0.4 ジュニア版　　関数プログラミング

前の章では、GAUSS の中での関数の使い方の大まかなことがわかったと思います。この章では、そこからさらに進んで、自分で簡単な関数を作成してみることをします。コマンドを中心としたクリックポンのパッケージソフト違って、GAUSS ではこの関数を自作することがその中心に位置するところです。

ｆｎ命令で関数を定義しよう

もう一度、fn で定義する簡単な関数定義のしかたをおさらいします。

プログラム【インプットが１つの場合】

```
new;
cls;
x=seqa(1,1,10);
fn f(x)=x^3;
fn g(x)=2*x;
y=f(x);
z=g(x);
print y~z;
```

画面表示

```
   1.0000000       2.0000000
   8.0000000       4.0000000
   27.000000       6.0000000
   64.000000       8.0000000
   125.00000       10.000000
   216.00000       12.000000
   343.00000       14.000000
   512.00000       16.000000
   729.00000       18.000000
   1000.0000       20.000000
```

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　スクリーンをクリアする

３行目　変数ｘに１から１ステップで 10 個のシークエンスを設定する（列ベクトル）

４行目　関数定義 fn で $f(x)=x^3$ を定義する

５行目　関数定義 fn で g(x)=2x を定義する

６行目　変数ｙに f(x)を入れる

７行目　変数ｚに g(x)を入れる
８行目　変数ｙと変数ｚの内容（ここでは両者列ベクトル）を「～」で水平方向にマージした上、まとめて画面表示する

プログラム【インプットが複数個ある場合】

```
new;
cls;
r=0.01;
A=100;
n={1,2,5,10,15,20,30};
print "              year              simple            compound";;
print/rz n~simple(r,A,n)~compound(r,A,n);
/* These are placed in memory just as normal built-in fuctions. */
fn simple(r,A,n)=(1+r*n)*A;
fn compound(r,A,n)=((1+r)^n)*A;
```

画面表示

year	simple	compound
1	101	101
2	102	102.01
5	105	105.10101
10	110	110.46221
15	115	116.0969
20	120	122.019
30	130	134.78489

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　スクリーンをクリアする
３行目　変数ｒに 0.01 を入れる（利子率１％）
４行目　変数Ａに 100 を入れる（金額 100）単位はドルでも円でも万円でも何でも同じ
５行目　変数ｎに 1,2,5,10,15,20,30 の列ベクトルを入れる（年）
６行目　引用符”　“の中のメッセージを画面表示する
７行目　関数 simple(r,A,n)と compound(r,A,n)の計算結果を、一度ｙとかｚに代入せずに、直接画面表示する（「/rz」は「右寄せ」「小数点以下ゼロの省略」のオプション）
８行目　/*　*/の内側はプログラムで実行はされないメッセージ部分
９行目　関数定義 fn で単利 simple(r,A,n)=(1+r×n)×A を定義する
10 行目 関数定義 fn で複利 compound (r,A,n)=$(1+r)^n$×A を定義する

上のプログラムは、利子の計算で期間ごとに倍数で利子増えていく単利と指数乗に増えていく複利の２つの関数を作って、それをプログラムの前半で呼び出して、その中に数値の入ったインプット変数を与えて、画面表示させています。インプット変数は、simple(r,A,n)や compound(r,A,n)のように、関数名の後の丸括弧の中に個数分だけカンマで区切って代入する形式になります。これは f(x,y,x)などの数学の関数定義と同じです。

なお、関数定義の置く場所は一度どこであってもかまいません。呼び出して数値を代入する場所が関数が定義してあるところよりも前であっても何ら問題はありません。関数は一度作成してしまえば、そのプログラム中にある限りは、通常の GAUSS 標準装備の組込み関数と全く同じようにして使うことができます。むしろ、GAUSS のプログラムの作法としては、関数の類はプログラムの末尾にまとめて置かることが普通です。

より一般的な proc 命令で関数を定義しよう

１行で定義される簡単な関数は fn で定義すればそれで足ります。しかしながら、関数の中には、たくさんの内部過程や時にはたくさんのアウトプットリターンを持っている場合があります。その場合には、１行関数定義 fn では対応できません。その場合には proc 命令を使います。ここで、proc とは procedure の略称で GAUSS プログラムの中心となる機能です。

プログラム【インプットが１つの場合】

```
new;
cls;
x={78,60,92,100,48,66,81,70,55,88};
y=heikin(x);
print y;
print heikin(x);

proc heikin(x);
    local n,sum,xbar;
    n=rows(x);
    sum=sumc(x);
    xbar=sum/n;
    retp(xbar);
endp;
```

画面表示

```
   73.800000
   73.800000
```

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　スクリーンをクリアする
３行目　変数ｘに数値を列ベクトルで入れる
４行目　関数 heikin(x)を呼び出してｘの値を入れて、その答えを変数ｙに入れる
５行目　変数ｙの内容を画面表示させる
６行目　関数 heikin(x)を呼び出してその答えを直接画面表示させる（５,６行目と同じ）
８～14 行目 proc heikin(x)から endp が１つの procedure のかたまり。ｘはインプット。
　　関数名 heikin はすでに定義されている関数名や予約語以外なら何でもよい
９行目　その procedure の内部だけで使う変数（ローカル変数）を列挙する
10 行目　組込み関数 rows を呼び出してｘの列数を求め、その答えを変数ｎに入れる
11 行目　組込み関数 sumc でもってｘの列の和を求め、その答えを変数 sum に入れる
12 行目　変数 sum を変数 n で割った答えを変数 xbar に入れる
13 行目　この procedure のリターン変数（アウトプット）を retp(　);の中に書く
　　すなわち、ここでは変数 xbar がリターン変数である
14 行目　procedure の終りを明示するために endp;は必ず必要

すなわち、上の procedure の場合のこの関数に対するインプットとアウトプットの関係は

インプット変数		**関数**		**アウトプット変数（リターン）**
x	→	heikin(x)	→	xbar

関数内部のみで使われる変数

n,sum,xbar

というｘを与えて何かをするブラックボックスの名前が heikin であって、これはｆであるとかｇであるとかいう一般的な関数の名前であってもかまわない。ローカル命令で関数の中でだけ使いますよと「宣言された」変数 n,sum,xbar は、再定義されなければ procedure の外では無効になる（procedure の外側では何もその変数名に入っていない、すなわちローカル変数とはその関数の中で使われて計算が終わると消え去る変数である）。したがって、プログラム中６行目で y=heikin(x);とこの関数のリターン（計算結果）がｙに入っていますが、別に xbar である必要はない（xbar としてもかまわない）。なお、特に xbar=heikin(x);としないかぎり、プログラム中 procedure の外部では xbar には何も入っていないことになります。上のプログラムでは、関数で計算したものをｙに入れた後にｙを画面表示しても、関数そのものの計算結果（リターン）を画面表示しても同じことになることがわかると思います。必要に応じて適宜使い分けてください。

なお、分散の場合には、以下のように procedure を作ることによって、同じように呼び

出して使うことができます。

```
proc bunsan(x);
    local n,xbar,sigma2;
    n=rows(x);
    xbar=meanc(x)';
    sigma2=sumc((x-xbar)^2)/(n-1);
    retp(sigma2);
endp;
```

procedure 各行の説明

１行目　この procedure の名前は bunsan であって、そのインプット変数はｘ

２行目　この関数の中だけで使われるローカル変数は n,xbar,sigma2

３行目　組込み関数 rows でｘの列数を求め、その結果を変数ｎに入れる

４行目　組込み関数 meanc でｘの各行の平均を求め、列で出てくる結果を転置したうえで変数 xbar に入れる（この段階で xbar は１×ｋ（ｘの列数）の行ベクトル）

５行目　ｘの各列のそれぞれの値から xbar の各列の値を引いた「推定平均からの乖離」を２乗したものを各列ごとに組込み関数 sumc ですべて足し合わせたものを、この場合平均を推定したもので引いているので各列の値の個数ｎから１引いた n-1 で割ったものをちらばりの尺度としての分散として、変数 sigma2 に入れている（この段階で xbar は引かれるｘの行数につられて自動的に同じ行ベクトルがｎ個あるｎ×ｋの行列に変わる）これは GAUSS 独特のものである後の章で詳述する

６行目　この関数のアウトプット（リターン）として sigma2 を関数の外に返している

７行目　この procedure の終りを示す

これをどう使うかは読者に手にゆだねましょう。前の章で扱ったような stdc(x)^2 で計算した結果を比べてみましょう。同じになるはずです。GAUSS の組込み関数は n-1 で割るバージョンの分散で標準偏差を計算していますが、これとは別にｎで割るバージョンを必要という人は、上の procedure の(n-1)の部分をｎに置き換えて使ってください。この関数は前の平均を求める heikin と同様、複数の列のデータ行列も比べられるようになっています。このように、GAUSS では最低限の関数が用意されていて、それをもとに様々な関数のバリエーションを自分でプログラムしていくのが GAUSS を扱う基本的なスタイルです。自分の意図する関数が GAUSS には装備されていなかったり、違った考え方の関数である場合には、自分で関数をプログラミングします。繰り返しになりますが、ここでは、

$$\text{平均}\bar{X}=\frac{\sum_{i=1}^{n}X_i}{n},\quad \text{分散}=\frac{\sum_{i=1}^{n}(X_i-\bar{X})^2}{n-1}$$

という変数Ｘ（列ベクトルまたはそのいくつか集まった行列）を、列ごとに考えて、その要素をすべて足し合わせた合計をその個数で割ったものを「平均」とし、その平均を使って、それぞれの要素からこの平均を引いた乖離をそれぞれ２乗して足し合わせた合計をその個数ｎから１を引いた n-1 で割ったものをちらばりの尺度の「分散」としています。

今度は、足し合わせるのではなくてかけ合わせる幾何平均を計算してみましょう。

プログラム

```
new;
cls;
x={78,60,92,100,48,66,81,70,55,88};
print heikin(x);
print gmeanc(x);

proc heikin(x);
    local n,sum,xbar;
    n=rows(x);
    sum=sumc(x);
    xbar=sum/n;
    retp(xbar);
endp;

proc gmeanc(x);     @ This procedure requires positive data. @
    local n,xbar;
    n=rows(x);
    xbar=prodc(x)^(1/n);
    retp(xbar);
endp;
```

画面表示

```
   73.800000
   71.998712
```

上のプログラムは、データの全ての要素が正である場合にのみ有効です。０が混じっていたり負の数が混じっていると、このままの計算方法では、その値の信憑性は全くありません。注意してください。結果の上の値が通常の平均、下の値が幾何平均です。このように通常は幾何平均の方が平均よりも小さな値になります（このことは後で取り上げる時系列データの移動平均をしてやるとより散らばりが小さくなる特徴を示します）。Procedure 部分だけの説明をしておくと、

プログラム各行の説明

１行目　proc 命令で gmeanc という名前の関数を作る（ここでインプットはｘ）
２行目　この関数内だけで使われるローカル変数はｎと xbar
３行目　組込み関数 rows でデータ行列ｘの列数を求めて、それを変数ｎに入れる
４行目　組込み関数 prodc でデータ行列ｘの列ごとの各要素の積を求めてその 1/n 乗をしたものを変数ｘbar に入れる
５行目　xbar をこの関数のアウトプット（リターン）として外部に返す
６行目　この procedure の終りを示す

このプログラムもこれまでと同様、系列の集まりの行列データに対応しています。ここでは１列のデータｘを与えています。「幾何平均」とは、「積のｎ乗根」で足し合わせてｎで割るのではなくて、掛け合わしてｎ乗根をとるというもので

$$幾何平均 = (X_1 \times X_2 \times ..... \times X_n)^{\frac{1}{n}}$$

ということです。プログラムでは定義そのままの計算ができます。ログをとって計算する方法は忘れてください。プログラムでは２乗であっても３乗であっても、また 0.4 乗であっても 1/10 乗あっても、すべて「^」の演算子で対応できます。このプログラムのメッセージは２つあります。１つは、heikin(x)という関数の中にも gmeanc(x)という関数の中にも xbar というローカル変数があります。それらは違った値を関数の内部で持っています。しかしながら、１つのプログラム上では何ら問題も混乱も生じません。なぜなら、それらは共にローカル変数であって、関数の外では消えてなくなるものだからです。アウトプットリターンとして外部に返していても同じことが言えます。全く問題はありません。２つ目は、プログラミングというのは、手計算や電卓計算と違って、定義やそのものの意味そのままをプログラムに表現できることです。対数表や統計テーブルを使うために人為的に公式に手を加える必要はないのです。

今度は、インプットが１個ではなくて、複数であるケースについてみましょう。
実際の経済データでは規模や人口または世帯数の異なる地域で得られたデータを平均しないといけないことがよくあります。そのような時には、上でやったような単純平均ではなくて「加重平均」が用いられます。データのうちその要素の出てくる場所や集団の大きさや規模に応じて、加重（ウエイト）をつけて平均するというものです。複雑に思われるかもしれませんが、これを列ごとにプログラムすると至って簡単なことがわかります。これまでと同様に関数部分だけを procedure で組み立てて示しておきましょう。

```
proc wmeanc(x,w);
    local wi,wxbar;
    wi=w/sumc(w);
```

```
    wxbar=sumc(wi.*x);
    retp(wxbar);
endp;
```

プログラム各行の説明

１行目　proc 命令で wmeanc という名前の関数を作ります（ここでインプットは行列ｘと同じ長さのウエイト列ベクトルw）

２行目　この関数の中で使われるローカル変数は wi と wxbar

３行目　それぞれのウエイトの値をウエイトの和で割ったもの（合計は１）を wi に入れる

４行目　合計が１になるように計算されたウエイト列ベクトルのそれぞれの要素をデータ行列Ｘとかけ合わせたものの和を計算して、それを wxbar に入れる

５行目　wxbar を関数のアウトプット（リターン）として外部に返している

６行目　この procedure の終りを示す

上の procedure 関数は、今までの組込み関数や自作関数と同じように複数の列からなるデータ系列を取り扱えます。実際に使うのは読者の手に委ねましょう。この関数をプログラムの末尾に置いて、通常の組込み関数と同じように呼び出してインプット変数を与えてやって画面表示させます。（関数は、ただプログラム中に置いても動きません。これまでの例と同じように、インプット変数を与えて結果を表示させなければ動きません。）ここで、上の heikin(x)と同じデータを使うのであれば、データの長さは 10 個で 10 個の要素からなるウエイト列ベクトルを設定する必要があります。例えば極端な例として、

```
w={1,2,3,4,5,6,7,8,9,10 };
```

を関数呼び出し部分よりも前に置いて、print wmeanc(x,w);とすれば、後ろに行けばいくほど大きなウエイトの加重平均が出来上がります。また、前章の７年分の経済データの場合

```
w={1,2,3,4,5,6,7};
```

とすれば同様に近年になればなるほど大きなウエイトの加重平均が列ごとに計算されます。その際には転置させて print wmeanc(data,w)';として表示させてみてください。このようにウエイト行列はデータ行列の列の長さ（行数）に等しい数の要素を持つことが必要です。そうでなければこの関数は動きません。また、ウエイトの要素をすべて１にすれば、その結果は平均の結果と全く同じになるはずです。それは、前にあげた平均の数式を１項１項ばらしてやると、ｎ分の１がそれぞれのデータの要素にかかっていて、それを合計したものが平均にほかならないからです。ウエイト行列の要素にすべて１を設定すると、まさにこの状態になります。ウエイトはこの関数内で合計されたものでそれぞれを割っていますから、ウエイト間の比が問題であって、絶対的な大きさは問題ではありません。上の場合１から７であっても 0.1 から 0.7 であっても結果は全く同じになります。このようにベクトルで計算するとすっきりとして簡単にプログラムできることがわかると思います。

インプットが複数あるものをもう１つ示しておきます。以下のプログラムでは２つのベクトルの内積と、２つのベクトルのおりなす cos θ を計算して両者の関係をみています。

プログラム【インプットが複数の場合】

```
new;
cls;
p={3,1,2};
x={1,6,3};
print "     inner product=" innerp(p,x);
print "         cos(theta)=" costheta(p,x);
print "|p||x|cos(theta)=" dist(p)*dist(x)*costheta(p,x);

fn innerp(a,b)=a'b;

fn dist(x)=sqrt(sumc(x^2));

proc costheta(a,b);
    local dista,distb,dista_b,costh;
    dista=dist(a);
    distb=dist(b);
    dista_b=dist(a-b);
    costh=(dista^2+distb^2-dista_b^2)/(2*dista*distb);
    retp(costh);
endp;
```

画面表示

```
   inner product=        15.000000
      cos(theta)=       0.59108280
|p||x|cos(theta)=        15.000000
```

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　スクリーンのクリアする

３行目　列ベクトルを変数ｐに入れる

４行目　列ベクトルを変数ｘに入れる

５行目　組込み関数 innnerp(a,b)を呼び出して、そのインプットとしてｐとｘをそれぞれ代入して出てきた結果を、引用符内のメッセージとともに、直接画面表示する

６行目　組込み関数 costheta(a,b)を呼び出して、そのインプットとしてｐとｘをそれぞれ

代入して出てきた結果を、引用符内のメッセージとともに、直接画面表示する

７行目　組込み関数 dist(x)と costheta(a,b)を呼び出して、そのインプットとしてｐとｘをぞれぞれ dist(p)*dist(x)*costheta(p,x)というふうに代入して３つの積を計算し出てきた結果を、引用符内のメッセージとともに、直接画面表示する

９行目　fn で a と b をインプットとする内積関数 innerp を定義する

11 行目 fn でｘをインプットとする距離関数 dist を定義する（距離は列ベクトルのそれぞれの要素を２乗したもののルートをとったもの）

13 行目　proc で a と b をインプットとする cos θ関数 costheta を作成する

14 行目　その関数内だけで使われるローカル変数は dist,distb,dista_b,costh の４つ

15 行目　関数 dist を呼び出して、その中に変数 a をインプットとして入れて計算された結果を変数 dista に入れる

16 行目　関数 dist を呼び出して、その中に変数 b をインプットとして入れて計算された結果を変数 distb に入れる

17 行目　関数 dist を呼び出して、その中に変数 a から変数 b を引いたものをインプットとして入れて計算された結果を変数 dista_b に入れる

18 行目　($dist^2$+$distb^2$-$dista\_b^2$)/(2*dista*distb)を計算した結果を変数 costh に入れる

19 行目　costh を関数のアウトプット（リターン）として外部に返している

20 行目　この procedure の終りを示す

上のプログラムはインプットが２つ（のベクトル）の場合を示していると同時に、多くの関数を作成しておいて、GAUSSの標準の組込み関数と全く同じように他の関数の中でも呼び出して使えるということを示しています。先に、fn命令でdistというベクトルの大きさを計算する関数を定義しておいて、それを関数costhetaの中で通常の組込み関数と同じように１関数として呼び出しています。なお、関数distとcosthetaの置く場所は前後していてもかまいません。一度プログラム上でfnまたはprocで関数を定義すると通常の組込み関数と区別なく使えるのです。上の計算は、高校生の時に習う余弦定理から、次のような

$$\cos\theta = \frac{|a|^2 + |b|^2 - |a-b|^2}{2|a||b|}$$ というベクトルと角度の関係が成り立ち、

内積 $< a,b >= |a||b|\cos\theta$

と２つのベクトルの内積が、ベクトルの大きさの積とcos θとの積に等しくなることを確かめています。この場合、内積<a,b>は正ですから、２つのベクトルのおりなす角度θは鋭角になります（負の場合は鈍角、０の場合は当然のことながら直角）。

関数の形式

インプットの数が１個または２個の場合の関数の形式を例とともにまとめておきます。

形式

```
fn f(x)= xからなる式;
```

例

```
fn f(x)=2*x^3;
```

形式

```
fn f(x,y)= xとyからなる式;
```

例

```
fn f(x,y)=x+3*y;
```

形式

```
proc f(x);
    local  ローカル変数;
    . . .
    式
    . . .
    retp(  リターン変数  または  式  );
endp;
```

例

```
proc f(x);
    local a,b;
    a=2*x;
    b=sin(a);
    retp(b);
endp;
```

代替例

```
proc f(x);
    retp( sin(2*x) );
endp;
```

代替例

```
fn f(x)=sin(2*x);
```

形式

```
proc f(x,y);
    local ローカル変数;
    . . .
    式
    . . .
    retp( リターン変数 または 式 );
endp;
```

例

```
proc f(x,y);
    local a,b,c;
    a=sin(x);
    b=cos(y);
    c=a+b;
    retp( c );
endp;
```

代替例

```
proc f(x,y);
    local a,b;
    a=sin(x);
    b=cos(y);
    retp( a+b );
endp;
```

上のようになります。インプット変数が３変数以上になっても「関数名( )」の中にカンマで区切ってそのインプット変数をその数だけ並べるだけです。少しややこしくなりますが、アウトプットリターンの数が１個ではない場合にはどうすればよいでしょう。１つの方法は、例えば、行と列の一致するようなアウトプットリターンが２個以上あれば、それらを水平方向に「～」でマージさせて、a~b~c などとすればよいでしょう。そうではなくて、独立にリターンとして関数の外に出すには、

```
proc(3)=f(x,y);
    local;
    a=
    b=
    c=
    retp(a,b,c);
endp;
```

というふうにリターンの retp( )の括弧の中にアウトプットのリターン変数をその個数分書き、その個数を冒頭で明記します。なお、print やグラフ専用の procedure 関数などリターンを必要としないものは、retp( );自体をなくすか、または retp;と括弧なしで同じ位置に書いた上で、冒頭で proc(0)=と表示します。これらは、後の章で様々な例の中で徐々に解説することにしましょう。