0.S2 ジュニア版 特別章(2) 積み上げ式 Procedure プログラミング

GAUSS に限らず、プログラミング方法には、そのプログラミング言語のアーキテクチャに反さないかぎり、それぞれの考え方がありまたそれに対応した方法があります。ここでは、コンパイルしないことを前提に、論文やレポート等の付属プログラムとして、誰にでも一目でプログラムの構造が理解できる判読可能でかつ単一ファイルにおさまるプログラミング法を提案します。また、私の著作のそのほとんどがこの体裁を採用しています。GAUSS には DOS の時代を経て発展してきていますので、一部研究者には、set ファイルを用いたり、複数ファイルを C ライクに include 命令でもってつなぎ合わせたりして実行させるものもあります。そうした方法も、ある意味で正統プログラミング法なのですが、ここでは（1）データファイルを除きプログラム部分は単一ファイル（2）procedure を細かく積み上げて内部組込み関数と同じ扱いをさせるようにさせる（3）速さや容量よりも容易に読むことができることを重んじる、そういったプログラミング方法を提案します。もちろん、この方法は Light 版においても共通に用いられるものです。

単一 Procedure の扱い方

これまでも学習してきましたように、procedure は１つのある複雑な関数を計算する関数部分です。基本的に Fortran などの subroutine とは扱いが多少異なりますので、ここではあえてそうは呼ばないことにします。２つのサイコロをｎ回振るプログラムをします。

プログラム

```
new;
cls;
print dice2(5);

proc dice2(n);
    local i,m,p1,p2;
    m=zeros(n,2);
    i=1;
    do while i<=n;
        p1=ceil(6*rndu(1,1));
        p2=ceil(6*rndu(1,1));
        m[i,.]=p1~p2;
        i=i+1;
    endo;
    retp(m);
endp;
```

画面表示

6.0000000	3.0000000
4.0000000	3.0000000
6.0000000	2.0000000
5.0000000	6.0000000
6.0000000	4.0000000

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　関数 dice2 の中にインプットとして５を代入して、その結果を画面表示させる。
４行目　単に区別をする改行。複数行改行してもかまわない。
５行目～１６行目　インプットがｎ、アウトプットがｍの dice2 という procedure の塊。
６行目　この procedure 内だけで使われるローカル変数は、i,m,p1,p2 の４つ。これらはたとえ procedure 外部で別に定義されていても影響を及ぼさない。
７行目　行列変数ｍをｎ行２列の零行列として確保しておく。以降行なうような index を用いた代入で何もない所に数字を代入することはできない。この作業は必須。
８行目　変数ｉの初期値に１を設定する。
９行目　ｉがｎ以下の間、endo までの部分の Do ループを続ける。送でなければ、endo の後に行く。
10 行目　組込み関数 rndu を用いて、一様乱数を１行１列分（つまり１個）発生させて、それを６倍したものを組込み関数 ceil を用いて小数点以下切り上げをして、それを変数 p1 に代入する。つまり、{１,２,３,４,５,６}の中から１つ引くこと。
11 行目　同様にして{１,２,３,４,５,６}の中から１つ引いて、結果を変数 p2 に代入する。
12 行目　変数 p1 と p2 を水平方向にマージしたものを行列変数 m のｉ行目に格納する。
13 行目　ｉを１つ増やして、それを新たにｉとして Do に戻る。
14 行目　Do ループの終わりの部分を表す。
15 行目　リターンとして、行列変数 m を外部に返す。
16 行目　procedure の終わりの部分を表す。GAUSS では、これは何時でも必要。

注意事項

１）GAUSS においてはべた書きをしない。見易いように字下げをする。
２）関数 Procedure は通常、下部に置かれる。改行によって独立させて見やすくする。
３）Procedure 内部では対応する動作の始めと終わりは同じレベルの字下げにする。

以上のプログラムは、後半の自作関数 procedure 部分と前半のそれを呼び出す部分の２つからなっています。すなわち、インプットｎが与えられているとき、１からｎ回まで２つの

乱数からサイコロの目を出し、それをいったん p1 と p2 に入れておいて、それらを水平方向に並べたものを、あらかじめｎ×２の零行列で確保されているｍのｉ行目に置くというものです。それを１からｎ行まで繰り返して、ｉがｎ以下でなくなれば Do～endo のループを脱出します。そういう意味で、do と endo のところは字下げがそろっていなければ見通しが悪くなります。最後に、ｎ行までサイコロの目が２列ずつ入った行列変数ｍをこの関数 procedure の外部に返しているわけです。この proc がら endp までが１つの自作関数であって、一度そのファイル上で作成されれば、そのファイル上にその関数があるかぎりは、通常の GAUSS 標準装備の組込み関数とまったく同等に使用できます。イメージとして、標準装備の組込み関数（ここでは rndu や ceil など）がさらにそのファイルの奥に存在すると言うか、あたかもその最下部で機能しているとでも想像してみてください。この自作関数 dice2 のインプットに５を入れたものを冒頭部分で使って、その結果を画面表示させているのです。

プログラム

```
new;
cls;
call dice2(5);

proc dice2(n);
    local i,m,p1,p2;
    m=zeros(n,2);
    i=1;
    do while i<=n;
        p1=ceil(6*rndu(1,1));
        p2=ceil(6*rndu(1,1));
        m[i,.]=p1~p2;
        i=i+1;
    endo;
    print/rz m;
    retp(m);
endp;
```

なお、上のように procedure 内部に print 文があってｍを出力をし、かつリターンとしてｍが存在する場合、冒頭では call 文を使うことによって、その関数だけを機能させてリターンを止めてしまうことも可能です。また、もともと **proc(0)=dice2(n);**と冒頭を改造し、retp 文をなくすことによって、リターンなしの procedure にすることも可能です。

複数の procedure の扱い方（その１　組込み関数と同等に）

まずは、自作の関数を組込み関数と同等に用いて、さらに別の procedure の中で使う例を見てみましょう。以下では、２つのサイコロをｎ回振って、その目の合計が sum 以下である確率を求めるシミュレーションをしてみます。

プログラム

```
new;
cls;
print "Prob=" simdice(5,8);

proc simdice(n,sum);
    local m,p;
    m=(sumc(dice2(n)').<=sum);
    p=sumc(m)/n;
    retp(p);
endp;

proc dice2(n);
    local i,m,p1,p2;
    m=zeros(n,2);
    i=1;
    do while i<=n;
        p1=ceil(6*rndu(1,1));
        p2=ceil(6*rndu(1,1));
        m[i,.]=p1~p2;
        i=i+1;
    endo;
    print/rz m;
    retp(m);
endp;
```

画面表示

```
Prob=
                3               3
                5               1
                6               6
                3               4
```

```
                1                3
        0.80000000
```

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　関数 simdice の中にインプットとしてｎと sum に対してそれぞれ５と８ををを代入して、その結果を、Prob=というコメントとともに、画面表示させる。

４行目　単に区別をする改行。複数行改行してもかまわない。

５行目～10 行目　インプットがｎと sum、アウトプットがｐの simdice という procedure の塊。

６行目　この procedure 内だけで使われるローカル変数は、m,p の２つ。これらは、たとえ procedure 外部で別に定義されていても影響を及ぼさない。実際、下の dice2 で違った意味で定義されて計算に使われている。

７行目　このファイル上にある自作関数 dice2(n)を用い、それを転置させたものの列ごとの和を組込み関数 sumc で求め(つまり行ごとの和を求め）それが要素ごとの比較で（ドットは要素ごとを表す）sum 以下であるものを１、そうでないものを０とする、合計ｎ×１の０と１の結果ベクトルを変数ｍに代入する。

８行目　変数ｍの合計を求め、それをｎで割ることにより、これを変数ｐに代入する。

９行目　この procedure のアウトプットとして p を外部に返す。

10 行目　この procedure の終わりの部分を表す。

(以降)　前述の dice2 の procedure と同じ。dice2 は組込み関数と同等に機能する。

上では、dice2 という下部にある procedure があたかも GAUSS 標準装備の組込み関数と同等に機能するかのごとく用いて、また別の simdice という procedure の中で用いられています。ここでは、最下部に置かれた dice2 という procedure のリターン m を、その上の simdice という別の procedure のインプットとする必要は必ずしもありません。そのファイル上に、そういう名前の自作関数があれば、その関数も組込み関数と同様に自由に使えるわけです。このようにすることによって、ここでは、２つのサイコロをｎ回振る関数と２つのサイコロをｎ回振って出た目の合計が sum 以下になる確率を求める関数との２つに作業を分割できるのです。こうした方がプログラムの読み手にとって、よりわかりやすい論理構造を示すことができます。我々人間は、ある一定の長さや論理展開をする他人のプログラムを理解するには相当の根気が必要です。これをそれぞれの作業を短くすることによって、誰にでも理解できるシンプルな論理展開に分割して、それを細かく積み上げていくのです。ただし、ここでは上部の simdice は下部の dice2 という procedure を計算するにあたって必要とします。GAUSS と言えども、バージョンを重ねるごとに新しい関数ができているわけで、このことはさほど問題とはならないと思われます。また、そのバージョンに最新バージョン

の関数がなければ、それに相当するものを自作すればよいわけです。
なお、画面結果では、実際のサイコロの目が表示されてしまっていますが、これは下部の dice2 という procedure の内部に prrint 文が含まれているからです。確率だけを表示したいのなら、この print 文のところを 1 行消去するか、またはこの procedure をこのまま残したいのならば、上部の simdice の procedure の dice2 を用いる前後を、

```
proc simdice(n,sum);
    local m,p;
    screen off;
        m=(sumc(dice2(n)').<=sum);
    screen on;
    p=sumc(m)/n;
    retp(p);
endp;
```

とすることによって、その部分の画面表示（スクリーン）を OFF にして止めることも可能です。ON にすればそこからまた通常通り画面表示されます。ただし、m には計算結果が入っているわけですから、その後の結果がなくなってしまうわけではありません。

注意事項

1）自作 procedure はファイル上のどこにおいても組込み関数と同等に使える。
2）自作 procedure を別の procedure で使う場合、必ずしもインプットとして扱わなくてもかまわない。
3）組込み関数は最深部（最下部）にあるかのごとく想像して、より原始的な自作関数を下部から上部へと積み上げていくプログラミング方法を推奨する。

複数の procedure の扱い方（その２　片方をもう片方のインプットとして扱う）

今度は、同じことをするプログラムを片方の procedure をもう片方のインプットとして扱う方法を見てみましょう。

プログラム

```
new;
cls;
print "Prob=" simdice(dice2(5),8);

proc simdice(m,sum);
    local x,p;
    x=(sumc(m').<=sum);
```

```
    p=sumc(x)/rows(x);
    retp(p);
endp;

proc dice2(n);
    local i,m,p1,p2;
    m=zeros(n,2);
    i=1;
    do while i<=n;
        p1=ceil(6*rndu(1,1));
        p2=ceil(6*rndu(1,1));
        m[i,.]=p1~p2;
        i=i+1;
    endo;
    print/rz m;
    retp(m);
endp;
```

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　関数 simdice の中にインプットとして m と sum に対してそれぞれ dice2(5)と 8 を代入して、その結果を、Prob=というコメントとともに、画面表示させる。インプット m の方は、関数 dice2(n)の入れ子になっている。

４行目　単に区別をする改行。複数行改行してもかまわない。

５行目～10 行目　インプットがｍと sum、アウトプットがｐの simdice という procedure

６行目　この procedure 内だけで使われるローカル変数は、x,p の２つ。

７行目　インプット m を転置させたものの列ごとの和を組込み関数 sumc で求め（つまり行ごとの和を求め）それが要素ごとの比較で（ドットは要素ごとを表す）sum 以下であるものを１、そうでないものを０とする、合計ｎ×１の０と１の結果ベクトルを変数 x に代入する。

８行目　変数 x の合計を求め、それを x の行数で割ることにより、これを変数ｐに代入。

９行目　この procedure のアウトプットとして p を外部に返す。

10 行目　この procedure の終わりの部分を表す。

(以降)　前述の dice2 の procedure と同じ。dice2 は組込み関数と同等に機能する。

注意事項

１）procedure を呼び出す際、丸括弧の中には別の関数も入れ子で書ける。

２）丸括弧の関数の入れ子が可能なのは呼び出し部分であって procedure の冒頭でない。

ほとんど同じ動作をするプログラムですが､プログラム言語によっては文法制約上または慣習上、この方法しか用いることのできないケースもありますから、そういう場合には、別の procedure のアウトプットリターンを､また別の procedure のインプットとして用いていくことを繰り返すことになります。GAUSS では、この方法に固執することは全く必要ありません。アルゴリズム上すっきりとした方、または、自分の説明しようとしていることにより近い論理構造の方法を選ぶべきでしょう。

複数の procedure の扱い方（その３　関数自体を別の procedure で最適化する方法）

　今度は、すこし違うことをやってみます。ある関数の最小値または最大値、あるいはその関数をゼロと置いた時の方程式の解を別の procedure で見つけ出す方法です。これには、呼び出す際に特別なやり方があります。

プログラム

```
new;
cls;
call gridsolve(&f,0,1,0.001);

proc gridsolve(&f,start,finish,step);
    local x,x0,f:proc;
    x=seqa(start,step,(finish-start)/step+1);
    x0=x[minindc(abs(f(x)))];
    print "Solution: x0=" x0;
    retp(x0);
endp;

fn f(x)=x^3+2*x-1;
```

画面表示

```
Solution: x0=          0.45300000
```

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　関数 gridsolve の中にインプットとして関数としての f、それに通常のインプット変数として、start に０を finish に１をそして step に 0.001 を入れ計算。

４行目　単に区別をする改行。複数行改行してもかまわない。

５行目～11 行目　インプットが関数ｆとそれに通常の変数 start,finish,step の計４つ。アウトプットが x0 の girdsolve という procedure の塊。

６行目　この procedure 内だけで使われるローカル変数は、x,x0 の２つに加えて、ｆという proc 分類の関数。

７行目　組込み関数 seqa を用いて、start から step ごとに(finish-start)/step+1 個の数列を作成しｘに代入する。

８行目　そのｘを関数ｆに入れた結果を組込み関数 abs で絶対値を取ったもの、さらに組込み関数 minindc で最小値を与えるインデックス番号（上から何番目かという行数）を得て、それを最終的に変数ｘの四角括弧の中に入れてｘのそのインデックス番号目の値を x0 に代入する。

９行目　変数 x0 を、Solution: x0=というコメントとともに、画面表示する。

10 行目　この procedure のリターンとして x0 を外部に返す。ただし、冒頭で call 文でこの関数を呼び出しているので、結果が２重になることはない。Print 文で呼び出すと、すでにこの関数の中に print 文があるので２重になる。

11 行目　この procedure の終わりの部分を表す。

12 行目　単に区別をする改行。複数行改行してもかまわない。

13 行目　1 行定義命令 fn を用いて関数 f(x)＝x^3+2*x-1 と定義する。これは、

```
proc f(x);
    retp(x^3+2*x-1);
endp;
```

としても同じことである。ｆｎは procedure の特殊形に分類される。

注意事項

１）関数の最大化、最小化、イコール０と置いた時の方程式の解を見つける際には、目的関数は&の印をつけてインプットとして呼び出す。

２）ローカル宣言では、目的関数には：proc とその属性を procedure であることを明記する。上の場合、：fn でもかまわない。

だたし、上の場合、始点０終点１の間をそのステップ 0.001 までの精度しか見ていない。この始点と終点の間に解（その関数の絶対値が０付近になる点）がなければこの計算は正しいとは言えない。その場合には、ファジー不等号の組込み関数 fne を用いて

```
_fcmptol=1e-2;
if fne(f(x0),0);
    errorlog "ERROR: Cannot find a solution.";
    retp(".");
```

endif;

などというプログラムを解を見つける procedure 内のリターンの前に置いてやると、その解がはずれている場合には、引用符内のエラーログを出すようにできる。上では、0.001 刻みの Grid 上の点だけを見たわけで、組込み関数(a,b)という a とｂが違う場合には１を、そうでない場合には０を 1e-12 の精度で返すものを使えばよいのだが、これでも精度は厳密すぎるので、刻みの１桁落ち程度の 1e-2 を（すなわち 0.01 までは確実に a の部分の x0 とｂの部分に入る０とは違うかどうか比べる）その関数のグローバル設定である**_fcmptol** に設定してやればうまくいくはずだ。この場合関数の値は、０までの距離を比べているので f(x0)でも abs(f(x0))でも同じことである。同様に、上のプログラムは絶対値の組込み関数を外すことによって、その区間での最小値を求める procedure になる。また、最小値を与えるインデックスを求める minindc を最大値のもの maxindc に変更すれば、その区間の最大値を求める procedure に簡単にできる。一応、上では、始点を０、終点を 0.9999 とすれば 0.0001 の１万点の高さを Light 版の制限内で比べられる。それ以上を比べるには、正式版を大容量メモリのもとで動作させるか、または、繰り返し計算をさせて、徐々に絞り込んでいくプログラムにすると方程式の解または最適点を求められる。少なくとも、２値を求めるケースまでは確実にこの方法で求められる。

練習問題

【問１】複数の procedure の扱い方のその１のところで、２つのサイコロを 100 回振りその目の合計が 10「以上」になる確率を求めたい。そのプログラムを改造し答えを求めてみよう。

【問２】最後の関数自体を別の procedure で最適化する方法で $f(x)=-x^2+0.5$ について０と１の間の f(x)＝0 とする時の解を求めてみよう。