

ガウス・ジョルダン法のプログラム方法

山本昌志*

2003年11月10日

1 準備

1.1 関数を使おう

ガウス・ジョルダン法は、連立1次方程式の解、あるいは逆行列を求める汎用的な手法で、あらゆるこの種の問題に適用できます。ということは、良いプログラムを書けば、再利用することができることを意味します。このように、再利用できそうなルーチンは、メイン関数には書かないで、それ専用の関数を作るほうが良いです¹。そうすると、ほかの場所で連立1次方程式を解きたい場合は、その関数をコールするだけですみます。また、ほかのプログラムを書く場合でも、それをコピーして使うことができます。そういうわけで、メイン関数ではなく、ガウス・ジョルダン法の専用の関数を作ってプログラムを書きましょう。C言語は関数の独立性が高いので、その辺は比較的簡単にできます。

ガウス・ジョルダン法の専用の関数を作る場合、どうするか?。まず、その入出力と機能を考えます。入力、係数行列の A と非同次項の b が必要です。そして、出力は逆行列の A^{-1} と解のベクトル x となります。要するに、図1のような機能のC言語の関数が欲しいわけです。

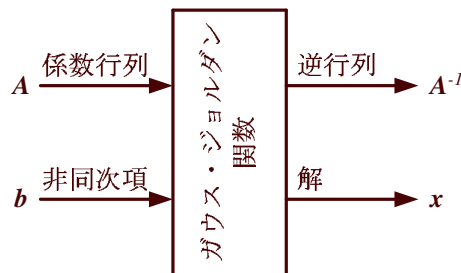


図 1: ガウスジョルダン法を計算する C 言語の関数のブラックボックス

この関数ができると、問題を解く時に必要な係数行列と非同次項を入力さえすれば、逆行列と解を計算してくれます。この関数、実際はガウス・ジョルダン法の手続きを C 言語でいかにして実現するかは、次節以降に述べます。

*国立秋田工業高等専門学校 電気工学科

¹もっと良いのは、ライブラリーにしてしまうことです。これは、学習の範囲外なので興味のある人は調べてみてください

1.2 関数へのデータの受け渡し

1.2.1 値の渡し方を思い出そう

機能は決まったので、つぎに考えることは、関数へのデータの受け渡しです。C言語のデータの渡し方は、少し複雑なので、復習をしましょう。

いかなるプログラム言語でも、C言語の関数 (main ではない) に対応するものが有ります。FORTRANでは、サブルーチンと呼ばれるものです。同じような処理がある場合、1つの独立した処理のブロックとしてまとめ、どこからでもコールすることができるようにすると便利です。例えば、 $\sin(x)$ などもこれにあたります。 $\sin x$ の計算が必要な都度、その処理を書いていたのではたまりませんので、独立した関数としてその処理が書かれているわけです。これは、ライブラリーとなっているので、その処理内容は通常はわかりません。

この関数にデータを与える変数のことを引数と言います。先ほどの例で言うと、C言語の $\sin(x)$ の x が引数です。プログラムの中では、 \sin という名前がついている処理に x が与えられ、それを処理することになります。

引数には、2種類 (実引数と仮引数) が有ります。それについて、次のプログラムで説明します。この場合、main 関数でコールするときの文、 $\text{add}(x,y)$ の x と y を実引数と呼びます。そして、その処理を書いている関数 $\text{add}(\text{double } xin, \text{double } yin)$ の xin と yin を仮引数と言います。

```
#include <stdio.h>
double add(double xin, double yin);

/* ===== main 関数 =====*/
main(){
    double x, y, wa;

    いろいろな処理

    wa=add(x,y);

    いろいろな処理
}

/* ===== 足し算の関数 =====*/
double add(double xin, double yin){
    double zout;

    zout = x+y;
    return(zout);
}
```

その実引数から仮引数に値を送る方法は、2通りあります。以前学習した通りで、

値渡し (Call by value) コールした (呼び出した) 関数と処理する関数では、別々のメモリー領域を用意します。そして、コールしたときに呼び出し側のメモリー (実引数) の値を処理する関数のメモリー (仮

引数)にコピーします。従って、処理する関数が仮引数の値を変えても、呼び出し側の実引数の値が変わることはありません。

アドレス渡し (Call by reference) 処理する関数では値を格納するメモリー領域を用意しません。実引数は、コールした関数の実引数の値が格納されているメモリーのアドレスとなります。そのアドレスが仮引数に渡されます。従って、処理する関数はそのアドレスを格納するメモリー領域を用意するだけです。処理する関数は、実引数のアドレスに格納されている値を処理することになります。この場合は、呼び出された関数が仮引数の値を変えると、呼び出し側の実引数の値も変わります。

です。

C 言語の場合、通常の変数 (配列でない) の場合、値渡しです。これは良くできた仕様と思います。処理する関数が呼び出し側のデータを変えることが無いので、プログラミングの時、余計な気を使わないですみます。関数の独立性が高いといわれる所以です。実際の例では、先の `add` という関数は、`main` 関数から呼び出されており、`main` の実引数の値 (`x,y`) の値が、`add` の仮引数 (`xin,yin`) にコピーされます。そこでの処理の結果は、戻り値 (返却値) `zout` に入れられて、元の関数に戻ります。元の関数の `wa` に、`zout` の値がコピーされます。

一方、配列を処理する関数に渡す場合は、アドレス渡しになります。一般に配列のデータは、変数よりもかなり大きく、それをいちいちコピーしていたら不経済ということらしいです。

ということで、今回の場合、配列を渡すためアドレス渡しになります。処理する関数でその配列の値を変えると、コールした関数のその値も変わります。しかし、これは便利なこともあります。いちいち戻り値を与える必要が無く、気軽に呼び出した関数に結果を返せます (FORTRAN と同じ)。

従って、図 1 のような入出力の関数を実現するための引数の書き方は、

```
#include <stdio.h>
void gaussjordan(double a[][100], double b[100],
                 double inv_a[][100], double x[100]);

/* ===== main 関数 =====*/
void main(){
    double a[100][100], b[100];
    double inv_a[100][100], x[100];

    いろいろな処理

    gaussjordan(a,b,inv_a,x);

    いろいろな処理
}

/* ===== ガウスジョルダン法の関数 =====*/
void gaussjordan(double a[][100], double b[100],
                 double inv_a[][100], double x[100]){

    いろいろな処理

    inv_a[i][j] = いろいろな計算
```

```
b[i] = これも計算  
  
いろいろな処理  
  
}
```

となります。処理する関数の方は、一番最初のサイズを除いて書く必要があります。これは、例えば `z[100][200][300]` の大きさの配列の `z[23][73][36]` というデータにアクセスする場合を考えましょう。このデータがあるアドレスは、`[Zのアドレス+配列1個のデータサイズ×(36+73×300+23×200×300)]` になります。したがって、最初の配列のサイズを除いてアドレス計算に必要となりますので、処理する関数側で明示する必要があります。

実際のプログラムでは、もう少し効率よく配列を使いますが、大筋はこの通りです。これで、関数に値を与える復習は終わり。

1.2.2 行列が特異な場合の警告

もし係数行列 A が特異だと、解 x は一意に決まりません。その場合、ガウス・ジョルダンを計算する関数は、呼び出し側へ警告を出す方が望ましいです。それは、呼び出し側へゼロを返すことで実現できます。それは、

- 行列が特異な場合、
`return(0);`
- 行列が正則な場合、
`return(1);`

と書けば良いのです。

1.2.3 行列のサイズはどうするの

本当にこれだけでよいのでしょうか？。先の例で配列を値を、処理するプログラムに知らせることができたのですが、これで全て計算の準備が整ったわけではありません。行列やベクトルのサイズを関数に知らせる必要があります。これは、大きな配列を用意しておいて、その一部分に係数や非同次項の値を入れるため、処理するときに行列やベクトルの大きさが必要になります。このような理由から、行列やベクトルのサイズを渡す必要が生じます。そこを考慮すると、ガウス・ジョルダン法の関数のプロトタイプ宣言は、次のようになります。

```
int gaussjordan(int n, double a[][100], double b[100],  
                double inv_a[][100], double x[100]);
```

2 実際のプログラム(手取り足取り)

ここでは、実際のプログラムを作成するときの考え方をしめします。最初に、何にも考えていないガウス・ジョルダン法から出発し、少しずつ機能を追加して、最終的にパッケージとして完成した関数を作成します。以下の順序でプログラムをブラッシュアップしていきます。

1. \mathbf{A} を対角化します。そして、 \mathbf{x} を求めます。
2. 行を交換するだけのピボット選択 (部分選択) の機能を追加します。
3. 逆行列を計算するルーチンを追加します。
4. 不要な配列を排除して、メモリー効率を上げる改造をします。

2.1 素朴なガウス・ジョルダン法 (行列の対角化のみ)

まずは、行列の対角化のみのプログラムを作成しましょう。これにより、 \mathbf{A} は単位行列に変換され、非同次項 \mathbf{b} は解ベクトル \mathbf{x} に変換されます。

この節のプログラムは、ピボット選択がないため、実用上問題を含んでいます。しかし、ガウス・ジョルダン法のプログラムの学習には良いでしょう。皆さんが、学習ではなく実際に使うプログラムを組むときはピボット選択は必要不可欠と考えてください。

さらに、このプログラムは行列が特異な場合でも、計算を続行しようとします。その場合、ゼロで割ることが生じますので、実行時エラーが発生します。

2.1.1 最外殻のループ

$N \times N$ の行列 \mathbf{A} を対角化することを考えます。この場合、 $a_{11}, a_{22}, a_{33}, \dots, a_{NN}$ と同じ手順で対角化を進めることになります。ということは、 N 回のループをまわすことになります。プログラムでは、以下のループ構造が一番外側で回ることになります。ループの回数が分かっている場合、for 文をつかいます。

```
for(ipv=1 ; ipv <= n ; ipv++){  
    対角化の処理  
}
```

ここで、 ipv は対角化する要素 $a_{ipv\ ipv}$ の添え字を表します。n は行列のサイズです。

2.1.2 対角成分を 1 に (ピボット行の処理)

次の処理は、対角成分を $a_{ipv\ ipv} = 1$ にすることです。 $a_{ipv-1\ ipv-1}$ 間では対角化できており、次の成分を対角化するという事です。もし、 $ipv = 1$ ならば最初の対角成分を 1 にすることになります。最初である

うが、途中であろうが同じ処理になります。具体定期には、

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & * & * & * & * \\ 0 & 1 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \end{pmatrix} \Rightarrow \mathbf{A}' = \begin{pmatrix} 1 & 0 & * & * & * & * \\ 0 & 1 & * & * & * & * \\ 0 & 0 & 1 & *' & *' & *' \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \end{pmatrix}$$

と変形したのである。係数行列 \mathbf{A} を変形させれば、同じ操作により非同次項 \mathbf{b} も処理しなくてはなりません。

数学では、対角成分を 1 にするために、その行を対角成分で割ります。しかし、コンピューターのプログラムでは予め逆数を計算して、それを乗じます。コンピューターは除算よりも乗算の方が得意なので効率が良くなります。非同次項 \mathbf{b} の演算は 1 回ですが、係数行列 \mathbf{A} は列毎なので N 回の演算が必要になります。対角成分を 1 にする処理は、次のようになります。

```
inv_pivot = 1.0/a[ipv][ipv];

for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
}

b[ipv] *= inv_pivot;
```

- ipv 行の j 列、 $j = 1, 2, 3, \dots, n$ を処理するために、for 文を用いたループになっています。
- $a[ipv][j] \leftarrow inv_pivot * a[ipv][j]$ を、通常、C 言語では $a[ipv][j] *= inv_pivot$ と書きます。あるいは、 $a[ipv][j] = inv_pivot * a[ipv][j]$ と書いても良いです。前者の方が少しかっこいいです。

これでピボットのある行の処理は終わりです。

2.1.3 ピボットのある列を 0 に (ピボット行以外の行の処理)

次は、ピボットがある行以外の処理です。それは、ピボットがある列を全てゼロにすることです。要するに次のように、係数行列 \mathbf{A} を変形します。

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & a_{1\ ipv} & * & * & * \\ 0 & 1 & a_{2\ ipv} & * & * & * \\ 0 & 0 & 1 & * & * & * \\ 0 & 0 & a_{4\ ipv} & * & * & * \\ 0 & 0 & a_{5\ ipv} & * & * & * \\ 0 & 0 & a_{6\ ipv} & * & * & * \end{pmatrix} \Rightarrow \mathbf{A}' = \begin{pmatrix} 1 & 0 & 0 & *' & *' & *' \\ 0 & 1 & 0 & *' & *' & *' \\ 0 & 0 & 1 & * & * & * \\ 0 & 0 & 0 & *' & *' & *' \\ 0 & 0 & 0 & *' & *' & *' \\ 0 & 0 & 0 & *' & *' & *' \end{pmatrix}$$

このように係数行列 A を変形させます。もちろん、方程式の解 x が変わらないように、非同次項 b も同じ操作をします。

このように変形するのは簡単です。例えば、 i 行を処理する場合を考えます。 i 行を、ピボットのある ipv 行を $a_{i\,ipv}$ 倍もので引けば良いのです。

$$\begin{array}{rcl}
 i \text{ 行} & \Rightarrow & \begin{array}{cccccc} 1 & 0 & a_{i\,ipv} & * & * & * \end{array} & b_i \\
 ipv \text{ 行} & \Rightarrow & -a_{i\,ipv} \times \left(\begin{array}{cccccc} 0 & 0 & 1 & * & * & * \end{array} \right) & -a_{i\,ipv} \times b_{ipv} \\
 \hline
 \text{新 } i \text{ 行} & \Rightarrow & \begin{array}{cccccc} 1 & 0 & 0 & *' & *' & *' \end{array} & b_i - a_{i\,ipv} b_{ipv}
 \end{array}$$

この処理の実際のプログラムは次のようになります。これで、 $i = 1, 2, 3, \dots, n$ 行 $j = 1, 2, 3, \dots, n$ の全ての A の成分を処理します。

```

for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}

```

- 2つの for 文で i 行 j 列を処理します。
- ipv 行は対角成分を 1 にすることで処理が済んでいるので、この処理はしません。そこで、if 文を用いて $i \neq ipv$ の時、列の処理をするルーチンが実行されるようになっています。 $i = ipv$ の時は何もしないようになっています。

これで対角化の処理はおしまい。

2.1.4 素朴なガウス・ジョルダン法のソースプログラム

以上をまとめると、ピボット選択は無く逆行列も求めないのガウス・ジョルダン法が完成します。ここで、ひとつ気が付いてほしいのは解ベクトル x のための配列は不要ということです。非同次項の配列が解になっています。従って、この最も素朴なガウス・ジョルダン法のプログラムは次のようになります。

```

/* ===== ガウスジョルダン法の関数 ===== */
void gauss_jordan(int n, double a[][100], double b[]){

    int ipv, i, j;
    double inv_pivot, temp;

    for(ipv=1 ; ipv <= n ; ipv++){

        /* ---- 対角成分=1(ピボット行の処理) ---- */
        inv_pivot = 1.0/a[ipv][ipv];
    }
}

```

```

for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
}
b[ipv] *= inv_pivot;

/* ---- ピボット列=0(ピボット行以外の処理) ---- */
for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}
}
}

```

2.2 ピボット選択機能追加 (行交換)

先ほどの素朴なガウス・ジョルダン法は、爆弾を抱えた関数になっています。もし、対角成分にゼロが現れたら、ゼロで割ることになり処理が破綻します。そこで、ピボット選択が登場します。

ピボット選択、ここでは行の交換のみの部分選択を考えます。その処理は、

- ピボット列で、最大の値を探す。
- 最大の値のある行をピボット行と交換する。

の2つの部分から構成されます。この処理を加える場合、先のプログラムの対角成分を1にする処理のまゝに入れます。

2.2.1 最大値探索

行交換のみを行う部分選択の場合、ピボットはピボット行以下の最大値とします。これは、今まで処理した、対角成分が1になっている部分を崩さないためです。

$$\begin{array}{l}
 \text{行の交換不可} \\
 \text{行の交換可}
 \end{array}
 \left\{ \begin{array}{l}
 \left(\begin{array}{cccccc}
 1 & 0 & * & * & * & * \\
 0 & 1 & * & * & * & * \\
 0 & 0 & * & * & * & * \\
 0 & 0 & * & * & * & * \\
 0 & 0 & * & * & * & * \\
 0 & 0 & * & * & * & * \\
 0 & 0 & * & * & * & *
 \end{array} \right)
 \end{array} \right.$$

最大値は、*ipv* 行よりも下の行で、*ipv* 列の最大値は、以降に示すプログラムにより探すことができます。最大値は、`fabs` という関数で絶対値を比較することにより求めます。

ここで、もし最大値がゼロの場合、行列は特異 (行列式がゼロ) ということになり、解は一意的に決まりません。その場合、関数の値としてゼロを返し、そのことをコールした関数に伝えるのが良いでしょう。

```
big=0.0;
for(i=ipv ; i<=n ; i++){
    if(fabs(a[i][ipv]) > big){
        big = fabs(a[i][ipv]);
        pivot_row = i;
    }
}
if(big == 0.0){return(0);}
row[ipv] = pivot_row;
```

このプログラムは、以下のことを行っています。

- `big` にその列の絶対値の最大が入る。
- 最大値 (新しいピボット) がある行は、`pivot_row` である。
- ピボットがゼロの場合、行列は特異です。その場合、処理を中断して、呼び出し側にゼロを返します。行列が正則な場合、1 を返すこととなりますが、これはこの関数の最後に書きます。
- *ipv* 番目に最大値になった行を、配列 `row[ipv]` に入れる。これは、後で使うことになる。

2.2.2 行の交換

ピボットとすべき値がある行 (`pivot_row`) がわかったので、*ipv* 行と入れ替えます。

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & * & * & * & * \\ 0 & 1 & * & * & * & * \\ 0 & 0 & \odot & \odot & \odot & \odot \\ 0 & 0 & * & * & * & * \\ 0 & 0 & \otimes & \otimes & \otimes & \otimes \\ 0 & 0 & * & * & * & * \end{pmatrix} \Rightarrow \mathbf{A}' = \begin{pmatrix} 1 & 0 & * & * & * & * \\ 0 & 1 & * & * & * & * \\ 0 & 0 & \otimes & \otimes & \otimes & \otimes \\ 0 & 0 & * & * & * & * \\ 0 & 0 & \odot & \odot & \odot & \odot \\ 0 & 0 & * & * & * & * \end{pmatrix}$$

入れ替えは、係数行列 \mathbf{A} と非同時項 \mathbf{b} の両方について行います。変数を入れ替える場合、一時的な変数を記憶する場所が必要です。ここでは、`temp` という変数を使っています。

ただし、もともと最大の値が *ipv* 行にある場合は、行の入れ替えは行いません。

```
if(ipv != pivot_row){
    for(i=1 ; i<=n ; i++){
        temp = a[ipv][i];
        a[ipv][i] = a[pivot_row][i];
```

```

    a[pivot_row][i] = temp;
}

temp = b[ipv];
b[ipv] = b[pivot_row];
b[pivot_row] = temp;

}

```

2.2.3 部分ピボット選択付きガウス・ジョルダン法のソースプログラム

この2.2節をまとめ、2.1節の素朴なガウス・ジョルダン法とあわせると、部分ピボット選択付きのガウス・ジョルダン法が完成します。

```

/* ===== ガウスジョルダン法の関数===== */

int gauss_jordan(int n, double a[][MAXN+10], double b[]){

    int ipv, i, j;
    double inv_pivot, temp;
    double big;
    int pivot_row, row[MAXN+10];

    for(ipv=1 ; ipv <= n ; ipv++){

        /* ---- 最大値探索 ----- */
        big=0.0;
        for(i=ipv ; i<=n ; i++){
            if(fabs(a[i][ipv]) > big){
                big = fabs(a[i][ipv]);
                pivot_row = i;
            }
        }
        if(big == 0.0){return(0);}
        row[ipv] = pivot_row;

        /* ---- 行の入れ替え ----- */
        if(ipv != pivot_row){
            for(i=1 ; i<=n ; i++){
                temp = a[ipv][i];
                a[ipv][i] = a[pivot_row][i];
                a[pivot_row][i] = temp;
            }
            temp = b[ipv];
            b[ipv] = b[pivot_row];
            b[pivot_row] = temp;
        }
    }
}

```

```

/* ---- 対角成分=1(ピボット行の処理) ----- */
inv_pivot = 1.0/a[ipv][ipv];
for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
}
b[ipv] *= inv_pivot;

/* ---- ピボット列=0(ピボット行以外の処理) ---- */
for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}

return(1);
}

```

2.3 逆行列計算ルーチンの追加

逆行列を計算するルーチンの追加は難しそうですが、実はそんなに難しいものではありません。単位行列を、係数行列 \mathbf{A} と同じ処理をすればよいのです。係数行列 \mathbf{A} が単位行列に変形されたならば、元の単位行列は逆行列に変換されます。これは、簡単に実現できます。

$$\mathbf{A} = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{pmatrix} \Rightarrow \mathbf{A}' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{A}^{-1'} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \mathbf{A}^{-1} = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{pmatrix}$$

これを実現するためには、以下の2つのことをすればよいだけです。

- 単位行列を作成する。
- 作成された単位行列を、係数行列 A を同じ操作をする。

2.3.1 単位行列の作成

まず、単位行列を作成する必要があります。これは、以下のようにすればよいでしょう。

```
for(i=1 ; i<=n ; i++){
    for(j=1 ; j<=n ; j++){
        if(i == j){
            inv_a[i][j]=1.0;
        }else{
            inv_a[i][j]=0.0;
        }
    }
}
```

2.3.2 逆行列の計算

単位行列 inv_a ができたので、これを係数行列 A と同じ処理をすれば、逆行列に変換できます。具体的には、2.2.3 節のプログラムを以下のように書き換えます。

- 部分ピボット選択で係数行列の行の交換を行ったならば、同じように inv_a も行の交換を行う。この部分の処理を

```
temp = a[ipv][i];
a[ipv][i] = a[pivot_row][i];
a[pivot_row][i] = temp;
temp = inv_a[ipv][i];          /* -- これ追加 -- */
inv_a[ipv][i] = inv_a[pivot_row][i]; /* -- これ追加 -- */
inv_a[pivot_row][i] = temp;    /* -- これ追加 -- */
```

と書き換えます。

- 対角成分を 1 にするためにピボット行をピボットの値で割るところでは、同じ値で割る。これも、

```
a[ipv][j] *= inv_pivot;
inv_a[ipv][j] *= inv_pivot;          /* -- これ追加 -- */
```

と書き換えます。

- ピボット行以外のピボット列を0にするために、ピボット行の定数倍を引くところでも同じ操作をする。これも、

```
a[i][j] -= temp*a[ipv][j];
inv_a[i][j] -= temp*inv_a[ipv][j];  /* -- これ追加 -- */
```

と書き換えます。

2.3.3 逆行列計算付きガウス・ジョルダン法のソースプログラム

いままで述べたことを全て網羅したガウス・ジョルダン法の計算の関数は、次のようになります。

```
/* ===== ガウスジョルダン法の関数===== */

int gauss_jordan(int n, double a[][MAXN+10], double b[],
                 double inv_a[][MAXN+10]){

    int ipv, i, j;
    double inv_pivot, temp;
    double big;
    int pivot_row, row[MAXN+10];

    /* ---- 単位行列作成 ----- */
    for(i=1 ; i<=n ; i++){
        for(j=1 ; j<=n ; j++){
            if(i==j){
                inv_a[i][j]=1.0;
            }else{
                inv_a[i][j]=0.0;
            }
        }
    }

    for(ipv=1 ; ipv <= n ; ipv++){

        /* ---- 最大値探索 ----- */
        big=0.0;
        for(i=ipv ; i<=n ; i++){
            if(fabs(a[i][ipv]) > big){
                big = fabs(a[i][ipv]);
                pivot_row = i;
            }
        }
        if(big == 0.0){return(0);}
        row[ipv] = pivot_row;

        /* ---- 行の入れ替え ----- */
```

```

if(ipv != pivot_row){
    for(i=1 ; i<=n ; i++){
        temp = a[ipv][i];
        a[ipv][i] = a[pivot_row][i];
        a[pivot_row][i] = temp;
        temp = inv_a[ipv][i];
        inv_a[ipv][i] = inv_a[pivot_row][i];
        inv_a[pivot_row][i] = temp;
    }
    temp = b[ipv];
    b[ipv] = b[pivot_row];
    b[pivot_row] = temp;
}

/* ---- 対角成分=1(ピボット行の処理) ----- */
inv_pivot = 1.0/a[ipv][ipv];
for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
    inv_a[ipv][j] *= inv_pivot;
}
b[ipv] *= inv_pivot;

/* ---- ピボット列=0(ピボット行以外の処理) ---- */
for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
            inv_a[i][j] -= temp*inv_a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}

}

return(1);
}

```

2.4 メモリー、計算効率の改善

昔、といってもそんなに過去のことでありません。プログラマーは出来るだけメモリーを大事に使いました。当時、使えるメモリーが限られていたので、その資源の有効活用しなくてはなりませんでした。いまでも、パソコンで 1G Byte のメモリーを使うのは何でもありませんが、たった 10 年ほど前のメインフレームと言われた大型のコンピューターでさえ、1 つのプログラムが使える領域は 10M Byte 程度でした。

メモリーと合わせて、計算効率も重要でした。大規模な計算になると、計算が終了するまで何日も費やす

場合があります。そのような場合、プログラムの改良により、速度が10%アップするとかなりのメリットがあります。

そこで、ここではメモリーの効率的な利用を考えましょう。

2.4.1 メモリーと計算の効率化

これまでの計算過程を考えてみましょう。 i 行 ipv 列までの処理が完了したとき、係数行列 \mathbf{A} を示す配列 $\mathbf{A}[i][j]$ と逆行列 \mathbf{A}^{-1} が最終的に格納される配列 $\text{inv_a}[i][j]$ の状態を見てみましょう。それぞれは、

$$\mathbf{A}' = \begin{pmatrix} 1 & 0 & 0 & * & * & * \\ 0 & 1 & 0 & * & * & * \\ 0 & 0 & 1 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \end{pmatrix}$$

$$\mathbf{A}^{-1'} = \begin{pmatrix} (*) & (*) & (*) & 0 & 0 & 0 \\ (*) & (*) & (*) & 0 & 0 & 0 \\ (*) & (*) & (*) & 0 & 0 & 0 \\ (*) & (*) & (*) & 1 & 0 & 0 \\ (*) & (*) & 0 & 0 & 1 & 0 \\ (*) & (*) & 0 & 0 & 0 & 1 \end{pmatrix}$$

となっているはずですが。この状態は、2.3.3 節の $i=4, ipv=3$ が完了したときです。この行列を良く見ると、係数行列 \mathbf{A}' では i 行 ipv 列までは、役に立つ情報をもっていないことになります²。同様に、 $\mathbf{A}^{-1'}$ 行列は、 \mathbf{A}' では i 行 ipv 列以降は役に立つ情報が無いことが分かります。これらの情報として役に立たない成分 0, 1 は、メモリーの無駄遣いなので、

$$\mathbf{A}' = \begin{pmatrix} (*) & (*) & (*) & * & * & * \\ (*) & (*) & (*) & * & * & * \\ (*) & (*) & (*) & * & * & * \\ (*) & (*) & (*) & * & * & * \\ (*) & (*) & * & * & * & * \\ (*) & (*) & * & * & * & * \end{pmatrix}$$

としたくなります。そうすると、メモリーが半分ですみます。これは、 $n = 10000$ の行列とすると、800M Byte の節約になります。

これを実現するのは、簡単です。次のようにプログラムを書けばよいのです。

```
/* ---- 対角成分=1(ピボット行の処理) ----- */
inv_pivot = 1.0/a[ipv][ipv];
```

²プログラム作成中のデバックでは別で、間違いを探すときに重要な情報をもたらします。

```

a[ipv][ipv]=1.0;          /* --- この行を追加 --- */
for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
}
b[ipv] *= inv_pivot;

/* ---- ピボット列=0(ピボット行以外の処理) ---- */
for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        a[i][ipv]=0.0;      /* --- この行を追加 --- */
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}

```

このようにするとメモリーは、2.3.3節のプログラムに比べて半分ですみます。さらに、計算時間も半分になります。2.3.3節では、計算結果が0や1の場合も計算していたのですが、このプログラムではそれを省いています。

2.4.2 逆行列の列の入れ替え

これで、メモリーと計算の効率化が図れたのですが、このままでは \mathbf{A}^{-1} を示す `inv.a[i][j]` の配列は、 \mathbf{A} の逆行列になっていません。ピボット選択により、行を入れ替えた行列 \mathbf{A}' の逆行列になっています。そこで、元の行列 \mathbf{A} の逆行列にするために、 \mathbf{A}^{-1} の列を入れ替える必要があります。なぜ、列を入れ替える必要が?。それは、以下のように考えます。

当然、 \mathbf{A} の逆行列ということは、乗算すると下のように単位行列になります。

$$\begin{pmatrix} * & * & * & * & * & \dots & * \\ * & * & * & * & * & \dots & * \\ \hline a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} & \dots & a_{in} \\ * & * & * & * & * & \dots & * \\ \vdots & & & & & \ddots & \vdots \\ * & * & * & * & * & \dots & * \end{pmatrix} \begin{pmatrix} * & * & * & \alpha_{1j} & * & \dots & * \\ * & * & * & \alpha_{2j} & * & \dots & * \\ * & * & * & \alpha_{3j} & * & \dots & * \\ * & * & * & \alpha_{4j} & * & \dots & * \\ \vdots & & & \vdots & & \ddots & \vdots \\ * & * & * & \alpha_{nj} & * & \dots & * \end{pmatrix} = \begin{pmatrix} 1 & & & & & & 0 \\ & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ 0 & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

すなわち、元の行列の行ベクトルと逆行列の列ベクトルの内積が、

$$(\text{元の行列の } i \text{ 行の行ベクトル}) \cdot (\text{逆行列の } j \text{ 列の列ベクトル}) = \delta_{ij}$$

という関係を満たしていることを意味します。 δ_{ij} は、クロネッカーの記号で、 $i = j$ のときその値は1、 $i \neq j$ のときその値は0です。

このようなわけで、元の行列の行を入れ替えた場合、その逆行列は元の行列の逆行列の列を入れ替えたものになります。従って、ピボット選択により係数行列の行を入れ替えると、逆行列の列を入れ替える必要が生じます。実際にプログラムでは、以下のようにします。

```

/* ---- 列の入れ替え (逆行列) ----- */
for(j=n ; j>=1 ; j--){
    if(j != row[j]){
        for(i=1 ; i<=n ; i++){
            temp = a[i][j];
            a[i][j]=a[i][row[j]];
            a[i][row[j]]=temp;
        }
    }
}

```

2.4.3 効率化したガウス・ジョルダン法のソースプログラム

これで、ほとんどガウス・ジョルダン法の計算の関数は、完成です。この関数は、かなり実用に使えるでしょう。残っている問題は、

- 列交換による完全ピボット選択の改良。これはそんなに難しくない。
- 丸め誤差を考慮した特異行列の判定。これは大変難しい。

くらいでしょう。

これ以上、改良するのは大変なので、ほとんど問題なく使える関数のプログラムをいかに示します。

```

/* ===== ガウスジョルダン法の関数===== */

int gauss_jordan(int n, double a[][MAXN+10], double b[]){

    int ipv, i, j;
    double inv_pivot, temp;
    double big;
    int pivot_row, row[MAXN+10];

    for(ipv=1 ; ipv <= n ; ipv++){

        /* ---- 最大値探索 ----- */
        big=0.0;
        for(i=ipv ; i<=n ; i++){
            if(fabs(a[i][ipv]) > big){
                big = fabs(a[i][ipv]);
                pivot_row = i;
            }
        }
        if(big == 0.0){return(0);}
        row[ipv] = pivot_row;
    }
}

```

```

/* ---- 行の入れ替え ----- */
if(ipv != pivot_row){
    for(i=1 ; i<=n ; i++){
        temp = a[ipv][i];
        a[ipv][i] = a[pivot_row][i];
        a[pivot_row][i] = temp;
    }
    temp = b[ipv];
    b[ipv] = b[pivot_row];
    b[pivot_row] = temp;
}

/* ---- 対角成分=1(ピボット行の処理) ----- */
inv_pivot = 1.0/a[ipv][ipv];
a[ipv][ipv]=1.0;
for(j=1 ; j <= n ; j++){
    a[ipv][j] *= inv_pivot;
}
b[ipv] *= inv_pivot;

/* ---- ピボット列=0(ピボット行以外の処理) ---- */
for(i=1 ; i<=n ; i++){
    if(i != ipv){
        temp = a[i][ipv];
        a[i][ipv]=0.0;
        for(j=1 ; j<=n ; j++){
            a[i][j] -= temp*a[ipv][j];
        }
        b[i] -= temp*b[ipv];
    }
}

}

/* ---- 列の入れ替え(逆行列) ----- */
for(j=n ; j>=1 ; j--){
    if(j != row[j]){
        for(i=1 ; i<=n ; i++){
            temp = a[i][j];
            a[i][j]=a[i][row[j]];
            a[i][row[j]]=temp;
        }
    }
}

return(1);
}

```

3 出来上がった関数の使い方

完成した 2.4.3 節のガウス・ジョルダン法を計算する関数は、次のようにして使います。もし行列が特異な場合、そのことを表示してプログラムが止まるようになっています。

```
if(gauss_jordan(n, a, b) == 0){
    printf("singular matrix !!!\n");
    exit(0);
};
```

引数と戻り値は、次のようになっています。

- **n** が連立方程式の次元を示す整数です。 **a** が係数行列を示す 2 次元配列、 **b** が同時項をしめす 1 次元配列です。計算結果、逆行列が **a** に、解が **b** に格納されます。
- 係数行列が特異の場合、即ち行列式がゼロの場合、この関数は整数のゼロを戻り値として返します。