3.6 プログラミング エラー処理と proc の作法　　ver. 0.3

エラー処理は、GAUSSのようなプログラム系の言語では重要です。表計算系またはワークシート連動型のソフトでは、たとえデータが不完全であってもデータが完全なところだけを計算します。しかしながら、行列を計算することを繰り返して目的とするプログラムを実行させるGAUSSでは、エラー処理の概念が重要になってきます。Ｉｆ分岐のところの最初で少し取り上げたように、プログラムが完全に動くという前提の上でも、インプットするパラメーターやデータの値が考えられる変域外の場合を想定してエラー処理のアルゴリズムを条件分岐で書いてやる必要があります。特に、公開したり他の人が後で別のことに一部利用したりするであろうことを前提としたprocedureを作るにあたっては、自分の使うパラメーターやデータの予期しない変域や種類の変数をも想定してエラー処理アルゴリズムを追加する必要がでてきます。χ^2分布の乱数を発生させるprocedure rndx2では、その自由度dfを設定するパラメーターは常にプラスの整数であるはずです。df=1.5とかdf=0とかいうものは実際には存在しません。そこで下のようなエラー処理を書き加えます。

プログラム

```
proc rndx2(r,df,seed);
    local x,x2,ss;
    if df<=0 or x-floor(x)/=0;
        errorlog “Df must be a positive integer.”;
        end;
    endif;
    rndseed seed;
    x=rndn(r,df);
    x2=x.*x;
    ss=sumc(x2');
    retp(ss);
endp;
```

インプットのパラメーターのうちの１つであるdfが、「０以下であるか、または、小数部分が０でない」場合、エラーログ”Df must be a positive integer.”と画面表示して、そこで計算をやめます。ここで、floorはその丸括弧の中の引数を－∞方向に小さくして整数にするものです。その数が正の小数であれば、小数点以下切り捨てと同じことをします。英語で底にくっつけるという意味です。（なお、その反対の＋∞方向に大きくして整数にする組込み関数はceilです。英語で天井にへばりつけるという意味です。また、四捨五入の組込み関数はroundです。）そのfloorした値とそのものの値の差をとってやって、同じでなければ、その値は小数ですからこのＩｆ分岐にひっかかります。そうでなくて、同じ値になるということは、整数そのものですから、この場合１枝分岐ですから直接endif文以降に抜けて通常

の計算をしていきます。Ｉｆ分岐でひっかかったものには、errorlogというエラー文専用のprint文と同じ役割をする命令で引用符のなかのメッセージを画面表示させます。ここのerrorlogはprintと書いてやっても同じことです。例えばこのprocの前にプログラムとして呼び出し部分を下のように書いてやって実行してみます。

```
new; cls;
print rndx2(100,1.1,1000);
```

この場合、df=1.1となって現実にはありえない数ですからエラーメッセージを

```
Df must be a positive integer.
Currently active call: rndx2 [7]
```

というふうに画面表示して、止まった地点のproc名を表示してくれます。後半部分のメッセージはGAUSSが自動的に作成するものです。[ ]括弧の中の７というのは、プログラムにおける行数のことで、ここでは上から７行目のrndx2というproc内で止まったということを意味しています。プログラムの実質的な長さによってこの数字は変化します。厄介なことには、太字の部分のエラー処理部分がなければGAUSSは勝手にdf=1.1に対して計算をしてしまいます。自ら直接パラメーターを代入する場合はエラー処理は特に必須ではありませんが、パラメーターを受け渡しする場合には、何かのミスやプログラム上の構造から小数になったり非正の整数になることもあるわけで、このような処理はたいへん重要です。

ポイント　round(x)　　四捨五入した値を返す
ceil(x)　　＋∞方向に一番近い整数にした数を返す
floor(x)　　－∞方向に一番近い整数にした数を返す
trunc(x)　　小数部分をなくした数を返す

例　round(1.6)→2,　ceil(1.6)→2,　floor(1.6)→1,　trunc(1.6)=1,
round(1)→1,　ceil(1)→1,　floor(1)→1,　trunc(1)=1,
round(-1.6)→**-2**,　ceil(-1.6)→**-1**,　fllor(-1.6)→**-2**,　trunc(-1.6)=**-1**

ポイント　エラー処理の方法(その１)

```
if 条件文;
    errorlog "   任意のエラーメッセージ ";
    end;
endif;
```

ここで、エラー処理の際の重要なテクニックがあります。それは、上のプログラムではエラーの際の分岐の先でend;で実行をそこで止めてしまいました。しかしながら、依然としてGAUSSが自動的に出すエラーメッセージが次の行に出てきてしまっています。これを防止して、なおかつ、正常にproc部分の実行を終えて、リターンを返すようにプログラムす

ることが可能です。よく使われる方法としては、下のようにリターンに０を直接入れて、エラーの際にはスケラーの０を返すというものです。

プログラム

```
new; cls;
print rndx2(10,2.1,1000);

proc rndx2(r,df,seed);
    local x,x2,ss;
    if df<=0 or x-floor(x)/=0;
        errorlog “Df must be a poitive integer.”;
        retp(0);
    else;
    rndseed seed;
    x=rndn(r,df);
    x2=x.*x;
    ss=sumc(x2');
    retp(ss);
    endif;
endp;
```

画面結果

```
Df must be a poitive integer.
0.00000000
```

上のように、retp文を２箇所作ってやって、エラーの際にはリターンに直接スケラー０が入るようにして、そうでない場合には、通常どおりの計算の結果がリターンに入るようにプログラムされることが常套手段です。論理のところでもふれましたように、０というのはGAUSSの論理ではFalseを表しますので、そのことに一致させているのです。つまり、エラーが発生したときに、リターンはFalseという意味の０を返させているのです。計算上０では不都合であれば、例えば、retp(“.”);としてやることによって、ミッシングバリューを表すドットを返すことも可能です。慣習上、０または－１を入れます。複数のリターンのケースには、例えばproc(4)に対応するものとしては、retp(0,0,0,0)などとすることができます。上の例では、もともとは１枝のIf～endifでありましたが、今度はend文を避けるために、２枝のIf～else～endifの形をとっています。なお、endif文以降は当然ながら、すでに、いづれの場合にもリターンを返しているわけで、endp文だけになるはずです。

ポイント　正常終了でprocを組み立てるには、エラーごとにretp(0)を作って、falseの

リターンにしてやる（複数リターンの場合、retp(0,0,..,0)となる）

errorlog文でエラーをprint文と同じように出力してやって、必要であれば、リターンにスケラー０または－１を入れてprocを出る方法が短いprocedureでは一般的ですが、もっと長いprocを書く場合にはエラーの種類も様々になってくるわけで、これとは別の方法でエラーを処理します。それは、goto文の行き先のラベルに数字やメッセージを添えて飛ぶという方法です。大きなprocで最後のgotoの行き先のラベル以降で一括エラーのメッセージ処理をします。規模が小さいので、少し例は悪いかもしれませんが、引き続き自前のrndx2のprocの中で２つ以上のエラー処理をprocが大規模だという前提でしてみます。

プロシジャー

```
proc rndx2(r,df,seed);
    local x,x2,ss,ret;
    if df<=0;
        goto errormsg(11);
    elseif df-floor(df)/=0;
        goto errormsg(12);
    else;
      rndseed seed;
      x=rndn(r,df);
      x2=x.*x;
      ss=sumc(x2');
      retp(ss);
      endif;
    errormsg:
      pop ret;
      if ret==11;
          errorlog "Df must be positive.";
      endif;
      if ret==12;
          errorlog "Df must be an integer.";
      endif;
      retp(0);
    endp;
```

上のprocedureでは、入力される関数のインプットが想定外のものである場合、エラーごとに異なるナンバーを持たせたラベルをgoto文で飛ばして、末尾にあるラベルの行き先以降の処理サブルーチンでそれぞれの番号に対応するエラーメッセージを画面表示させているも

のです。一見すると複雑ですが、この方がどういうときにエラーをエラー処理をしているのかを末尾でまとめて示せて、かえってすっきりした読みやすいプログラムになります。赤字のところの分岐で、該当するエラーがあれば、後半末尾の赤字のエラー処理に飛ばせて、その中で一括処理させています。以前少しふれたように、goto label文にはlabel:のところまでプログラムを飛ばせる役割があります。goto文を使えば強引にループもさせることも可能です。ここでは、goto文のもう１つの側面である、ラベル名に数字やメッセージを添えて飛ばすことができるという性質を使っています。上の例では、自由度dfが０または負の場合に、プログラム末尾ののerrormsg:というところ以降のブロックにプログラムが飛ばされていて、かつ、その際にそのラベル名errormsgという中に11という数字が添えられて飛ばされています。自由度dfが、（正の数で、かつ）小数である場合にも、同じerrormsg:というところに飛ばされていて、今度は12という数字をラベル名にともなって飛ばされています。それ以外の通常の場合には、計算のうえ、retp(ss)で乱数をリターンしてそこで終わっています。上の２つの範疇のうちいづれかに引っかかったものは、さらに特別にプログラム末尾のエラー処理のところを実行してretp(0)でFalseリターンを返してprocを終えています。そのerrormsg:以降のところでは（ラベル名は任意です。飛ばす地点と飛ばした先の地点の名前が同じであれば何でもかまいません）、ラベルに含まれる数字や文字をpopさせて取り出して、retという任意の変数に代入しています。その取り出されたものが（すなわちretの値が）11であれば、Df must be positive.というメッセージを表示させ、12であれば、Df must be an integer.というメッセージを表示させています。そしてその後にプログラムの流れはendif文以降にもどり、retp(0)でFalseリターンを返しています。赤字のエラー処理部は、エラーの場合にのみ実行されるブロックです。goto文は、ただ単にラベルの場所に飛ばす役割しかありませんが、もうすでにretp文が正常な計算の部分にあるので、その場合には、プログラムはそこまでを実行してリターンを返してprocから抜けます。（厳密には、エラー処理部分はgosub文で独立した部分として書くべきなのですが、retp文があればそこからプログラムは出てしまうことと、retp文自体がprocの中でのend文的役割も果たしているので、正常な場合のプログラムの流れはretp(ss)からproc外に出てそこで完結しています。）したがって、赤字のエラー処理の部分にはプログラムの流れは、エラーをおこさないかぎり来ません。エラー処理部分の２つの２枝分岐は、ここではend文が分岐の先についていませんから、プログラムの流れは、それぞれの分岐に行ったあとendifごとにもとの流れに戻ります。ここで、新しい概念popという命令が出てきますが、これは機械語などで使われるpopのような厳密なものではなくて、ただ単にラベル直後の丸括弧内に書かれてラベルとともに飛ばされた数字を取り出してやるというします。繰り返しますが、retのところはどんな変数名でもかまいません。取り出した数字をretという任意の変数に代入しているのです。

ラベル名( )のところには、数字を直接書けるほかに、引用符にくるんで直接メッセージを書くこともできます。（ただし、割当てメモリーの関係上、長い文章やあまりにもたくさんの数の使用は不可。）こうしてしまうとエラー処理を末尾で一括処理する必要がなくなって

しまうのですが、応用ということで改変したprocを下に示しておきます。その概念を理解してもらえると思います。

プロシジャー

```
proc rndx2(r,df,seed);
    local x,x2,ss,ret;
    if df<=0;
        goto errormsg("Df must be positive.");
    elseif df-floor(df)/=0;
        goto errormsg("Df must be an integer.");
    else;
        rndseed seed;
        x=rndn(r,df);
        x2=x.*x;
        ss=sumc(x2');
        retp(ss);
    endif;
    errormsg:
        pop ret;
        errorlog ret;
        retp(0);
endp;
```

上のprocは以前のものとまったく同じ動作をするものです。その違いは、数字ではなくて、エラーメッセセージそのものを含んだラベルをエラー処理部に飛ばして、そこでpopさせてメッセージを取り出しretという変数に代入して、その変数をerrorlog文でprint文と同じように画面表示させて、falseリターンを返しています。この場合も、エラー処理部は、errormsg:という行き先以降のブロックで、該当するエラーがあったときにのみこのブロックは実行されます。注意すべきことは、文字列は引用符” “で包んで丸括弧の中に書いて、それを飛ばすということです。数字の場合はそのまま、１なら１、２らな２と直接書きます。またgoto文の行き先を表すその行き先には、ラベル名の後に、セミコロン；のマークではなくて、コロン：のマークが必要です。**ラベル名：**の部分を他の行とともに一括して書いてもかまいません。例えば、errormsg:　pop ret;と書いても同じことです。このように、goto文のラベルは数字またはメッセージをともなって飛ばせるということを覚えておいてください。ここでは２つだけのエラーの範疇を考えましたが、５つ以上多数のエラーの範疇を想定する場合などには、エラー処理部を一括して末尾に書いた方が、その都度errorlog文で書いていくよりもプログラム構造上すっきりとします。

ポイント　goto ラベル名;　はそのラベル名の後ろの丸括弧の中に数字または引用符つきのメッセージを添えてあわせてサブルーチン部に飛ばすことができる

ポイント　pop 変数名;　　ラベルに添えられた数字またはメッセージを取り出して、変数名に代入格納する

エラー処理つきprocを組み立てるのによく使う組込み関数とその使用例を見てみます。まず、行列関係の数学上のprocを作る際には、そのインプット引数に使われる行列を実数と虚数に区別して、虚数の場合を扱えない場合にはエラーを発生させる必要があります。

プログラム

```
new; cls;
x={1 2 3, 4 5 6, 7 8 9};
print shuffle(x);

proc shuffle(x);
    local m;
    if hasimag(x);
        errorlog "Input matrix must be real.";
        end;
    else;
        x = real(x);
    endif;
    m=vecr(x)~rndn(rows(vecr(x)),1);
    m=sortc(m,2);
    m=m[.,1];
    print m;
    m=reshape(m,rows(x),cols(x));
    retp(m);
endp;
```

ローカル変数の宣言の後の太字のところが、インプットされる行列ｘが虚数である場合を考えた処理です。数学的な行列処理のprocを新たに考える場合、この処理は必要です。ここで、hasimagという組込み関数は見て字のごとく「虚数部分を持っている」という英語の略称です。これがTrueであれば１を返して、そこからelse文の前までの命令が実行されます。そして、ここではend文がありますから、endif文以下には行かずに、そのend文の場所で終わります。Falseであれば０を返して、else以降endif文以前の命令が実行されます。そして、endif文以降が実行されていきます。つまりここでは、行列ｘが虚数部分を含んでいれば、

errorlogのメッセージを表示して、end文で強制終了させています。そうでない場合には、通常は必要はないものですが、時として虚数部が空欄になったような特殊な行列またはスケラーが存在するので、そのことを想定して一応、realという行列の実数部だけを得る組込み関数で変換します。このように、hasimgとrealという組込み関数の処理ルーチンがついていた方がGAUSSのプログラムデザインに、より適合したものになります。

プログラムのアルゴリズムの本体は、ｘという行列の要素を乱数を用いてシャッフルした行列を求めています。行列ｘをもとに、それを組込み関数vecrで１列の列ベクトルに変換します。そして、それと同じ行数のStandard Normalにふるまう乱数ベクトルを水平方向にマージします。これをmとおきます。それを２列目、つまり乱数の値にもとづいて行ごと小さいものから大きいものへ並べ替えます。これで、２列目に付随して１列目もソートされます。それを、またmとおきます。mのうちの１列目を、またmとおきなおして、最後はこれをreshapeで元の行列の行数と列数に一致するようにして行列に戻しています。これによって、行列ｘが実数行列である場合、要素がシャッフルされます。

また、これとは別に行列が実際のデータである場合にはミッシングバリューであるドットを含んでいる場合も多々あります。エラーを表示するプログラムを上と同様にして、

```
if ismiss(x);
    errorlog "Data X has missing value(s).";
    end;
endif;
```

のルーチンを加えればよいでしょう。組込み関数ismissはその引数の行列に１つでもミッシングバリュー（ドット）があれば１を返し、そうでなければ０を返すものです。これは英語で「ミス（ミッシングバリュー）がある」という意味の略です。ｘがドットを含んでいるならば、このismiss(x)が１になって、if文はTrueになり、errorlog文とend文が実行されて強制終了します。ｘがドットを含んでいなければ、endif文以降が実行されていきます。動くかどうかは、上の虚数行列をはじくプログラムの太字部分と上のプログラムの一部とを差し替えて、行列にドットを加えて動くかどうか自分で確かめてください。

ミッシングバリューがあるというのは、経済の人たちはあまり気にしないようですが、カットしてしまったり０と置きかえたりするばかりが能ではありません。このことについては、フィルターの節でプログラムとともに説明することにして、ここでは、単純にミッシングバリューを含む行を水平方向にまとめて削除してしまう方法と、ドットをゼロに置き換える方法だけを示しておきましょう。とりあえずは実用にたえられるはずです。

ミッシングバリューがある行を行ごとに削除するにはpackrという組込み関数を使います。下のアルゴリズムではenif文以下に、この行ごと削除の作業の後に戻ってプログラムを実行していきます。メッセージは自分の好きな英語をつければよいでしょう。任意です。

```
if ismiss(x);
    errorlog "Delete the row(s) with missing value(s).";
```

```
        x=packr(x);
    endif;
```

ミッシングバリューのところの値を０に置き換えるのであれば、missrvというミッシングバリューのところを任意の数に置き換える組込み関数を使います。下の場合は、その第２要素に０が入っているように、０に置き換えます（この数は任意です）。

```
    if ismiss(x);
        errorlog "Each missing value is replaced by 0.";
        x=missrv(x,0);
    endif;
```

これを以前の行列のshuffleのプログラムに組み込むと次のようになります。

プログラム

```
new; cls;
x={1 2 3, 4 5 6, 7 . 9};
print shuffle(x);

proc shuffle(x);
    local m;
    if ismiss(x);
      errorlog "Each missing value is replaced by 0.";
      x=missrv(x,0);
    endif;
    m=vecr(x)~rndn(rows(vecr(x)),1);
    m=sortc(m,2);
    m=m[.,1];
    m=reshape(m,rows(x),cols(x));
    retp(m);
endp;
```

画面結果

```
Each missing value is replaced by 0.

      7.0000000       0.00000000       5.0000000
      9.0000000        6.0000000       4.0000000
      3.0000000        1.0000000       2.0000000
```

ポイント　packr(x)　　行列ｘの.の含まれる行を行ごと取り除いた行列を返す

missrv(x,0)　　行列ｘに含まれる.を０に置き換えた行列を返す

その他のエラーアルゴリズムで重要であると思われるものに、行列とスケラーの区別があります。これは、GAUSSではことわりのないかぎりすべての変数は行列として認識されるため、時としてそれが邪魔になってくることもあります。そんな場合には、たとえば、

```
if rows(x)/=1 or cols(x)/=1;
    errorlog "Error: Parameter must be a scalar";
    end;
endif;
```

などというものを加えれば、パラメーターｘの行と列の数が１でない場合、すなわち、ｘがスケラーでない場合には、エラーを発してそこで止まります。通常は汎用性を持たせるために、このような設定は必要とはなりませんが、GAUSSの変数のデフォルトが行列タイプであることを念頭においていれば、こういう設定も時には重要となってくることもあるでしょう。GAUSSにおいて変数は常に行列であることを意識していることは重要です。

Procedureの作法

まず、基本的なこととして２つのことがあります。１つ目は、グローバルにprocの外で設定された変数をprocedureの中でも利用できることです。もう１つは、procedureの中ではローカル変数は必ず直接または間接に定義されていなければならないことです。

次のような単純なプログラムを考えます。

プログラム

```
new; cls;
a=100;
print perc(0.25);

proc perc(x);
    local p;
    p=x*a;
    retp(p);
endp;
```

画面表示

```
   25.000000
```

この例は、小数で与えられた数値を100をかけてパーセンテージ形式に直すという極めて単純なことをするprocを考えています。a=100というところがprocのローカル変数として、procの内部にあればよいのですが、上のように、外部においてしまって一切ローカル宣言をしないことも可能です。ここでa=100という設定が、（新たに設定し直されるまで）ずうっとプログラムの最終行まで通っているということを意味します。たいへん便利な機能なの

ですが、これを多用すると他人には到底読むことのできないプログラムになってしまいます。また、後でproc部だけ切りとって動作を確認したり、他のプログラムに利用しようとしたりする妨げにになることも多々あります。オープンソース、そしてprocの相互利用の立場のGAUSSから言えば、グローバルに定義した変数をproc部が含んでいる場合には、コメント行でグローバルに設定された変数であることをことわっておく必要があるでしょう。極力、こうしたproc内での外部定義の数値の利用は避けるようこころがけるべきです。

上のプログラムを外部定義の変数をなくして正規のやりかたで書くと、

プログラム

```
new; cls;
print perc(0.25);

proc perc(x);
    local a,p;
    a=100;
    p=x*a;
    retp(p);
endp;
```

となります。内側に変数aを入れると、ローカル宣言をしないといけないことは言うまでもありません。それから、太字の２行では、aの方には直接100という数が入っています。また、pの方には間接的に関数の引数ｘとaをかけ合わせた数値が入る予定になっています。どちらも、なんらかの形で定義されて直接数値がはいっているか、間接に入る予定になっています。これ以外の方法で、ローカル宣言した変数をそのままに放っておくとエラーメッセージVariable not initializedと返されることになります。

これまでは、procedureは独立した関数であったのですが、procedureの中からprocやfnで定義された関数を呼ぶことも、プログラムの最中には直面することの１つであると思います。こんなときには、呼び出すprocやfnは、今プログラムしているprocedureのローカル宣言のところで、普通のローカル変数とは区別して、ローカルprocやローカルfnとして宣言してやる必要があります。以下は、その例です。

今、$f(x_1,x_2)=x_1^3+x_2^3-9x_1x_2+27$　という２値関数の極小値を求めます。Ifのところでプログラムしたニュートンステップを汎用性のあるprocedureに書きなおした上で、さらに、数値的にgradientとHessianを求めるのではなくて、あらかじめ手で計算したものを代入する形で極小値を求めます。

プログラム

```
new; cls;
fn f(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
fn g(x)=(3*x[1]^2-9*x[2])~(3*x[2]^2-9*x[1]);
```

```
    fn h(x)=(6*x[1])~(-9)|(-9)~(6*x[2]);
    beta0={10,10};
    betamin=min(&f,&g,&h,beta0);
    print betamin;
/*
** Minimization by simple Newton step with gradient and Hessian
** (C) Copyright 1999-2002 Yosuke Amijima. All Rights Reserved.
**
**    PROC MIN
**
**    FORMAT
**          beta = min(&f,&g,&h,beta)
**    INPUT
**          &f - pointer to a procedure(or a function) of the objective function
**               to be minimized.
**          &g - pointer to a procedure(or a function) of its gradient vector.
**               Notice that this is a row vector.
**          &h - pointer to a procedure(or a function) of its Hessian matrix.
**          beta - vector of start values
**
**    OUTPUT
**          beta - vector of parameters at minimum
**
** Remarks: You could change step size(step), maximum number of iterations
**(maxiter) and convergence tolerance(tol) and/or start values to get vector at
** minimum.This is a prototype procedure of minimum. You could upgrade it.
**
*/
proc min(&f,&g,&h,beta);
    local step, maxiter, tol, i, f:proc, g:proc, h:proc;
        step=1;
        maxiter=1e+3;
        tol=1e-5;
    i=1;
    do while i<=maxiter;
        print f(beta)~beta';
```

```
        if abs(g(beta)) < tol;
            break;
        endif;
        beta = beta - step*(g(beta)'/h(beta));
    i=i+1;
    endo;
    print; print /rz "# of iterations=" i-1;
    retp(beta);
endp;
```

画面表示

```
        -1127.0000          10.000000          10.000000
        -122.66456          5.8823529          5.8823529
        -9.7897612          3.9478879          3.9478879
       -0.31549144          3.1835238          3.1835238
  -0.00090256401          3.0100031          3.0100031
 -9.8803525e-009          3.0000331          3.0000331
  7.1054274e-015          3.0000000          3.0000000

# of iterations=         7.0000000

         3.0000000
         3.0000000
```

上のプログラムのアルゴリズムはIfのところで書いたプログラムと全く同じです。その違いは、gradientとHessianの手計算を直接代入して計算しているところにあります。まず注目してほしいのが、proc maxの引数の中で、f、g、hのそれぞれにポインターを表す&のマークがついていることです。これは、ｆ、ｇ、ｈといった変数を引数にするのではなくて、どこかで既に定義されている「関数」ｆ、「関数」ｇ、「関数」ｈの指すものという意味です。さらに、普通はprocedureの中では引数はローカル変数の宣言をする必要はないのですが、外部の関数の場合（組込み関数は除く）にはコロン：のしるしの後に、それがprocなのかfnなのか、それとも次章で説明するkeywordであるのかを書いて定義します。繰り返しますが、引数にある関数もローカル宣言でどういったタイプの関数であるか定義する必要があります。そして、ニュートンステップのステップ間隔を前と同じように１、最大iteration数を1e+3、そしてgradient計算の０になる有効桁数を1e-5と設定した上で、ステップ数を１とした時のニュートンステップ　ｂ＝ｂ－step*g/H;　で最小値を求めるアルゴリ

ズムを組み立てています。前回はgradientとHessianをgradpとhesspを用いて数値計算で求めたのに対して、今回は、既に計算されたgとhにベクトルbetaを直接代入して最小値になるようにiterationをしています。途中、関数値と、その時のbetaの値をベクトルを転置させて１行ごとになるように各iterationごとに表示させています。最後に、終えたiterationの最終的な数を表示して、最終的な最小値でのベクトルbetaをリターンとして返しています。目的関数をf(x)とおきます。そしてそのgradientとHessianをそれぞれ、g(x)とh(x)と1行関数の定義ｆｎで設定し、スタート値をbeta0としておいています。この場合は、２値関数なので、２×１のベクトルになります。そして、proc maxを呼び出してそのリターンをbetaminとおいたものを、print文で画面表示しています。

なお、f:proc, g:proc, h:proc　とあるところを試しに消してみてください。そうすると、それぞれの外部関数定義でリターンの行が一致しないためにプログラムは動きません。もっと単純なプログラムでは、ｆ、ｇ、ｈに相当する部分をグローバルな組込み関数と同じようにみなすことによって動くこともあります。この場合にはそれらの関数は、そのプログラム全体に共通するグローバルな変数と同じような働きをします。しかしながら、計算が複雑になると、関数宣言をローカル宣言のところであわせてしなければ、それぞれの関数の返すリターンに別のリターンも含まれることがあって、プログラムは動かなくなってしまいます。微妙なところです。たとえ引数であっても外部参照の関数は必ずタイプ宣言（:proc，:fn，:keyword）をする必要があります。なお、上のように、fnで定義したものをprocとローカル宣言することも可能です。一般的なprocedureにするためには、すべてをprocと宣言した方が、ほかのものもprocedureの仲間なので汎用性の高いプログラムになります。狭義のｆｎと宣言する必要はありません。

ポイント　proc **関数名(&外部関数名,変数名);**
　　local **外部関数名**:proc, **変数名;**
procedureの引数に外部関数が含まれるときには上のように設定する。
ただし、procはfnも兼ねる。関数は引数にくれば必ずローカル宣言する。

次に、proc(n)で扱ったlse(y,x)のprocedureと別の機能を持つprocedure間の引数の受け渡しを行なって、複数のprocedureを用いてより高度なプログラムを書くことをしてみます。OLSのプログラムを少し改造してできる簡単なRidge回帰を試みます。今、Ｄドライブに、datafile9.txtというファイル（Prof.Lin[2001]のlongley.txtを使用）があるものとします。lse(y,x)のprocedureの中味はそのままで、さらに３つの変数k、s2hat、varbを利用するためにリターンの数を３つ増して９にしてから、それらのリターンをridgeという新しく加えたprocedureに受け渡して、計算をさせます。

プログラム

```
new; cls;
```

```
load data[17,7] = d:datafile9.txt;
data = data[2:17,.];
y=data[.,7];
x=ones(rows(data),1)~data[.,1 2 3 5];

{b,se,t,shat,df,r2,k,s2hat,varb}=lse(y,x);
{c,br,ser}=ridge(x,y,k,s2hat,b,varb);

print "OLS regression:";
print "-----------------------------------------------------";
print "        Coefficient                                  Std Error";
print "-----------------------------------------------------";;
print b~se;
print;
print "c(Hoel,Kennard and Baldwin)=" c;
print "Ridge Regression:";
print "-----------------------------------";
print "        Coefficient            Std Error";
print "-----------------------------------";;
print br~ser;

proc(9)=lse(y,x);
    local b,uhat,n,k,df,s2hat,shat,varb,se,t,r2;
    b=inv(x'x)*x'y;
    uhat=y-x*b;
    n=rows(x);
    k=cols(x);
    df=n-k;
    s2hat=uhat'uhat/df;
    shat=sqrt(s2hat);
    varb=s2hat*inv(x'x);
    se=diag(sqrt(varb));
    t=b./se;
    r2=1-uhat'uhat/(y'(eye(n)-1/n*ones(n,1)*ones(n,1)')*y);
    retp(b,se,t,shat,df,r2,k,s2hat,varb);
endp;
```

```
    proc(3)=ridge(x,y,k,s2hat,b,varb);
        local c,br,w,vbr,ser;
        c=(k-1)*s2hat/(b'b);
        br = inv(x'x+c*eye(k))*x'y;
        w=inv(eye(k)+c*inv(x'x));
        vbr = w*varb*w';
        ser=sqrt(diag(vbr));
        retp(c,br,ser);
    endp;
```

画面表示

```
OLS regression:
------------------------------------------------------
     Coefficient                         Std Error
------------------------------------------------------
       1169087.5                          835902.44
      -576.46430                          433.48748
      -19.768071                          138.89276
     0.064393974                        0.019951886
    -0.010145254                         0.30856947

c(Hoel,Kennard and Baldwin)=  1.3033358e-006
Ridge Regression:
---------------------------------
     Coefficient        Std Error
---------------------------------
       383946.41         274523.27
      -169.31733         142.41137
      -74.271670         127.62128
     0.050485564        0.014228889
      0.10747194         0.28500166
```

{b,se,t,shat,df,r2,k,s2hat,varb}=lse(y,x); と {c,br,ser}=ridge(x,y,k,s2hat,b,varb); との間でいくつかのリターンの受け渡しをします。lse側のリターンのをｂとk,s2hat,varbを次のridge側に、ｘとｙとともに引数として渡します。だいぶん２つのprocedureに重なりがあって非効率なプログラムですが、ここはリターンと引数の受け渡しの練習と思って我慢して

ください。すなわち、

{b,se,t,shat,df,r2,k,s2hat,varb}=lse(y,x);

↓

上の関数のリターンのうち
b, k,s2hat,varbだけを利用

↓

{c,br,ser}=ridge(x,y,k,s2hat,b,varb);

とすることによって、１つのprocedureで計算させたリターンの一部を他のprocedureの引数として活用して、さらにそのprocedureで別の計算をさせているのです。なお、ｘとｙはもともとある変数です。これにより、ridgeというprocedureはRidge回帰のｃと係数brおよびそのおのおののStandard Errorであるserを計算して返します。これをprocedureの外部でわかりやすいようにprint文で整形して画面表示させています。ここで、簡易に、

$$Hoel = Kennard = Baldwin[1975] \quad c = \frac{(k-1)\hat{s}^2}{b'b}$$

を利用して、

$b_R=(X'X+cI)^{-1}X'y$　または $(I+c(X'X)^{-1})^{-1}b$

$Var(b_R)=(I+c(X'X)^{-1})^{-1}Var(b)(I+c(X'X)^{-1})^{-1}$ または w Var(b)w'

ここで$w=(I+c(X'X)^{-1})^{-1}$

となるRidge回帰の係数とそのStandard Errorを求めます。OLSのときにも注意しなければならない基本的なこととして、行列での計算では常に行列ｘの行数がｎ、列数がｋです。行列を使わない計算のときには、コンスタントを含まない独立変数の数がｋになっている書物もあるので注意してください。行列の計算では、常にdf=n-kです。df=n-k-1ではないので注意してください。本書では、行列ｘの列数をｋとして統一します。Ridge回帰とは、通常のOLSではMulticollinearityが出てしまうときに、縮小パラメータｃによって、それを補正してやる方法で、ｃの求め方にはいろいろなバージョンや考え方があります。しかしながら、それ以外のところは共通しています。ここで、Hoel=Kennard=Baldwin[1975]の簡便なｃの導出方法で補正したOLSを実行させています。プログラムの構造としては、２つのprocedureを末尾にならべています。まず、ファイルを呼び出し、１行目のラベルの部分を取り除き、７列目を従属変数ｙに、コンスタントタームの１の列ベクトルと1,2,3,5列目を従属変数ｘとして２つのprocedureを呼び出して、相互にリターンと引数を受け渡して、それらの結果をprint文で、OLS側はｂとseを、Ridge側はｃとbrそれにserを表示させています。プログラムを実際に書くときには、それぞれのprocedureのリターン数と括弧の中の数字が等しいこと、それから、呼び出した文の{　}括弧の中のリターンの変数の数と、もともとのprocedureのリターン数が一致することが必要です。リターンは必ずしもすべて使いきる必要はありません。また、呼び出すところで{　}括弧の中のリターン名前を若干変更しても構いませんが、リターンの順番と合計数は一致していなければなりません。上の

プログラムのridgeのprocedureでｘとｙの引数を、procedureのところと呼び出すところの２つのところで消去することも可能です。これは、ｘとｙがグローバルに通った変数であることを意味します。しかしながら、グローバルに通した変数を引数でもなくローカル変数でもなくprocedureの内部で使用することは、プログラムの全体の見通しを悪くするばかりではなく、あとでプログラムの一部を使おうとする人たちの思考の妨げになるので、できるだけそういったことは避けるようにしたいものです。

このほかにproc内では無闇にformat文を使わないことも重要です。Fomat文は非常に強力なコマンドで設定を変更するかGAUSSを終了するまでその設定は有効になりますから、procの内部でFormat文を使うことは極力避けましょう。どうしてもつかわないといけない場合には、外部で使うか、１行限りに有効なprintfmまたはprintfmtを使うようにします。