3.8 プログラミング 最適化アルゴリズム　　ver. 0.3

ここでは、プログラムのコアの部分になる最小化（またはマイナス関数であるときの最大化）のアルゴリズムの数学的な理論と、それとは全く異質な方法であって、最近最も注目されているグリッドサーチ法とそれにまつわる簡単なプロトタイププログラムについて説明します。

Newton-Raphson法（ニュートン法）

これは基本的に、エラー処理とprocの作法のところなどで示したgradientとHessianを用いて、その関数の表面が２次形式であると見たててgradientが0ベクトルに近くなるまで計算する方法です。再掲すると下のようなprocedureになります。

```
/*
** 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;
```

通常ニュートン法と呼ばれるのはこの方法です。なお、gradientとHessianの計算にGAUSSの数値近似のコマンドを使ってしまうと、厳密な意味でのニュートン法にはなりません。それらはあくまでも数値近似になります。数学的に説明すると、目的関数f(x)をx*の近傍で２次までのテイラー展開します。ベクトル形式表示で、

$$f(x)=f(x^*)+(x-x^*)'g(x^*)+1/2(x-x^*)'H(x-x^*)$$

ここで、xで微分してdf(x)/dx=0（これはg(x)そのもの）のときにx*の項はなくなって、Hのまわりの２つ分かれている(x-x*)の２乗の項を関数と見たてると1/2は２乗とキャンセルアウトして１になって、関数の中の微分も１になるから、

$$g(x)=g(x^*)+H(x-x^*)$$

ここで、g(x*)は関数ｇのx*における値であるが、H(x-x*)はHというHessianに(x-x*)をかけた形であることを区別して、さらに、f(x)が最小となる点でg(x*)=0なので、

$$g(x)=H(x-x^*)$$

となる。両辺にH^{-1}をかけて、

$$H^{-1}g(x)=x-x^*$$

最終的に移行して、(ただし、$H^{-1}g(x)$とg(x)/Hは同じこと)

$$x^*=x-H^{-1}g(x) \quad \text{または} \quad x_{k+1}=x_k-H_k^{-1}g(x_k)$$

となる。これをプログラムしてgradientベクトルｇ(x)があるConvergence Tolerance まで、例えば、0.001まで繰り返し計算したものがニュートン法と呼ばれるものです。GAUSSのライブラリーなどでは、ベクトルg(x)自身の値ではなくて、$g(x_k)-g(x_{k-1})$という相対変化をとって、これがConvergence Toleranceに至るまで繰り返し計算をさせているようです。ここで何をやっているのかイメージとして理解しておくべきことは、関数を２次元平面で単純化して考えると、HessianのHは関数の屈曲具合curvatureを示し、gは関数に対する接線の傾きを表します。ニュートン法は２次形式の計算を用いているので、もし目的関数が２次形式であらわされるならば、iterationは１回で終了することは言うまでもありません。実際には通常、目的関数（あるいは目的関数のマイナスの関数）が３次以上の複雑な形式であるので、２次形式であると前提したニュートン法の計算は１回では終了しないで、関数の屈曲具合に応じて（すなわちHessianに応じて)、最低点（あるいはマイナス関数であれば最高点）を行きすぎたり、あるいは、まだまだ到達しないで、何度もiterationを繰り返すことになります。純粋のニュートン法の場合には、上の最終式で表されるようにステップ幅は１になります（Hの前に１がかかっていると思ってください)。一般にステップ幅をHalfにするとgradientがおおよそ０になる収束までのiteration数は増加します。その反対に、２倍にするとiterationの数は減少します。ただし、このステップ幅をいつも１としていては、スタートベクトルから最終最適ベクトルまでの距離が（２次形式の意味で）遠ければ、かなりの回数、ときには永遠近く、このiterationは続いていくわけで、このステップ幅を１ではだめなら２倍で、それでもだめならさらに２倍に､... というぐあいに長くするようにGAUSSのMAXLIKやCMLなどのMaximum Likelihoodを計算するライブラリではなされます。その反対に、iterationが１では長すぎる場合には、２分の１、それでもだめなら、さらに２分の１にするようになります。このようにして、GAUSSのライブラリの内部アルゴリズムでは、最速で最適ベクトルに到達するように、ステップ幅を変化させて最低回数のiterationで最適ベクトルに到達するようにすることも可能です。純粋なニュートン法では、収束するスタートベクトルの範囲は限られていて、マイナス関数の最大値を求める場合、グローバルにconcaveではないLog-Likelihood関数の場合には、収束をもたらすスタートベクトルの範囲は限られており、非常に収束しにくいとされています。

準ニュートン法

上述のニュートン法のようにHessianがいつもいつも明示的に解けるとは限りません。また、繰り返し計算で使われるグローバルな範囲で、GAUSSの数値的に近似計算するコマンドでHessianがいつも求められるとは限りません。そこで、直接Hessianを求める代わりに、差分近似計算として、

DPF法(Davidon, Fletcher, Powell)

$$H_{k+1}^{-1} = H_k^{-1} + \frac{\Delta x_{k+1}\Delta x'_{k+1}}{\Delta x'_{K+1}\Delta g_{k+1}} - \frac{H_k^{-1}\Delta g_{k+1}\Delta g'_{k+1}H_k^{-1}}{\Delta g'_{K+1}H_k^{-1}\Delta g_{k+1}}$$

BFGS法(Broyden, Fletcher, Goldfarb, Shanno)

$$H_{k+1}^{-1} = H_k^{-1} + \frac{\Delta x_{k+1}\Delta g'_{k+1}H_k^{-1}}{\Delta x'_{k+1}\Delta g_{k+1}} - \frac{H_k^{-1}\Delta g_{k+1}\Delta x'_{k+1}}{\Delta x'_{k+1}\Delta g_{k+1}} + (1+\frac{\Delta g_{k+1}H_k^{-1}\Delta g_{k+1}}{\Delta x'_{k+1}\Delta g_{k+1}})\frac{\Delta x_{k+1}\Delta g'_{k+1}}{\Delta x'_{k+1}\Delta g_{k+1}}$$

のいずれかの方法によって、H^{-1}を求めます。$x^*=x-\lambda H^{-1}g(x)$のλが１のステップ幅だったのがニュートン法なのですが、今度は$x^*=x-\lambda H^{-1}g(x)$において、ラインサーチによりf(x)が最小になる（マイナス関数の場合には最大になる）ようなステップ幅λを求めます。これをHが任意の行列からgradientの変分がConvergence Toleranceよりも小さくなるまで繰り返して、最適ベクトルを見つけます。これらの方法では、HessianのHは直接には求められず再帰的に差分で近似していくことになります。経験的には、これらの準ニュートン法特に、後者のBFGS法は、最新のHessianの推定値を繰り返し用いているため、目的関数の関数形にもよりますが、一般に最もgradientが収束しやすい方法であると言われています。

SD法(Steepest DescentまたはAscent)

この方法は、H^{-1}の部分を計算しないでＩとして、

$x^*=x-\lambda g(x)$

を計算するもので、ステップサイズλを与えられたものとして、最もきつい傾きの増加のベクトルをもとめるものです。今度は２次形式ではなくて１次までの$f(x^*)+(x-x^*)'g(x^*)$の部分を、決められた距離の範囲内の制約、つまり$(x-x^*)$のスクエアディスタンスの下に最小化（または最大化）するものと言えます。

BHHH法(Berndt, Hall, Hall, Hausman)

この方法は、統計計量経済学の分野でのみ重点的に使われる方法で、Hessianに相当するものを求める際に、すべてのデータ行列を必要とします。WWFのレスラーHHH（トリプルエイチ）のような名前からも、計算的にかなりの力技であることがうかがい知れます。これは全データを用いた（gradientベクトルではなくて）N×Kのgradient行列のクロスプ

ロダクトK×K行列をHに用います。なお、ここで

$$-E\left[\frac{\partial^2 \ln L(\beta \mid data)}{\partial\beta\partial\beta'}\right] = E[G(\beta \mid data)'G(\beta \mid data)]$$ であることに注意して、

N→∞につれて、この前者はHになりますから、結果的に

G'(β | data)G(β | data) → H

K×N N×K K×K

(この場合マイナスはつかない)

になることを用いてiterationをgradientがConvergence Toleranceまで続けます。ただし、（後の節で述べるMAXLIKおよびCMLを列ベクトルの形で設定したLog-Likelihoodを用いるのは１つにこのためでもあります。）すなわち、Hの代わりにこのデータにもとづくgradient行列のクロスプロダクトをHに用いて、$x^*=x-\lambda H^{-1}g(x)$で計算するものです。この方法の特徴は、関数がその地点でconvexであろうとconcaveであろうと（目的関数をすでにマイナス関数にしてある場合）常にpositive definiteで、常に増加するという特徴をもっています。そういう意味で、concaveな地点を求めようとするのに、マイナス関数の場合に実はconvexな地点を求める方向に収束していく（またはその逆）という欠点があるニュートン法の代わりになるものなのです。ただし、最適点が余りにも近すぎる場合には収束に失敗する場合もあり得ます。この方法がなぜ計量経済学者だけに好まれて使われるのは、このgradient行列のクロスプロダクトが、Information Identityの考え方により、モデルが正しい時、Nが十分に無限大に近づく時、そしてTrueパラメータのときに、Hになるという極めて計量経済学的な性質を利用しているためです。（なお、$\Sigma_i(X_ie_i)(X_ie_i)'$の場合にはouter productになりますがここでの計算は行列形式でGAUSS計算上、ベクトル形式のLog-Likelihoodを微分したものを使っているのでouter productではなくてcross productになりますのでご注意を。いずれにしても、HessianはK×Kになります。）

とりあえず、プロトタイプのprocedureを示しておきます。このままでは、ほとんどの場合、収束しなかったり永遠に近いほどのループを繰り返すことになります。ステップ幅を手動で変更したり、ステップ幅をその都度最適化するプログラムを加えた上、途中でほかのアルゴリズムにスイッチするようにすれば収束しやすくなります。なぜならHessianの推定（すなわち屈曲具合の推定）そのものが、Nが小さい時には無理があることとパラメータの推定値の近傍以外の場所では屈曲面そのものがかなり違うということにあります。

```
/*
** BHHH method for optimization of vector-component Log-Likelihood
** (C) Copyright 2002 Yosuke Amijima. All Rights Reserved.
**
**     PROC BHHH
```

```
**
**    FORMAT
**          beta= bhhh(&ll,beta);
**    INPUT
**          &ll - pointer to a procedure of vector-component Log-Likelihood
**               function to be optimized.
**          beta - vector of start values
**    OUTPUT
**          beta - vector of parameters at optimum
*/
      proc bhhh(&ll,beta);
          local step,maxiter,tol,i,g,g_1,h,ll:proc;
          step=0.5; maxiter=1e+7; tol=1e-5; g=1;
          i=1;
          do while i<=maxiter;
              g_1=g;
              g=gradp(&ll,beta);
              h=g'g;
              if abs(g-g_1)<tol;
                  break;
              endif;
              beta=beta-step*inv(h)*sumc(g);
              i=i+1;
          endo;
          print; print /rz "# of iterations=" i-1;
          retp(beta);
          endp;
```

原理的には、ニュートン法と同じなのですが、GAUSSでは微分の結果は列方向に出てくることに注意をして、HessianのHの導出にその都度、データをともなって微分した行列のクロスプロダクトg'gを用います。また、gradientにはLog-Likelihoodのベクトル成分で表した微分の列の合計sumc(g)を採用します。中心となる式は、

```
h=g'g;
beta=beta-step*inv(h)*sumc(g);
```

となります。g－g_1では相対gradientの計算をしていて、その初期値にはスケラー１を代

入してあります。２ループ目では28列３行のGAUSSの計算結果のgradient－スケラー１で28列３行の１の要素ばかりの行列で引く計算をしてくれます。BHHHの完全なプログラムにするのには、最低限、ステップ幅を自動調節させるルーチンを加えることが必要ですが、BHHHの原理を知るのにはこれで十分でしょう。Dドライブにdatafile1.txtというｙとｘのシリーズのデータがあって、そのうち28行分のデータを使うものとします。-1/2*ln(2*pi)-1/2*ln(s^2)-1/2*e^2/s^2を最大化する関数として、スタート値を{0,0,1}として３つのパラメータを推定することにします。（実際には、これは通常のLog-Likelihoodを最大化するパラメータを求めているのですが、くわしいことについては、第４節の4.4 MAXLIK(2)をご覧ください。同じ計算をさせています。）なお、ニュートン法とそれに類するルーチンは最小値（または極小値）を求めるアルゴリズムであって、厳密には、上のLog-Likelihoodの全体にマイナスをつけた形を最小化するべきです。ここでもベクトル形式のLog-Likelihoodの全体にマイナスをつけた形を、proc ll(b);からendp;までのブロックで、28行１列のベクトル形式のLog-Likelihoodとしてリターンするものとします。また、その前のセクションでは、データを28行２列に加工して呼び出すものとします。

プログラム（収束にかなりの時間を要する）

```
new; cls;
load data[29,2]=d:datafile1.txt;
data=data[2:29,.];
start={0,0,1};
b=bhhh(&ll,start); print b';

proc ll(b);
    local beta, s, y, x, e;
    y=data[.,1];
    x=ones(28,1)~data[.,2];
    beta=b[1:2];
    s=b[3];
    e=y-x*beta;
    retp( -( -1/2*ln(2*pi)-1/2*ln(s^2)-1/2*e^2/s^2 ) );
endp;
/*
** BHHH method for optimization of vector-component Log-Likelihood
** (C) Copyright 2002 Yosuke Amijima. All Rights Reserved.
**
**     PROC BHHH
**
```

```
**    FORMAT
**          beta= bhhh(&ll,beta);
**    INPUT
**          &ll - pointer to a procedure of vector-component Log-Likelihood
**               function to be optimized.
**          beta - vector of start values
**    OUTPUT
**          beta - vector of parameters at optimum
*/
      proc bhhh(&ll,beta);
          local step,maxiter,tol,i,g,g_1,h,ll:proc;
          step=0.5; maxiter=1e+7; tol=1e-5; g=1;
          i=1;
          do while i<=maxiter;
              g_1=g;
              g=gradp(&ll,beta);
              h=g'g;
              if abs(g-g_1)<tol;
                  break;
              endif;
              beta=beta-step*inv(h)*sumc(g);
              i=i+1;
          endo;
          print; print /rz "# of iterations=" i-1;
      retp(beta);
      endp;
```

画面表示

```
iterations=          407428
          77.797758          52.009116          50.426568
```

繰り返しになりますが、BHHH法だけは他の最大値の計算と異なり、データを必要とする計算です。上の例の場合、最大化する関数はN×１のベクトル、gradientはN×３の行列になります。関数形とスタート値だけからは求めることはできませんのでご注意ください。

このほかに、準ニュートン法の変種にBroyden法というのがあるほか、Steepest Descent法の改良版にPRCG法(Polak-Ribiere-type Conjugate Gradient)というのもあります。またH^{-1}に相当する部分がそもそもpositive definiteであることを仮定しないでiterationを繰り

返す方法に、Greenstadt法やQuadratic-Hill Climbing法などがあります。

Grid Search法

最後に、微分に全く頼らないグリッドサーチについて説明しなければなりません。２変数のインプットまでのケースはプログラムもシンプルで、必ず収束して最大値を探し出します。近年その存在意義があらためてみなおされ、多変数の場合について最適化のオプションに直接または間接に組み込まれるようになってきています。

グリッドサーチのプログラムを作る前に、まず、そのコアとなるプログラムについて説明しましょう。使う命令は、maxcとmaxindcの２つです。いま、４×４のグリッドを考えて、その中に適当に数字が入っているとします。そのなかから最大値を求めます。それから、最大値を与えてくれた座標（すなわち行番号と列番号）を求めることにします。

プログラム

```
new; cls;
let grid[4,4]=
 1 5 8 4
 2 7 9 3
 3 6 1 6
 0 4 2 1
 ;
print maxc(grid); print;
print maxc(maxc(grid)); print;

print "row=" maxindc(maxc(grid'));
print "col=" maxindc(maxc(grid ));
```

画面表示

```
       3.0000000
       7.0000000
       9.0000000
       6.0000000

       9.0000000

row=       2.0000000
col=       3.0000000
```

上のように４×４の行列データをgridという変数にまず代入してあります。通常どおり、

grid={1 5 8 4, 2 7 9 3, 3 6 1 6, 0 4 2 1};としても同じことです。データをカンマなしにペーストしてやる方法では、必ず変数にディメンションをつけてletで定義して、最後にはセミコロンをつけてやらないといけません。さて、maxc命令で各列の最大値を求めて列ベクトルで返します。数学系のソフトから移ってきた人にはなぜ列ベクトルなんかで返すような面倒なことをするのかと思われるかもしれません。これには理由があります。統計計量の数値は列ごとのシリーズデータになっていて、それのなかの比較をして最大とか最小とかを求めるわけです。さらに、求まった最大値の列ベクトルの中の最大値をmaxcという列の最大値を求める命令にもう１度かけてやれば、最終的にデータを転置させることなくその行列のなかの最大値を求めることができるのです。列で返される最大の理由はここにあります。すなわち、maxc(maxc(gird))と２重に列の最大値を求める命令をしてやれば行列全体の最大値が１つもとまることになります。

1	5	8	4
2	7	9	3
3	6	1	6
0	4	2	1

上の行列では、実際に、１列目の最大値は３、２列目は７、３列目は９、４列目は６ですから、リターンは3,7,9,6の列ベクトルになっています。さらにもう一度maxcをしてやると、この中の最大値は９ですからスケラーの９が返されることになります。次に、この９がいったい行列のどこから来たかを求めるには、maxindcという命令を使います。これは、列ベクトルのなかの最大値を与える行数を返す関数で、上の一番最初の答えでmaxcで出てきた列ベクトルの中で最大のもの９は３列目にありますから、maxindc(maxc(grid))は３行目の３となります。これはちょうど最大値のある列番号を答えていることになります。行番号は、もともとのグリッドを転置させてやれば列と行がひっくり返りますから、maxindc(maxc(grid')')で最大値を与える行番号がわかることになります。基本的にこの２つの命令を駆使してグリッドサーチを繰り返すことになります。

ポイント　maxc(maxc(**行列**))　**行列のなかの最大値を求める**
ポイント　maxindc(maxc(**行列**')')　**行列のなかの最大値を与える座標の行番号を求める**
maxindc(maxc(**行列**))　**列番号を求める**

プログラムを作るにあたっては、ループのところでやってきたように、いままでどおり、ｉとｊの２つでまわして配列を作ります。グリッドの配列は常にｉ行ｊ列になるようにします。そのgrid[i,j]という座標に実際の座標(x,y)を対応させます。それぞれのグリッドサーチに対して、さらに細かいグリッドサーチにするために、最大iteration数までｋ回まわし

て段々とグリッドを小さくしていくことになります。それぞれのグリッドサーチにおいても仮想のグリッドgrid[i,j]という座標と実際の座標(x,y)の扱いは一貫して同じ変数を使ってｘとｙの動く範囲を段々と１段階ずつせまくしていきます。

ここでは、以前と同様に、f(ｘ,y)＝ｘ^3＋y^3－9ｘy＋27　という２値関数の極小値を求めます。目的関数は－f(ｘ,y)を採用して、これを(0,0)から(5,5)までの四角にLight版GAUSSの性能限界いっぱいの100×100の格子点をつけて、その点ごとの高さをｆ（または－ｆ）とします。列ごとに最大値ｆをもとめて列行列にし、その中からさらに最大値を求めることによって、そのグリッドの最大値を求めます。さらに、そこでの最大値を与えてくれる座標のまわりを小さくしぼって再び100×100の格子点をつけて最大値を求めます。これを例えば100回繰り返してやります。

プログラム

```
new; cls;
fn f(x,y)=-(x^3+y^3-9*x*y+27);
start={0,0}; finish={5,5};
b=gsearch(&f,start,finish); print "(x,y)=" b; print "f(x,y)=" f(b[1],b[2]);
/*
** Maximization by grid search method
** (C) Copyright 2002 Yosuke Amijima. All Rights Reserved.
**
**    PROC GSEARCH
**
**    FORMAT
**          beta=gsearch(&f,start,finish)
**    INPUT
**          &f - pointer to a procedure(or a function) of the objective function
**               to be maximized. The objective function is defined as
**               fn f(x,y)=... or use proc.
**          start - vector of start values for the gird of the first time
**          finish - vector of end values for the grid of the first time
**
**    OUTPUT
**          beta – 2 x 1 vector of parameters at maximum
*/
proc gsearch(&f,start,finish);
    local iter,length,xstart,ystart,xstep,ystep,x,y,xmax,ymax,xend,yend,grid,k,i,j,f:proc;
```

```
    iter=100; length=99;
    xstart=start[1]; ystart=start[2]; xend=finish[1]; yend=finish[2];
    k=1;
    do while k<=iter;
    print /rz "# of iterations=" k;;
    xstep=(xend-xstart)/length; ystep=(yend-ystart)/length;
    grid=(-1e256).*ones(length+1,length+1);
        x=xstart;
        i=1;
        do while i<=length+1;
            y=ystart;
            j=1;
            do while j<=length+1;
                grid[i,j]=f(x,y);
                j=j+1;
                y=y+ystep;
            endo;
            i=i+1;
            x=x+xstep;
        endo;
        xmax=xstart+(maxindc(maxc(grid'))-1)*xstep;
        ymax=ystart+(maxindc(maxc(grid ))-1)*ystep;
        print "            f=" f(xmax,ymax);
        xstart=xmax-xstep; xend=xmax+xstep;
        ystart=ymax-ystep; yend=ymax+ystep;
        k=k+1;
    endo;
    retp(xmax|ymax);
endp;
```

画面表示

```
# of iterations=          1          f=    -0.0036566048
# of iterations=          2          f= -8.4316913e-007
# of iterations=          3          f= -4.6840398e-010
# of iterations=          4          f= -4.2632564e-014
# of iterations=          5          f=  7.1054274e-015
```

```
# of iterations=                6          f=       0.00000000
# of iterations=                7          f=   7.1054274e-015
# of iterations=                8          f=       0.00000000
   .
   .
   .
# of iterations=               98          f=       0.00000000
# of iterations=               99          f=       0.00000000
# of iterations=              100          f=       0.00000000
(x,y)=
        3.0000000
        3.0000000
f(x,y)=        0.00000000
```

上のプログラムのなかでグリッドサーチのそれぞれのサーチでコアとなる部分は、

```
grid=(-1e256).*ones(length+1,length+1);
        x=xstart;
        i=1;
        do while i<=length+1;
            y=ystart;
            j=1;
            do while j<=length+1;
                grid[i,j]=f(x,y);
                j=j+1;
                y=y+ystep;
            endo;
            i=i+1;
            x=x+xstep;
        endo;
```

グリッドの長さを99に決めてやると始点も含めてグリッドの縦横の格子は100×100になります。（もちろんグリッドの長さを最初から100にしてやってもかまわないのですが、Light版のユーザーのことを考えていっぱいいっぱいの99を採用しています。）それぞれの格子点を初期化するために、-1e256を代入しています。これは任意の小さな値でかまいません。慣習上-1e256にしています。ループのプログラムでやったようにディメンションを作成するのにはｉとｊを採用しています。これにｉのループには実際のｘ座標の値も合わせてまわして、さらにｊのループには実際のｙ座標の値をつけてまわしています。このステージ

での最大値を与えてくれるｘ座標の値とｙ座標の値は、maxindcを使った文で、

xmax=xstart+(maxindc(maxc(grid'))-1)*xstep;

ymax=ystart+(maxindc(maxc(grid))-1)*ystep;

によって求まります。仮想のグリッド座標には(0,0)というところは存在しなくて、(1,1)から始まりますから、１を引いたものにステップ幅をかけたものが最初の起点からの実際の距離になりますから、起点のxstartにこの距離を足し合わせています。

次のステージのより細かいグリッドサーチに移行するには、

xstart=xmax-xstep; xend=xmax+xstep;

ystart=ymax-ystep; yend=ymax+ystep;

とすることによって、そのステージで最高点を与えてくれる座標からステップサイズだけそれぞれ差し引いたものを新しい起点xstartに、またステップサイズを足し合わせたものを新しい終点xendにしています。そして、iterationのｋを１から１つ増してk=k+1;として、do while k<=iter;のところにもどって、第２ステージのグリッドサーチを、これらの新しいより狭い範囲の起点と終点をもとに同じように行ないます。これをここでは100回繰りかえしているのです。そういうprocedureを

b=gsearch(&f,start,finish);

として関数ｆを設定して、startベクトルとfinishベクトルを与えることによって呼び出して計算させています。出てきたｂと、それを分解したb[1]とb[2]をもとにf(b[1],b[2])を計算して合わせて出力させています。なお、procedureのなかで、fnまたはprocをインプットの１つとして呼び出す場合には、:procとしてその変数をprocのローカル変数宣言をしなければならないことは、このほかの最適化のprocedureでも行なったのと同じです。

なお、上のprocedureの中でf(x,y)のところをf(x|y)とすれば、２×１のベクトルをインプットにする関数の最大化になります。また、maxindcとmaxcの組込み関数を、それぞれ、最小値を求めるのに対応する組込み関数minindcおよびmincとすれば、上のprocedureは最小値を求めるものに簡単に変更できます。（もちろん、&fの関数のリターンにマイナスサインをつけることによって、最大値を求めるprocedureで最小値を求めることもできます。）

以下では、簡単な実用例として、２×１のベクトルをインプットにするような、今度は最小値を考えるproceureに変更してみます。また、Toleranceレベルを設定して、途中でiterationを打ちきるようにし、なおかつ、初期値をもとの初期値と得られた最適値を与えるベクトルのプラスマイナス0.1,0.01,0.001,0.0001をそれぞれ考えて都合５重のiterationをしてやって精度を高めるものを考えます。

プログラム

```
/*
** Minimization by 5 times grid search method
** (C) Copyright 2002 Yosuke Amijima. All Rights Reserved.
**
```

```
**    PROC G5SEARCH
**
**    FORMAT
**          beta=g5search(&f,start,finish)
**    INPUT
**          &f - pointer to a procedure(or a function) of the objective function
**               to be minimized. The objective function is defined as
**               fn f(x|y)=... or use proc.
**          start – 2 x 1 vector of start values for the gird of the first time
**          finish – 2 x 1 vector of end values for the grid of the first time
**
**    OUTPUT
**          beta – 2 x 1 vector of parameters at minimum
**
**    NOTICE
**          Default tolerance level is tol=1e-10.
**          Each grid search has the initial window of +-0.5,0.1,0.01,0.001
**          of the optimum solution vector in its previous round.
*/
proc g5search(&g,start,finish);
    local b,e,g:proc;
    print "1st Round";
    b=gsearch(&g,start,finish);
    e=0.5;
    start=(b[1]-e*b[1])|(b[2]-e*b[2]); finish=(b[1]+e*b[1])|(b[2]+e*b[2]);
    print "2nd Round";
    b=gsearch(&g,start,finish);
    e=0.1;
    start=(b[1]-e*b[1])|(b[2]-e*b[2]); finish=(b[1]+e*b[1])|(b[2]+e*b[2]);
    print "3rd Round";
    b=gsearch(&g,start,finish);
    e=0.01;
    start=(b[1]-e*b[1])|(b[2]-e*b[2]); finish=(b[1]+e*b[1])|(b[2]+e*b[2]);
    print "4th Round";
    b=gsearch(&g,start,finish);
    e=0.001;
```

```
    start=(b[1]-e*b[1])|(b[2]-e*b[2]); finish=(b[1]+e*b[1])|(b[2]+e*b[2]);
    print "5th Round";
    b=gsearch(&g,start,finish);
    retp(b);
endp;

proc gsearch(&f,start,finish);
    local iter,length,xstart,ystart,xstep,ystep,x,y,xmin,ymin,xend,yend,grid,k,i,j,f:proc;
    local fmin,temp,tol;
    iter=100; length=99; tol=1e-10;
    xstart=start[1]; ystart=start[2]; xend=finish[1]; yend=finish[2];
    fmin=0;
    k=1;
    do while k<=iter;
    print /rz "# of iterations=" k;;
    xstep=(xend-xstart)/length; ystep=(yend-ystart)/length;
    grid=(-1e256).*ones(length+1,length+1);
        x=xstart;
        i=1;
        do while i<=length+1;
            y=ystart;
            j=1;
            do while j<=length+1;
                grid[i,j]=f(x|y);
                j=j+1;
                y=y+ystep;
            endo;
            i=i+1;
            x=x+xstep;
        endo;
        xmin=xstart+(minindc(minc(grid'))-1)*xstep;
        ymin=ystart+(minindc(minc(grid ))-1)*ystep;
        temp=f(xmin|ymin);
        if abs(fmin-temp)<tol;
            break;
        endif;
```

```
        fmin=temp;
        print "            f=" fmin;
        xstart=xmin-xstep; xend=xmin+xstep;
        ystart=ymin-ystep; yend=ymin+ystep;
        k=k+1;
    endo;
    print "            f=" fmin;
    retp(xmin|ymin);
endp;
```

使い方はgsearchと全く同じなので省略する(使用例は５節のBootstrap GMMを参照)が、２変数の推定をする際、特にGMMのように最適値の近傍がglobalにconcaveではないような複雑なものに有効である。上のプログラムは２つのprocedureから成り立っており、下部のものは、前述のgsearchをf(x|y)の形に直し、最小点バージョンにするために、minindcおよびmincを用いたもので基本的には同じである。それに、許容桁数tolを例えば1e-10に設定しておき、前の回の最小値fminと今回の最小値tempの差の絶対値が許容桁数を表す小数より小さい時にiterationをやめてループからbreakで出る部分を付け加えている。次に、上部のprocedureは、異なる初期ベクトルstartとfinishに対して下部のgsearchを５回呼び出している。最初の呼び出しにはもともとのstartとfinishのベクトルを使い、次のラウンドからはその前のラウンドで得られた最小値を与えるベクトルの前後0.5だけをstartとfinishに用いこの作業をそれぞれのRoundの解の前後0.1,0.01,0.01だけについてアップデートしていくことにより、前のラウンドに捉えられなかったような微細な点を求められるように、原始的な方法だが改良を加えたものである。なお、ローカル宣言では下部のローカル宣言と同様に使用する関数名を特別に：procと指定してやることを忘れてはいけない。