x_n & = & b_{n-1}^{(n-1)},\\
 & & & & & & & a_{n,n}^{(n)}x_n & = & b_n^{(n)}
\end{array}
\right. \tag{6.2}
$$

**後退代入**

(6.2) は次のようにして解ける:

$$\begin{aligned} x_n &= b_n^{(n)}/a_{nn}^{(n)}, \\ x_{n-1} &= \left(b_{n-1}^{(n-1)} - a_{n-1,n}^{(n-1)} x_n\right)/a_{n-1,n-1}^{(n-1)}, \\ &\vdots \qquad\qquad \vdots \\ x_k &= \left(b_k^{(k)} - \sum_{i=k+1}^{n} a_{ki}^{(k)} x_i\right)/a_{kk}^{(k)}, \\ &\vdots \qquad\qquad \vdots \\ x_1 &= \left(b_1^{(1)} - \sum_{i=2}^{n} a_{1i}^{(1)} x_i\right)/a_{11}^{(1)}. \end{aligned}$$

### 6.1.3 前進消去過程の行列表示

よく行うように、方程式 $Ax = b$ を $A$ と $b$ を並べた拡大行列

$$A^{(1)} = (A\ b) = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & b_n^{(1)} \end{pmatrix}$$

で表現する。ここで

$$a_{ij}^{(1)} = a_{ij} \quad (i,j = 1,2,\cdots,n), \quad b_1^{(1)} = b_i \quad (i = 1,2,\cdots,n)$$

である。

第 $k$ 行で第 $k+1, \cdots, n$ 行を消去するという第 $k$ 段は

$$A^{(k)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1,n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2,n}^{(2)} & b_2^{(2)} \\ & & \ddots & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{k,n}^{(k)} & b_k^{(k)} \\ & & & a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} & b_{k+1}^{(k)} \\ & \mathbf{0} & & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{n,k}^{(k)} & a_{n,k+1}^{(k)} & \cdots & a_{n,n}^{(k)} & b_n^{(k)} \end{pmatrix}$$

を変形して

$$A^{(k+1)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1k}^{(1)} & a_{1,k+1}^{(1)} & \cdots & a_{1,n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2k}^{(2)} & a_{2,k+1}^{(2)} & \cdots & a_{2,n}^{(2)} & b_2^{(2)} \\ & & \ddots & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{kk}^{(k)} & a_{k,k+1}^{(k)} & \cdots & a_{k,n}^{(k)} & b_k^{(k)} \\ & & & & a_{k+1,k+1}^{(k+1)} & \cdots & a_{k+1,n}^{(k+1)} & b_{k+1}^{(k+1)} \\ & \mathbf{0} & & & \vdots & & \vdots & \vdots \\ & & & & a_{n,k+1}^{(k+1)} & \cdots & a_{n,n}^{(k+1)} & b_n^{(k+1)} \end{pmatrix}$$

とするわけだが、これは

$$M_k = \begin{pmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -q_{k+1,k} & 1 & & \\ & & & -q_{k+2,k} & 0 & \ddots & \\ & & & \vdots & \vdots & \ddots & \ddots \\ & & & -q_{n,k} & 0 & \cdots & 0 & 1 \end{pmatrix}$$

を左から掛けていることに相当する:

$$A^{(k+1)} = M_k A^{(k)}.$$

そこで

$$\mathcal{L} \stackrel{\text{def.}}{=} M_{n-1} M_{n-2} \cdots M_2 M_1 A^{(1)}$$

とおくと

$$\mathcal{L}A^{(1)} = A^{(n)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} & b_3^{(3)} \\ & & & \ddots & \vdots & \vdots \\ & & & & a_{nn}^{(n)} & b_n^{(n)} \end{pmatrix}. \tag{6.3}$$

これを拡大しない行列で表現すると

$$\mathcal{L}A = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ & & & \ddots & \vdots \\ & & & & a_{nn}^{(n)} \end{pmatrix} \stackrel{\text{def.}}{=} U, \quad \mathcal{L} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{pmatrix}.$$

なお

$$(M_k)^{-1} = \begin{pmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & q_{k+1,k} & 1 & & \\ & & & q_{k+2,k} & 0 & \ddots & \\ & & & \vdots & \vdots & \ddots & \ddots \\ & & & q_{n,k} & 0 & \cdots & 0 & 1 \end{pmatrix}$$

である (このことは掛け算して直接確かめることもできる。また後述の注意も参照せよ。)。さらに

$$L \stackrel{\text{def.}}{=} \mathcal{L}^{-1} = (M_1)^{-1} (M_2)^{-1} \cdots (M_{n-1})^{-1}$$

とおくと、

$$L = \begin{pmatrix} 1 & & & & & \\ q_{21} & 1 & & & 0 & \\ q_{31} & q_{32} & 1 & & & \\ \vdots & \vdots & q_{43} & \ddots & & \\ \vdots & \vdots & \vdots & \ddots & 1 & \\ q_{n1} & q_{n2} & q_{n3} & \cdots & q_{n,n-1} & 1 \end{pmatrix}.$$

そして $\mathcal{L}A = U$ より

$$A = LU.$$

これが後で定義を述べる $A$ の LU 分解である。

なお、$L$ の対角線分がすべて 1 であることから、

$$\det A = \det L \cdot \det U = \prod_{k=1}^{n} a_{kk}^{(k)}.$$

つまり LU 分解は行列式の計算にも利用できる (しばしば最も有利なアルゴリズムとなる)。このことはよく知られているが、後のために一般化して次の命題を準備しておこう。

**補題 6.1.1 (主座行列式と枢軸の関係)** 正則行列 $A \in GL(n;\mathbf{R})$ に対して、枢軸選びがなく Gauss の消去法を第 $r$ 段まで行うことができたとする。枢軸を順に $a_{kk}^{(k)}$ $(k = 1, 2, \cdots, r)$ とするとき、$A$ の $k$ 次主座行列式を $\delta_k$ とおくと (ただし $\delta_0 = 1$ とする)、

$$a_{kk}^{(k)} = \frac{\delta_k}{\delta_{k-1}}$$

がなりたつ。

**証明** 消去演算は基本行列を左から掛けることであるが、その場合の基本行列の行列式は 1 である。ゆえに $k$ 段まで消去したとき、$k$ 次主座行列式は、$U$ の対角成分の最初の $k$ 個の積になる:

$$\delta_k = a_{11}^{(1)} a_{22}^{(2)} \cdots a_{kk}^{(k)}.$$

これから主張を得る。 ■

## 6.2 LU 分解の存在

**定理 6.2.1 (LU 分解が存在するための条件)** $n$ 次正則行列 $A = (a_{ij})$ が LU 分解可能であるための必用十分条件は、$A$ の主座小行列式がすべて $\neq 0$ であること:

$$\delta_k \overset{\text{def.}}{=} \det \begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{pmatrix} \neq 0 \quad (k = 1, 2, \cdots, n).$$

さらにこのとき、Crout タイプの LU 分解、Dolittle タイプの LU 分解、LDU 分解のいずれも一意的に可能である。

**証明** 補題 6.1.1 から明らかであろう。なお、山本・北川 [10] を見よ。また津田 [9] にもある (これは Rice [6] によるらしい)。■

## 6.3 対称行列の LU 分解

(現時点でこの節の記述は気に食わない。書き直せ。森 [5] が参考になるかも。)

### 6.3.1 対称行列の修正 Cholesky 分解

**定理 6.3.1 (修正 Cholesky 分解)** $N$ 次対称行列 $A = (a_{ij})$ の主座行列式 $\delta_k$ ($k = 1, 2, \cdots, N$) がすべて 0 でないならば、一意的に

$$A = LDL^T \quad (L \text{ は単位下三角行列, } D \text{ は対角行列})$$

と分解できる。

**証明** 行列を

$$L = \begin{pmatrix} 1 & & & & \\ \ell_{21} & 1 & & \mathbf{0} & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{N1} & \ell_{N2} & \cdots & \ell_{N,N-1} & 1 \end{pmatrix}, \quad D = \begin{pmatrix} d_1 & & & \mathbf{0} \\ & d_2 & & \\ & & \ddots & \\ \mathbf{0} & & & d_N \end{pmatrix}$$

とおくと、$LDL^T$ の第 $(i,j)$ 成分は

$$(\ell_{i,1}\ \ell_{i,2}\ \cdots\ \ell_{i,i-1}\ 1\ 0\ \cdots\ 0) \begin{pmatrix} d_1\ell_{j1} \\ d_2\ell_{j2} \\ \vdots \\ d_{j-1}\ell_{j,j-1} \\ d_j \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \sum_{k=1}^{\min\{i,j\}} d_k\ell_{ik}\ell_{jk}.$$

これが $A$ に等しいとするのだが、対称性から $i \le j$ についてだけ条件を書けばよい。$\ell_{ii} = 1$ に注意すると

$$a_{ij} = \sum_{k=1}^{i} d_k\ell_{ik}\ell_{jk} = \begin{cases} d_i + \displaystyle\sum_{k=1}^{i-1} d_k\ell_{ik}^2 & (j = i) \\ \displaystyle\sum_{k=1}^{i-1} d_k\ell_{ik}\ell_{jk} + d_i\ell_{ji} & (j = i+1, i+2, \cdots, N) \end{cases}$$

を得る。これから $i = 1, 2, \cdots, N$ の順に

$$\begin{aligned} d_i &= a_{ii} - \sum_{k=1}^{i-1} d_k\ell_{ik}^2, \\ \ell_{ji} &= \frac{1}{d_i}\left(a_{ij} - \sum_{k=1}^{i-1} d_k\ell_{ik}\ell_{jk}\right) \quad (j = i+1, \cdots, N) \end{aligned}$$

で $\{d_i\}, \{\ell_{ji}\}_{i \le j}$ が求まる。実際 $d_i$ は Gauss の消去法を $i$ 段まで進めた時の枢軸 $d_{ii}^{(i)}$ に等しいので $\neq 0$ であるから。 ■

最初のうち

$$
\begin{aligned}
d_1 &= a_{11} = \delta_1 \neq 0, \quad \ell_{j1} = a_{j1}/d_1 \quad (j = 2, \cdots, N), \\
d_2 &= a_{22} - d_1\ell_{21}^2 = \delta_2/\delta_1 \neq 0, \quad \ell_{j2} = (a_{2j} - d_1\ell_{21}\ell_{j1})/d_2 \quad (j = 3, \cdots, N), \\
d_3 &= a_{33} - d_1\ell_{31}^2 - d_2\ell_{32}^2 = \delta_3/\delta_2 \neq 0, \quad \ell_{j3} = (a_{3j} - d_1\ell_{31}\ell_{j1} - d_2\ell_{32}\ell_{j2})/d_3 \quad (j = 4, \cdots, N),
\end{aligned}
$$

これを $A$ の**修正 Cholesky 分解**とよぶ。
対称性を利用してうまく計算すると、通常の LU 分解の約半分の計算量で分解できる。

### 6.3.2 正値対称行列に対する Cholesky 分解

$A$ が正値対称行列であるとき、すべての主座行列式は正であるから、定理 6.2.1 から $A$ は LDU 分解可能である:

$$A = LDU.$$

また既に注意したように $L = U^T$ であるから

$$A = U^T DU.$$

さらに

$$D = \begin{pmatrix} a_{11}^{(1)} & & & \\ & a_{22}^{(2)} & & \\ & & \ddots & \\ & & & a_{NN}^{(N)} \end{pmatrix}$$

において

$$a_{kk}^{(k)} > 0.$$

ゆえに

$$G = \begin{pmatrix} \sqrt{a_{11}^{(1)}} & & & \\ & \sqrt{a_{22}^{(2)}} & & \\ & & \ddots & \\ & & & \sqrt{a_{NN}^{(N)}} \end{pmatrix}$$

とおくと、$G^2 = D$, $G^T = G$ であるから、

$$A = U^T DU = U^T G^2 U = U^T G^T GU = (GU)^T (GU) = S^T S.$$

ただし

$$S \overset{\text{def.}}{=} LG$$

とおいた。
このような $A = S^T S$ という分解を **Cholesky 分解**という。
Cholesky 分解は、対角成分の符号を除いて一意的に定まる (一意的でないのは、$D$ の平方根として上の $G$ 以外のものが取れることから明らか)。
この $S$ は以下のように直接求めることができる。$S^T S = A$ を成分で表わすと

$$s_{1i}s_{1j} + s_{2i}s_{2j} + \cdots + s_{ii}s_{ij} = a_{ij}$$

これから

$$s_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} s_{kj}^2}, \quad s_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} s_{ki}s_{kj}\right) / s_{ii}.$$

## 6.4 要点

- 正則行列 $A$ の LU 分解があるとき、$A$ を係数行列とする連立 1 次方程式が少ない計算量で解ける。特に $A$ が疎行列の場合に顕著で「逆行列は必要ない (有害とさえ言える)」。
- ピボット選択なしの Gauss の消去法 (ピボットを割り算して 1 にしないもの) は Crout タイプの LU 分解をしていると解釈できる。
- 任意の正則行列 $A$ が LU 分解できるとは限らない。しかし、適当な置換行列 $P$ に対して $PA$ は必ず LU 分解できる。
- ピボット選択ありの Gauss の消去法 (ピボットを割り算して 1 にしないもの) は、適当な置換行列 $P$ に対して、$PA$ を Crout タイプの LU 分解していることに相当する。
- 正則行列の LU 分解は、できたとしても、一意であるとは限らない。しかし Crout タイプ、Dolittle タイプの LU 分解は一意である。

## 6.5 つぶやき

しかし、読みづらい本が多い… きちんとした定義、命題、証明が書いていないことが多くて閉口する。洋書ならあるのかもしれないが、和書では最近厚い本が出されなくなった関係か、ちょっと物足りない本が多い (大きな声では言えないが)。杉原・室田コンビが本を書くはずと読んだのだけど、待てど暮せど出て来ない…

今後、補充したいこととして、

- 枢軸の選択の話
- LINPACK, LAPACK の話。一体何か？入手先、参考文献。
  (ちょっと書いた文書があるのでマージしたい)
- 条件数。定義と情報へのポインター。
  (別の文書に書いてあるので、きちんとリンクをはろう)
- 実験のためのテクニック。特に `rnd()` など。
  (疑似乱数についてはちょいと書いたものがあった。どこに行ったか？)

# 関連図書


[1] 菊地文雄, 山本昌宏. 微分方程式と計算機演習. 山海堂, 1991.

[2] 伊理正夫(いりまさお). 一般線形代数. 岩波書店, 2003. 伊理正夫, 線形代数 I, II, 岩波講座応用数学, 岩波書店 (1993, 1994) の単行本化.

[3] 伊理正夫, 藤野和建(よりたけ). 数値計算の常識. 共立出版, 1985.

[4] 菊地文雄. 有限要素法概説. サイエンス社, 1980. 新訂版 1999.

[5] 森正武. 数値解析法. 朝倉現代物理学講座 7. 朝倉書店, 1984.

[6] J. R. Rice. *Numerical Methods, Software, and Analysis.* McGraw-Hill, 1983.

[7] 齋藤正彦. 線型代数演習. 東京大学出版会, 1985.

[8] G. D. Smith. *Numerical solution of partial differential equations third edition.* Clarendon Press Oxford, 1986. 第一版の邦訳が G. D. スミス著, 藤川洋一郎訳, コンピュータによる偏微分方程式の解法 新訂版, サイエンス社 (1996) である.

[9] 津田孝夫. 数値処理プログラミング. 岩波書店, 1988.

[10] 山本哲朗, 北川高嗣(たかし). 数値解析演習. サイエンス社, 1991.