## 0.7 ジュニア版　　記述統計量

モデルをもった回帰分析に進む前に、平均や分散から始まって、データの性質を知るための様々な値を１から求めてみましょう。すでにコマンドとして組み込まれているものもありますが、それらをあらためてプログラムするということはその仕組みを実感すると同時に、今後進んでいくプログラミングの土台の部分となります。（下では、もうすでに組込み関数として存在する関数を１から作成するので、名前を区別するために敢えて日本語名をつけます。世界共通語のプログラムの通常関数名は英語の略称が望ましいことは言うまでもありません。）

では、これまでと同じようにプログラムエディター上にデータをカット＆ペーストまたは手で打ち込む方法でデータを入力して、それを利用して各種の統計の値を調べてみましょう。ただし、組込み関数は極力使わないものとします。

**合計**

プログラム

```
new;
cls;
let data[10,3]=
1   3 10
2   7   9
3   8   8
4   2   7
5   1   6
6   4   5
7 10   4
8   9   3
9   6   2
10 5   1
;
print goukei(data);

proc goukei(x);
    local n,i,sum;
    n=rows(x);
    sum=0;
    i=1;
```

```
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    retp(sum');
endp;
```

画面表示

```
          55.000000
          55.000000
          55.000000
```

プログラム各行の説明

１行目　メモリを初期化する

２行目　画面をクリアする

３～１４行目　変数 data に１０×３のディメンションでセミコロンまでのデータを入れる

１５行目　自作関数 goukei を呼び出して、その中に変数 data を代入して、その答えを画面表示させる

１７～２７行目　proc から endp;までが goukei という名前でインプットｘの関数

１８行目　この関数の中だけで使われるローカル変数は n,i,sum の３つ

１９行目　組込み関数 rows を用いて変数ｘの行数を調べて、それを変数ｎに入れる

２０行目　今から足しあわせていく変数 sum の中に０を入れておく

２１行目　do ループに入る前にそのカウンタｉを１に設定しておく

２２行目　カウンタｉがｎ以下「の間は」endo;までのことを繰り返す

２３行目　変数 sum と変数ｘのｉ行目を足し合わせて、あらためて変数 sum に入れる（x[i,.]のドットマークは「すべての」列という意味）

２４行目　カウンタｉを１増やして、あらためてｉとする

２５行目　do ループの終わり（ここまでを繰り返す）

２６行目　変数 sum を転置させたものをこの関数のリターンとして外部に返す（GAUSS の内部関数と同じ出力形式にするため転置させただけである）

２７行目　この procedure の終わり（ここまでが１つの procedure）

上のプログラムではデータの部分を手で入力するか、または Copy&Paste で貼りつけたものを利用します。後半に置いている goukei という関数を通常の組込み関数と同じように呼び出してその中に変数を入れて print 命令で画面表示させています。

プログラム中のデータと同じ次のような１０行３列の 10×３の配列の行列を考えよう。

$$data = \begin{bmatrix} & 1列 & 2列 & 3列 \\ 1行 & 1 & 3 & 10 \\ 2行 & 2 & 7 & 9 \\ 3行 & 3 & 8 & 8 \\ 4行 & 4 & 2 & 7 \\ 5行 & 5 & 1 & 6 \\ 6行 & 6 & 4 & 5 \\ 7行 & 7 & 10 & 4 \\ 8行 & 8 & 9 & 3 \\ 9行 & 9 & 6 & 2 \\ 10行 & 10 & 5 & 1 \end{bmatrix}$$

GAUSS では配列は

**変数[ 行 , 列 ]**

と表される。例えば data[5,3]は data の５行３列目の成分のことで、この場合６になる。同様にして、data[10,1]は data の 10 行１列目の成分で 10 になっている。また、列または行をワイルドカードにすることも可能である。例えば、data[1,.]は data の１行目すべてを表し、この場合{ 1 3 10}のことである。同様にして、data[10,.]は 10 行目すべてで{10 5 1}のことである。また列方向に考えると data[.,1]は１列目すべてのことで{1,2,3,4,5,6,7,8,9,10}のことである。これを利用して、カウンタｉを１から１つずつ増して 10 までを考えると

$$\begin{bmatrix} & 1列 & 2列 & 3列 \\ i & 1 & 3 & 10 \\ & 2 & 7 & 9 \\ & 3 & 8 & 8 \\ & 4 & 2 & 7 \\ & 5 & 1 & 6 \\ & 6 & 4 & 5 \\ & 7 & 10 & 4 \\ & 8 & 9 & 3 \\ & 9 & 6 & 2 \\ & 10 & 5 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} & 1列 & 2列 & 3列 \\ & 1 & 3 & 10 \\ i & 2 & 7 & 9 \\ & 3 & 8 & 8 \\ & 4 & 2 & 7 \\ & 5 & 1 & 6 \\ & 6 & 4 & 5 \\ & 7 & 10 & 4 \\ & 8 & 9 & 3 \\ & 9 & 6 & 2 \\ & 10 & 5 & 1 \end{bmatrix} \cdots \rightarrow \cdots \begin{bmatrix} & 1列 & 2列 & 3列 \\ & 1 & 3 & 10 \\ & 2 & 7 & 9 \\ & 3 & 8 & 8 \\ & 4 & 2 & 7 \\ & 5 & 1 & 6 \\ & 6 & 4 & 5 \\ & 7 & 10 & 4 \\ & 8 & 9 & 3 \\ & 9 & 6 & 2 \\ i & 10 & 5 & 1 \end{bmatrix}$$

というふうにｉ行目を表す data[i,.]が data[1,.],data[2,.],…data[10,.]と移動していく。この

ことを利用して、sum=0 としておいた変数 sum に１行目から順に最大行数の 10 行目までを足し合わせたものが最終的に sum となって、それをアウトプットリターンにしている。この場合、それぞれの列はそれぞれの列のことろにある数と足し合わされていくので、列のことは気にする必要がない。すなわち、ドットのワイルドカードとして考えています。もう少し対応関係の例を示しておくと、

data[1,1]　　**行列変数 data の１行１列目**

data[10,1]　　**行列変数 data の 10 行１列目**

data[ . ,1]　　**行列変数 data の（行はワイルドカードで）１列目すべて**

data[1, . ]　　**行列変数 data の１行目すべて（列はワイルドカード）**

のようになります。

## 平均

今度は上の列ごとの合計をもとにして少しだけ変更して平均を求めてみます。

プログラム

```
new;
cls;
let data[10,3]=
1   3 10
2   7   9
3   8   8
4   2   7
5   1   6
6   4   5
7 10   4
8   9   3
9   6   2
10 5   1
;
print heikin(data);

proc heikin(x);
    local n,i,sum,m;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
```

```
        sum=sum+x[i,.];
        i=i+1;
    endo;
    m=sum/n;
    retp(m');
endp;
```

画面表示

```
        5.5000000
        5.5000000
        5.5000000
```

プログラム各行の説明（前のプログラムとの相違点）

１８行目　ローカル変数は n,i,sum に加えて m の４つである

２６行目　変数 sum を行数ｎで割ったものをｍとする

２７行目　変数ｍを転置させたものをこの関数のアウトプットリターンとして返す

平均の関数は合計の関数までのプログラムを事実上１箇所だけでできてしまいます。つまり、合計で行ごとに足し合わせてできた変数 sum を行数ｎで割ってやるだけでよいわけです。このように、関数を少し変更するには、新たにその関数内部で使うローカル変数を local 命令の後に加えることを忘れないで、関数名と retp( )の括弧の中のアウトプットリターンの変数を新しく計算した変数に変更するだけでできます。

注意事項

１）プログラムでは何も入っていない変数にいきなり何かを足したり引いたりできない。したがって、sum には必ず何か計算の支障にならない数を入れておく必要がある（この場合は０としている）。

２）この sum もカウンタの ｉ も共にループに入る前に外側で初期値を設定する必要がある。ループの内部で設定すると毎回その初期値に戻ってしまって都合が悪くなる。

３）GAUSS の組込み関数では、次に続く計算のことを考えて、計算結果は列として出てくることが多いので、自作関数の場合もこれにならい最後に行を列に転置させている。

## 最大値

新しい概念であるソート関数を用いて、これまでどおり行列を列ごとに考えて、小さいものから大きいものものに並べ替えてみましょう。GAUSS では、ソートはすでに高速なものが組込み関数としてあるので、そこから出発することが効率的です。その使い方は、

**sortc(変数,列番号)**

としてやると、「その列番号のところを基準にして」「行としてまとめて」データが小さいものから大きいものへとソートされます。技術的には、私たちは複数の列を考えていますから、行列を１列ずつ取り出して、i=1 から列数まで１列ごとに第１列目のソートを繰り返します。すなわち、

**変数[ . ,i]=sortc(変数[ . ,i],1)**

を do while ループで i を１つずつ動かしてそれぞれの列ごとにソートします。

プログラム

```
（　これ以前は省略　）
print saidai(data);

proc saidai(x);
    local n,k,i,max;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    max=x[n,.];
    retp(max');
endp;
```

画面表示

```
        10.000000
        10.000000
        10.000000
```

プログラム（procedure 部分）各行の説明

１～１２行目 proc から endp までが saidai という関数名でインプット変数がｘの関数

２行目　この関数の中だけで使われるローカル変数は、n,k,i,max の４つ
３行目　組込み関数 rows を用いて変数ｘの行数を調べて、それを変数ｎに入れる
４行目　組込み関数 cols を用いて変数ｘの列数を調べて、それを変数ｋに入れる
５行目　ループのカウンタｉを１に設定しておく
６～９行目　do while から endo までをｉ≦ｋであるあいだ繰り返す
７行目　変数ｘのｉ列目からなる列ベクトルを（第１列目を基準にして）小さいものから大きいものへソートして、あらためて変数ｘのｉ列目に入れる
８行目　カウンタｉを１だけ増して、あらためてｉに入れる
10 行目　変数ｘのｎ行目のすべてを変数 max に入れる
11 行目　変数 max を転置したものをこの関数のアウトプットリターンとして外部に返す

すこしめんどうかもしれませんが、わからない場合には、

```
i=1;
do while i<=k;
    x[.,i]=sortc(x[.,i],1);
    i=i+1;
endo;
```

の５行をソートをするものと考えて今後利用してみてください。ｉ列目だけを（１列しかありませんが、第１列目を基準に）ソートして、それをもとの変数のｉ列目に入れています。ループをまわさなければ、ただ第１列目だけに注目したソートになっていまいます。その他の列は、第１列めに行ごとに一括ソートされてめちゃくちゃになっていまいます。なお、ｎは変数ｘの行数のことですから、ワイルドカードのドットを用いて、x[n,.]は変数ｘのｎ行目となります。これにより、列ごとにソートされた後の第ｎ行目、すなわち最大値ばかりが入った行ベクトルが取り出せます。それを最終的に GAUSS の出力方法に合わせて、転置させてアウトプットリターンとしています。

## 最小値

列ごとの最大値を求めたことを発展させて、簡単に最小値も求められます。ソートした後の結果のｎ行目ではなくて、最小値は第１行目になります。

プログラム

```
（　これ以前は省略　）
print saisho(data);

proc saisho(x);
    local n,k,i,min;
    n=rows(x);
```

```
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    min=x[1,.];
    retp(min');
endp;
```

画面表示

```
        1.0000000
        1.0000000
        1.0000000
```

上のように、列ごとのソート結果の変数ｘの１行目（列はワイルドカード）を min と置いて、それを GAUSS の出力に合わせて転置させてアウトプットリターンとしています。なお、上で扱った行列の列ごとの「合計」「平均」「最大」「最小」は、GAUSS の組込み関数

```
        sumc(data);
        meanc(data);
        maxc(data);
        minc(data);
```

と直接しても求められます。上の結果が同じになるのか自分で確認してみましょう。

## 範囲（レンジ）

最大値と最小値がわかっていれば、それらの差である

レンジ　＝　最大値　－　最小値

も最大と最小のプログラムを合わせて少し改造すれば求まるはずです。

プログラム

```
（　これ以前は省略　）
print hani(data);

proc hani(x);
    local n,k,i,min,max,range;
    n=rows(x);
    k=cols(x);
```

```
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    min=x[1,.];
    max=x[n,.];
    range=max-min;
    retp(range');
endp;
```

画面表示

```
        9.0000000
        9.0000000
        9.0000000
```

プログラム各行の説明（前のプログラムとの相違点）

10 行目　ソート後の行列変数ｘの１行目（列はワイルドカード）を変数 min に入れる
11 行目　ソート後の行列変数ｘのｎ行目（列はワイルドカード）を変数 max に入れる
12 行目　max から min を引いたものを変数 range に入れる
13 行目　変数 range を転置させたものをアウトプットリターンとして外部に返す

最大と最小のプログラムを両方一緒に行なって、最後にその差をとっています。もし以前からのプログラムを変更してこのプログラムを書くには、procedure の最初の部分の名前を例えば hani て、その次の行でローカル変数に max と min の両方に加えて、新しく range があるか確認します。もちろん、この procedure を呼び出す場合も新しい procedure 名である hani の丸括弧の中に変数 data を入れる必要があります。

## 中位値

今度は平均と並んでよく使われる、データの並びの真ん中の順位にある値を求めてみます。ただし、データの個数が偶数の場合（例えば 10 個の場合）真ん中には数はデータがありませんからその前後の平均を計算します。そこで、If 分岐を導入してプログラムしてみましょう。つまり、データの個数が偶数の場合と奇数の場合に分岐させます。下のプログラムでは、その判断をするためにデータの個数ｎを２で割った数を四捨五入した数とｎ割る２が等しい場合（偶数に相当）とそうでない場合（奇数に相当）に分岐させています。

プログラム

```
（　これ以前は省略　）
print chuichi(data);
```

```
proc chuichi(x);
    local n,k,i,med;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    if n/2==round(n/2);
        med=(x[n/2,.]+x[n/2+1,.])/2;
    else;
        med=x[(n+1)/2,.];
    endif;
    retp(med');
endp;
```

画面表示

```
          5.5000000
          5.5000000
          5.5000000
```

## パーセント点

プログラム

```
（　これ以前は省略　）
print percent(data,0.1);
print percent(data,0.5);
print percent(data,0.9);

proc percent(x,a);
    local n,k,i,per;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
```

```
        i=i+1;
    endo;
    per=x[round(n*a),.];
    retp(per);
endp;
```

画面表示

```
        1.0000000        1.0000000        1.0000000
        5.0000000        5.0000000        5.0000000
        9.0000000        9.0000000        9.0000000
```

## 四分位点

プログラム

```
（　これ以前は省略　）
print "1st" shibuni(data,0.25);
print "2nd" shibuni(data,0.5);
print "3rd" shibuni(data,0.75);
e={0.25,0.5,0.75};
quantile(data,e);

proc shibuni(x,p);
    local n,k,i,a,b,f,q;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    a=floor(n*p);
    f=n*p-a;
    q=x[a,.]+f*(x[a+1,.]-x[a,.]);
    retp(q);
endp;
```

画面表示

```
1st         2.5000000          2.5000000          2.5000000
2nd          5.0000000          5.0000000          5.0000000
```

```
3rd          7.5000000          7.5000000          7.5000000

          2.5000000          2.5000000          2.5000000
          5.0000000          5.0000000          5.0000000
          7.5000000          7.5000000          7.5000000
```

### 「中位値」「パーセント点」「四分位点」の違い

これら３つは、定義によっては、等しくなる。しかしながら、GAUSS をはじめとする統計計量ソフトウエアでは、中位値と第２四分位点は通常一致しない。四分位計算では補正が行なわれているからだ。また、四捨五入した 50%の点であるパーセント点とも異なる。

プログラム

```
new;
cls;
rndseed 1000;
data=rndn(99,2);

print "median" chuichi(data)';
print "     50%" percentile(data,0.5)';
print "     1st" shibuni(data,0.25);
print "     2nd" shibuni(data,0.5);
print "     3rd" shibuni(data,0.75);

proc shibuni(x,p);
    local n,k,i,a,b,f,q;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    a=floor(n*p);
    f=n*p-a;
    q=x[a,.]+f*(x[a+1,.]-x[a,.]);
```

```
    retp(q);
endp;

proc chuichi(x);
    local n,k,half,i,med;
    n=rows(x);
    k=cols(x);
    half=n/2;
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    if half==round(half);
        med=(x[n/2,.]+x[n/2+1,.])/2;
    else;
        med=x[n/2,.];
    endif;
    retp(med');
endp;

proc percentile(x,p);
    local n,k,i,xp;
    n=rows(x);
    k=cols(x);
    i=1;
    do while i<=k;
        x[.,i]=sortc(x[.,i],1);
        i=i+1;
    endo;
    xp=x[round(p*n),.];
    retp(xp');
endp;
```

画面表示

| | | |
|---|---|---|
| median | -0.11799732 | 0.22035185 |
| 50% | -0.10196121 | 0.22575022 |

```
1st       -0.71336306        -0.46742059
2nd       -0.10997926         0.22305103
3rd        0.82544983         0.66333897
```

プログラムを見てわかるように、GAUSS の動きをそのまま忠実に再現する定義では、中位値（メディアン）は、データ数が奇数の場合には小さい方から並べなおして中間の番目の値、偶数の場合にはその前後２つの平均値である。一方、単純パーセント点は、データ数にパーセントの小数値をかけたものを四捨五入しして整数化したものの番目の値である。最後に、四分位計算で求める第２四分位点はａとａ＋１の間に位置する実際のパーセント番目をｘのａ番目とａ＋１番目を比例配分するように補正した値にする。最初の１から 10 までのデータなのではこれらは一致することもあるが、上のように通常は一致しない。教科書によって、あるいはソフトウエアによって定義が異なるおそれがあるので、使う前に比較計算が必要である。上の四捨五入の単純パーセント点を、複雑な計量計算で、簡単化のために使うこともよくあることなのでどれが誤りでどれが正しいということではない。

## 分散

プログラム

```
new;
cls;
let data[10,3]=
1   3 10
2   7   9
3   8   8
4   2   7
5   1   6
6   4   5
7 10   4
8   9   3
9   6   2
10 5   1
;
print bunsan(data);

proc bunsan(x);
    local n,sum,i,xbar,s2;
    n=rows(x);
```

```
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    s2=sumc((x-xbar)^2)/(n-1);
    retp(s2);
endp;
```

画面表示

```
        9.1666667
        9.1666667
        9.1666667
```

ただし、上の計算では n-1 で割るバージョンである。サンプルではなくてポピュレーションを考えなくてはいけない場合やその他の理由によりnで割るバージョンも多用される。GAUSS では常に上のような n-1 で割るバージョンの計算なので、それが不都合ならば適宜自分で上のような関数を作り n-1 で割るところを n で割るものにする必用がある。

## 変動係数

プログラム

```
（　これ以前は省略　）
print cv(data);

proc cv(x);
    local n,sum,i,xbar,s;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    s=sqrt(sumc((x-xbar)^2)/(n-1));
    retp(s./xbar');
```

```
endp;
```

画面表示

```
0.55048188
0.55048188
0.55048188
```

## 標準化

プログラム

```
（　これ以前は省略　）
print standard(data);

proc standard(x);
    local n,sum,i,xbar,s,z;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    s=sqrt(sumc((x-xbar)^2)/(n-1));
    z=(x-xbar)./s';
    retp(z);
endp;
```

画面表示

```
-1.4863011   -0.82572282    1.4863011
-1.1560120    0.49543369    1.1560120
-0.82572282   0.82572282    0.82572282
-0.49543369  -1.1560120     0.49543369
-0.16514456  -1.4863011     0.16514456
 0.16514456  -0.49543369   -0.16514456
 0.49543369   1.4863011    -0.49543369
 0.82572282   1.1560120    -0.82572282
```

```
1.1560120        0.16514456        -1.1560120
1.4863011       -0.16514456        -1.4863011
```

## 歪度（スキューネス）

```
（　これ以前は省略　）
print hizumi(data);

proc hizumi(x);
    local n,sum,i,xbar,s,z,sk;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    s=sqrt(sumc((x-xbar)^2)/(n-1));
    z=(x-xbar)./s';
    sk=sumc((z)^3)/n;
    retp(sk);
endp;
```

画面表示

```
 -4.4408921e-017
 -4.4408921e-017
  4.4408921e-017
```

## 尖度（クルトーシス）

プログラム

```
（　これ以前は省略　）
print togari(data);
```

```
proc togari(x);
    local n,sum,i,xbar,s,z,kur;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    s=sqrt(sumc((x-xbar)^2)/(n-1));
    z=(x-xbar)./s';
    kur=sumc((z)^4)/n;
    retp(kur);
endp;
```

画面表示

```
        1.4383636
        1.4383636
        1.4383636
```

## 共分散

プログラム

```
（　これ以前は省略　）
print kyobun(data);

proc kyobun(x);
    local n,sum,i,xbar,cov;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
```

```
    cov=(x-xbar)'(x-xbar)/(n-1);
    retp(cov);
endp;
```

画面表示

```
        9.1666667        2.3888889       -9.1666667
        2.3888889        9.1666667       -2.3888889
       -9.1666667       -2.3888889        9.1666667
```

## 相関係数

プログラム

```
（　これ以前は省略　）
print soukan(data);

proc soukan(x);
    local n,sum,i,xbar,cov,s,corr;
    n=rows(x);
    sum=0;
    i=1;
    do while i<=n;
        sum=sum+x[i,.];
        i=i+1;
    endo;
    xbar=sum/n;
    cov=(x-xbar)'(x-xbar)/(n-1);
    s=sqrt(sumc((x-xbar)^2)/(n-1));
    corr=cov./(s.*s');
    retp(corr);
endp;
```

画面表示

```
        1.0000000        0.26060606       -1.0000000
       0.26060606        1.0000000       -0.26060606
       -1.0000000       -0.26060606        1.0000000
```

以上、普段よく見かけるような単純な「記述統計量」でさえ、考え方や定義により微妙な違いが生じることがよくわかったと思います。プログラミングとは、これらの違いをそのまま受け入れるのではなくて、自分なりにカスタマイズして利用することにあります。

そうしたカスタマイズの延長上に、いくつかある条件のうち１つを緩めて、新しいあるいは高度な統計や計量の計算をすることがあります。

**練習問題**

【問１】

上の補正された四分位計算を使って、

**（四分位範囲）＝（第３四分位点）－（第１四分位点）**

となる四分位範囲を求めると同時に、標準偏差を n-1 で割る従来の分散を求める計算のルートをとったものとして表示して、両者を比較しよう。

【問２】

分散および共分散の関数をｎで割るものに変更して、それぞれ bunsan2 と kyobun2 としてやり、それを呼び出す形で分散と共分散を表示してやろう。その際、共分散行列の対角成分が分散に一致していることを確認しよう。

【問３】

分散の計算で n-1 で割るものの方が、ｎで割るものよりも若干大きくなることをまず確認しよう。いくつかの統計学または計量経済学の教科書を見て、なぜ n-1 で割るものとｎで割るものがあるのかを調べて、他の人に説明できるようにしよう。