4.2 特殊関数とライブラリ 最適値問題(2) ver. 0.1

制約条件なしの最小化(実際には極小化)問題では、QNewton という組込み関数を使います。使い方は、eqSolve の設定にきわめて似ています。今、次のような２変数の関数の極小を求めます。

$h(x_1,x_2)=x_1^3+x_2^3-9x_1x_2+27$

まず最初に、qnewtonset;で QNewton のグローバル変数の設定を初期化します。関数定義のｆｎで２変数の関数を定義します。その際インデックスにはｘ[1]とｘ[2]を使います。そして、スタートベクトル、ここでは２変数ですから、２×１のベクトルｘ0 を設定してから組込み関数 QNewton で、その引数で、関数 h をそれを指し示すポインターを表す&のマークをつけて呼び出して、スタートベクトルとあわせて、代入して関数 h が最小となるベクトル(x_1,x_2)を計算します。（なお、qnewtonset;と QNewton の命令は、call で呼び出しても、そのまま命令としておいても結果は同じになります。ここでは、一貫性を保つため graphset;などと同様に、qnewtonset;はそのまま命令を置く形をとることにします。ただし、QNewton を call なしにそのまま書くと、余分なリターンが羅列されて、同じ答えやベクトル、リターンコードが２回違った形で現れることになります。）

プログラム

```
new; cls;
qnewtonset;
fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
x0={0,0};
call QNewton(&h,x0);
```

画面表示

```
return code =    0
normal convergence

Value of objective function       27.000000

Parameters    Estimates    Gradient
----------------------------------------
P01               0.0000        0.0000
P02               0.0000        0.0000

Number of iterations         1
Minutes to convergence     0.00000
```

上のように、リターンコードが０になっていれば、一応、計算は成功したと言えます。こ

の場合、$(x_1,x_2)=(0,0)$のとき、$h(x_1,x_2)=27$ となるという意味です。ただし、上を見て明らかなように、iteration が１回きりです。ただ、gradient ベクトルが最初から零ベクトルですから、gradient ベクトルを零ベクトルにするように動きようがなくて、ここは極値にはちがいはないのですが、なにかおかしいということになります。そこで iterarion が1回きりであることを避けるためにスタートベクトルを(0,0)から少しずらしてやりましょう。たとえば(1,1)としてやりましょう（この設定は、任意でかまいません）。いま、iteration をすべて表示する設定に下のグローバル変数

```
_qn_PrintIters=0;
```

をデフォルトの０（各 iteration を表示しない）から、１の表示するに変更します。
プログラム

```
new; cls;
qnewtonset;
fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
x0={1,1};
_qn_PrintIters=1;
call QNewton(&h,x0);
```

画面表示

```
step length         1.0000000
--------------------------------------------------
 iter 1          function =     16.18399988
--------------------------------------------------
       parameter      direction        gradient       relative gradient
             1.3       0.30000001       -6.6299998       0.53256302
             1.3       0.30000001       -6.6299998       0.53256302


 step length         1.0000000
--------------------------------------------------
 iter 2          function =     10.68797099
--------------------------------------------------
       parameter      direction        gradient       relative gradient
       1.7096639       0.40966386       -6.6181231        1.0586449
       1.7096639       0.40966386       -6.6181231        1.0586449
```

```
 step length      0.010000000
-------------------------------------------------
 iter 3          function =     10.81970396
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       3.9924521        228.27882        11.886953        4.3862651
       3.9924521        228.27882        11.886953        4.3862651


 step length        1.0000000
-------------------------------------------------
 iter 4          function =      1.80854329
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       2.5260763       -1.4663758       -3.5915028        5.0164185
       2.5260763       -1.4663758       -3.5915025        5.0164181


 step length        1.0000000
-------------------------------------------------
 iter 5          function =      0.15604856
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       2.8663229       0.34024666        -1.149484        3.2947924
       2.8663229       0.34024664        -1.149485        3.2947952


 step length        1.0000000
-------------------------------------------------
 iter 6          function =      0.00634819
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       3.0264807       0.16015775         0.2404291       0.72765403
       3.0264808       0.16015785        0.24043144       0.72766113
```

```
 step length        1.0000000
-------------------------------------------------
 iter 7          function =      0.00001347
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       2.9987763     -0.027704358     -0.011007955      0.033010394
       2.9987763     -0.027704483     -0.011008903      0.033013236


 step length        1.0000000
-------------------------------------------------
 iter 8          function =      0.00000000
-------------------------------------------------
       parameter      direction        gradient       relative gradient
       2.9999892     0.0012129207   -9.6634159e-005    0.00028990144
       2.9999893     0.0012129759   -9.6160462e-005    0.00028848035


 step length        1.0000000
-------------------------------------------------
 iter 9          function =      0.00000000
-------------------------------------------------
       parameter      direction        gradient       relative gradient
               3   1.0730012e-005    4.7369516e-007   1.4210855e-006
               3   1.0700453e-005   -4.7369517e-007   1.4210855e-006


return code =    0
normal convergence


Value of objective function        0.000000


Parameters    Estimates    Gradient
-----------------------------------------
```

P01 3.0000 0.0000

P02 3.0000 0.0000

Number of iterations 9

Minutes to convergence 0.00083

今回も、リターンコード０でnormal convergenceとなっていますが、今度はiterationの回数が９回と複数回まわって正常にgradientベクトルが徐々に絶対値で小さくなって、およそ０の値に収束しています。これは、$h(x_1,x_2)=x_1^3+x_2^3-9x_1x_2+27$が、(0,0)以外に(3,3)も極値になっていて、そのうち、(0,0)の方は鞍点、(3,3)が極小点になっているからです。したがって、(0,0)から少しでもはずれた地点からスタートすれば、(0,0)の鞍点ではなく、(3,3)の極小点に到達するようにiterationが行なわれるわけです。これは、

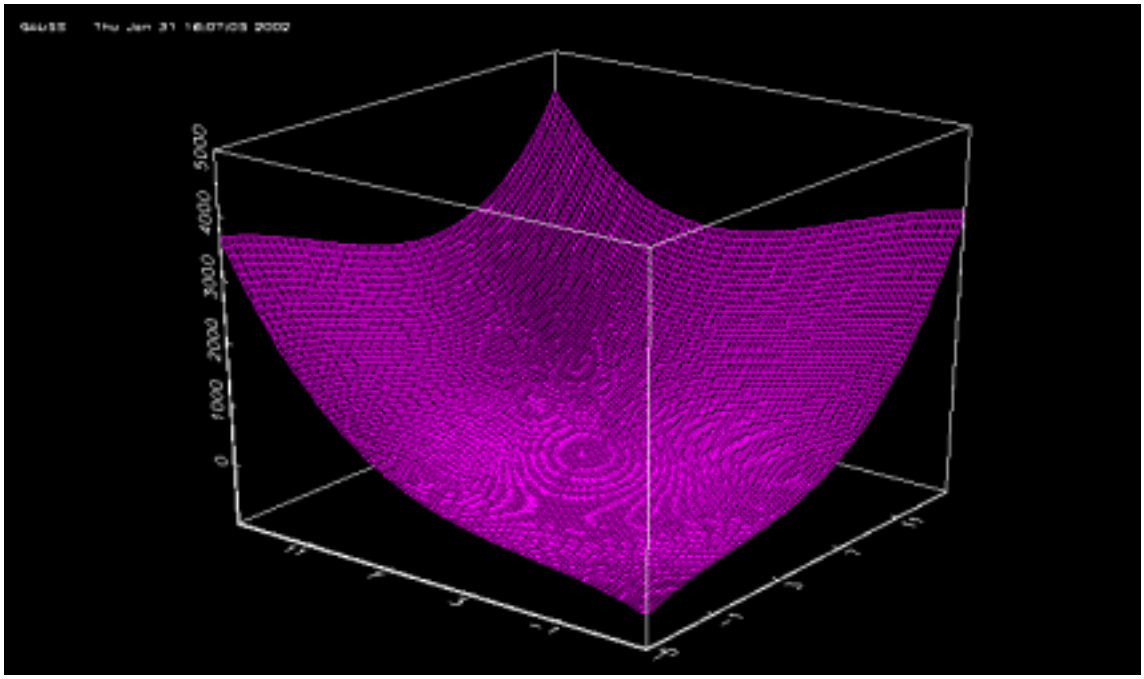
$$h_1=3x_1^2-9x_2,\quad h_2=3x_2^2-9x_1,\quad \begin{vmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{vmatrix}=\begin{vmatrix} 6x_1 & -9 \\ -9 & 6x_2 \end{vmatrix}=36x_1x_2-81$$

となりますから、(0,0)と (3,3)の両者とも、１階のh_1とh_2は０になっていますが、２階の方は、(0,0)では0、－81で零、負、(3,3)では18と243で正、正になっていて、(0,0)の方は鞍点、(3,3)が極小点となります。これは、グラフ表示からも明らかです。

プログラム

```
new; cls;
library pgraph;
graphset;
x1=seqa(-5,0.2,99)';   x2=seqa(-5,0.2,99);
x=x1+zeros(99,1);   y=x2+zeros(1,99);
z=x^3+y^3-9*x.*y+27*ones(99,99);
surface(x1,x2,z);
```

グラフ表示



プログラムにあるように、contourグラフなどを作るときには、第１節のグラフィックスのところでやりましたように、まずシークエンスを使ってｘ軸とｙ軸に相当するベクトルを作成したうえで、それらとベクトルの型が行と列が正反対になる零ベクトルを足し合わせることによって、（ｘ,ｙ）の座標のｘのところばかりの行列とｙのところばかりの行列ができます。これらを、要素対要素の計算でかけ合わせたり足し合わせたりします。べき乗＾はそれ自体が要素対要素計算ですからそのままでいいでしょう。かけ算は要素対要素のドット．をつけた演算になります。それから定数項は、今作成している行列空間と同じディメンションの同じ数が入った定数項になりますから、定数×すべての要素が１の行列になります。上の例でいえば、99×99のディメンションになるように計算しています。（なお、100×100まではライト版のGAUSSで扱えます。Contourグラフの方が奇数の座標を要求しますから、ここではライト版でも動作するようにグラフの座標計算の区切りの数に99を採用しています。）

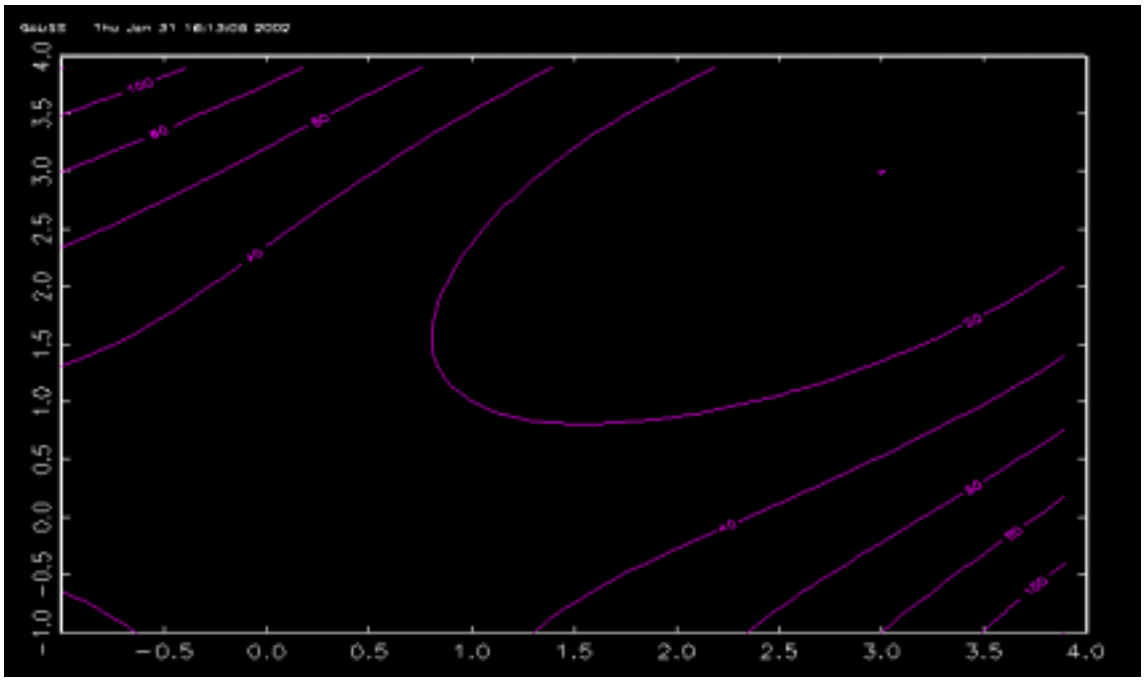
上のグラフのように、奥の高い(＋,＋)から一度くぼんでから、手前の(－,－)の方向に少し登ってから落ち込んでいます。(＋,－)方向と(＋,－)方向には左右対称に登っていて、谷になっています。さらに、ｘ軸とｙ軸ともに－１から４の座標に限って、さらに限られた領域のcontourグラフを描くと、谷の底の周辺の等高線の状況がわかります。

プログラム

```
new; cls;
library pgraph;
graphset;
x1=seqa(-1,0.05,99)';    x2=seqa(-1,0.05,99);
```

```
x=x1+zeros(99,1);   y=x2+zeros(1,99);
z=x^3+y^3-9*x.*y+27*ones(99,99);
contour(x1,x2,z);
```

グラフ表示



上のように、明らかに(3,3)が極小値になっていて、左下の(－,－)の方向にいけば行く下り坂になりますから、極小値から左下の方向に移動していくと、左下の(0,0)のあたりで一度登って、そのあと下りになります。その(0,0)のあたりでは、依然として周囲はゆるやかな谷になっています。ということで、(0,0)の周辺は鞍点になっていることがわかります。

本題のQNewtonに話を戻しますと、eqSolveのときのように、グローバル変数の設定として、手計算の１階の微分を計算に加えて精度を高めることもできます。今、１階の微分の２×１の列ベクトル（行ベクトルではないので注意）の関数をｊとして、これをグローバル変数_qn_GradProcに＆のポインターを表すしるしをつけて呼び出します。

プログラム

```
new; cls;
qnewtonset;
fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
fn j(x)=(3*x[1]^2-9*x[2])|(3*x[2]^2-9*x[1]);
x0={1,1};
_qn_GradProc=&j;
_qn_RelGradTol=1e-7;
QNewton(&h,x0);
```

画面表示

```
return code =    0
normal convergence

Value of objective function        0.000000

Parameters    Estimates    Gradient
-----------------------------------------
P01              3.0000        0.0000
P02              3.0000        0.0000

Number of iterations       10
Minutes to convergence     0.00650

        3.0000000
        3.0000000
 -1.4210855e-014

 -1.4566126e-013
 -1.4566126e-013
      0.00000000
```

この場合は結果はほぼ同じになりが、直接命令でべた出力させると、gradientがさらに０に近い値になることがわかります。これは、gradientのiterationの計算の有効桁数を設定するグローバル変数

```
_qn_RelGradTol=1e-5;
```

を変更して、1e-7まで有効桁数を下げているためです。-1.4566126e-013の小数点以下７桁ですから13－(７－１)＝７で、デフォルトの小数点以下５桁ではなくて、小数点以下７桁よりも小さくなるまでiterationが続けられます。手計算の１階の微分のベクトルを加えたので、さらに計算精度をここでは上げています。繰り返しますが、１階の微分の関数を設定する際に、１×２の行ベクトルではなくて２×１の列ベクトルであることに注意してください。そういう意味で、垂直方向にマージする｜の記号を使っています。

ただし、このQnewtonの方法にも問題があります。この関数は、上のグラフでもわかるように(－,－)の方向へはどんどん落ちていくような関数であって、QNewtonは極小値を求める関数であるため、もしスタートベクトルを(－,－)のどこかの点にとると永遠に下がり続

けて最小値を探しにいきます。今、プログラムを元のシンプルなものに戻して、iterationの数を設定するグローバル変数

```
_qn_MaxIters=1e+5;
```

だけを用いて、1e+5のデフォルト値から少し増やして1e+6にでもしてやりましょう。**なお途中でiterationをリアルタイムで画面表示させるのには「P」のキーを押してください。また、「C」のキーを押すとそこでiterationを打ち切ります。**計算は、マシンの速さにも依存しますが、かなりの時間を要します。Pのキーを押すと各iterationの画面表示に余計に時間がかっかって、さらに時間を要することになります。

プログラム

```
new; cls;
qnewtonset;
fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
x0={-1,-1};
_qn_MaxIters=1e+6;
call QNewton(&h,x0);
```

画面表示

```
return code =    2
maximum number of iterations exceeded

Value of objective function -10392736603.908516

Parameters    Estimates    Gradient
----------------------------------------
P01          -1730.5759   9000254.1573
P02          -1730.5763   9000258.2088

Number of iterations    1000000
Minutes to convergence     4.99733
```

上のように、最小値を探しに行ってしまって、どんどん(－,－)の小さな方向に行きます。そして、最終的に、iterationが1e+6回、すなわち1000000回行なったところで計算をやめてそこでの結果を出してきます。リターンコードは、正常収束の０ではなくて、iterationがその最大設定値を超えたというコードの２を返します。そして目的関数の値は、マイナスのかなり桁数のある数字になって、x_1とx_2に相当するP01とP02も、ほぼ同じ数で、マイ

ナスの小さな数になっています。その地点のgradientは、当然のことながら、０とは違う数になっています。この場合の計算のやり直しは、スタートベクトルを取りかえるしか方法はありません。多少、不便ではありますが、このスタートベクトルを取りかえるという方法は、GAUSSのほかの関数や主なライブラリ関数に共通するものです。特に極値が１つ以上ある場合や、変曲点や鞍点などをもった関数にGAUSSの関数をあてはめて計算させる場合には、いくつかのスタートベクトルを試してみる作業が必ず必要になります。下にリターンコードの番号別の意味をしめしておきますと、

リターンコード

0	**正常収束**
1	**強制終了**
2	**最大iteration回数超過**
3	**関数値計算失敗**
4	**gradient計算失敗**
5	**ステップの長さの計算失敗**
6	**初期値での関数が評価不能**

のようになります。リターンコードが２だからといって、最大iteration数を大きく変更しても問題の解決にはならないことは、上に示したとおりです。call形式で呼び出すほかに、

```
{ x,f,g,retcode } = QNewton(&h,x0)
```

として書いても、call形式の呼び出しと同じ結果になります。ただし、これにより４つのリターンを取り出すことができて、自由に加工したり、別の計算や関数にこれらのリターン変数を受け渡すことができるようになります。なお、ｘは解のベクトル、ｆはｘでの目的関数の最小値、ｇはgradientベクトル、recodeはリターンコードです。

これとは別に、ノンリニアのプログラミング、すなわち、非線型の制約条件つきの最小化(極小化)をする組込み関数として、sqpSolveが標準で用意されています。QNewtonで最初に用いて形状もすでにわかっている目的関数を使って、非線形の制約式や変数の変域を設定しなおして、あらためて計算しなおして見ましょう。まずは、下のように変更します。

$$\text{Min } h(x_1,x_2)=x_1^3+x_2^3-9x_1x_2+27$$
$$\text{s.t. } x_1=x_2$$
$$x_1^2+x_2^2<=8$$
$$x_1>=0,\ x_2>=0$$

プログラム

```
new; cls;
```

```
sqpSolveset;

fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
fn eqc(x)=x[1]-x[2];
fn ineqc(x)=-x[1]^2-x[2]^2+8;

_sqp_EqProc=&eqc;
_sqp_IneqProc=&ineqc;
_sqp_Bounds={ 0 1e256,
              0 1e256};

start={1,1};
call sqpSolve(&h,start);
```

画面表示

```
return code =    0
normal convergence

Value of objective function        7.000018

Parameters    Estimates
-----------------------------------------
P01               2.0000
P02               2.0000


Linear Equality Lagrangean Coefficients
               .

Linear Inequality Lagrangean Coefficients
               .

Nonlinear Equality Lagrangean Coefficients
  3.6870063e-011

Nonlinear Inequality Lagrangean Coefficients
       1.5000011
```

```
Bounds Lagrangean Coefficients

       0.00000000        0.00000000
       0.00000000        0.00000000

Number of iterations          6
Minutes to convergence      0.00000
```

上のように、ノンリニアの等号制約に、$x_1 - x_2 - 0$ を定義した任意の関数ineqcを設定しノンリニア不等号制約には、不等号関係をひっくり返すのに－１をかけた$-x_1^2 - x_2^2 + 8$を定義した任意の関数eqcを設定します。（不等号制約の書式の向きは＞＝の方向です。したがって、ひっくり返すために全体に－１をかけています。）上の設定のように、_sqp_EqProcにはノンリニアの等号制約の関数のポインターが、_sqp_IneqProcにはノンリニアの不等号制約(＞＝)の関数のポインターがきます。その名前を指ししめすものとして記号＆が関数名の前につけられて呼び出されています。変数のバウンダリーコンディションには_sqp_Boundsというグローバル変数が使われて、この場合、２つの変数とも０から無限大までですから、

```
_sqp_Bounds={ 0 1e256,
              0 1e256};
```

というぐあいに、１行目がx_1の変域を表し、２行目がx_2の変域を表すように設定します。なお、無限大には1e256、無限小には-1e256を使います。

$$0 <= x_1 <= 1e256$$
$$0 <= x_2 <= 1e256$$

デフォルトは、_sqp_Bounds={ 0　1e256 }となっていて、いくつ変数があっても同じ変域であれば１行でまとめて設定できます。このeqpSolveも擬似ライブラリ的にeqpSolveset;で、そのグローバル変数の設定値をクリアしてから計算をさせます。リニア制約でもノンリニアとして扱っても一向に差し支えはありません。（２次形式のプログラミングQProgと同様に、リニア制約の行列およびベクトルA,b,C,dを別に設定することもできます。）この関数の呼び方は他の最適値関係の関数と同じように、その引数に目的関数をポインターとして、スタートベクトルをともなって代入します。上の数値例の場合には、ルート８の半径の円内で、ふたつの変数の値が同じで非負の変域で、グラフに描いた目的関数を最小化(極小化)しています。もともとの極小点は(3,3)であったわけですが、今度は、円の内側である制約がきいてきて、円の制約上の(2,2)で極小になっています。このbinding、non-bindingの関係は、それぞれの制約のラグランジアンを見ることでわかります。この場合、ノンリニアの不等号制約のラグランジアンが約1.5で０ではないことが、この制約上で極小値が決まっていることを示しています。

なお、スタートベクトルを任意の(－,－)の値から始めるとQNewtonの結果から推測できるように、この目的関数は (0,0)から下降していくのですが、その(0,0)がboundaryになっているので、そこで止まるはずです。
画面結果

```
return code =    0
normal convergence

Value of objective function       27.000000

Parameters    Estimates
----------------------------------------
P01               0.0000
P02               0.0000


Linear Equality Lagrangean Coefficients
                  .

Linear Inequality Lagrangean Coefficients
                  .

Nonlinear Equality Lagrangean Coefficients
      0.00000000

Nonlinear Inequality Lagrangean Coefficients
      0.00000000

Bounds Lagrangean Coefficients

      0.00000000       0.00000000
      0.00000000       0.00000000

Number of iterations         2
Minutes to convergence     0.00550
```

上のように、リターンコードも正常な０で、(0,0)になっています。しかしながら、以前の関数の値７と比べて、ここの27という数字は明らかに大きいので、前回の(2,2)での関数の

値７が最小値になります。ただし、前回の７という数字は、明らかに手計算で７なのですが、gradient、Hessian関係の計算誤差によって端数の小数第何位かから下は信頼性はありません。その場合には、手計算で、これらの値をグローバル変数として代入して、精度を上げることになります。（ただし、精度が上がる場合とそうでない場合があります。）

プログラム

```
new; cls;
sqpSolveset;

fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
fn eqc(x)=x[1]-x[2];
fn ineqc(x)=-x[1]^2-x[2]^2+8;

_sqp_EqProc=&eqc;
_sqp_IneqProc=&ineqc;
_sqp_Bounds={ 0 1e256,
              0 1e256};
fn gradient(x)=(3*x[1]^2-9*x[2])~(3*x[2]^2-9*x[1]);
_sqp_GradProc=&gradient;
fn hessian(x)=(6*x[1])~(-9)|(-9)~(6*x[2]);
_sqp_HessProc=&hessian;
start={1,1};
call sqpSolve(&h,start);
```

画面表示

```
return code =    0
normal convergence

Value of objective function        7.000018

Parameters    Estimates
-----------------------------------------
P01               2.0000
P02               2.0000


Linear Equality Lagrangean Coefficients
                .
```

```
Linear Inequality Lagrangean Coefficients
                .

Nonlinear Equality Lagrangean Coefficients
       0.00000000

Nonlinear Inequality Lagrangean Coefficients
        1.5000011

Bounds Lagrangean Coefficients

       0.00000000        0.00000000
       0.00000000        0.00000000

Number of iterations         6
Minutes to convergence      0.00000
```

この場合の結果は、等号の制約が０になった以外は、以前と同じであまり精度があがったとは言えません。依然として、７であるものが、7.000018などとなっています。上のプログラムのように適当に名前をつけた関数に、この場合は２変数ですから１×２のgradientベクトルと２×２のHessian行列を定義してから、それをグローバル変数_sqp_GradProcおよび_sqp_HessProcでポインタとして&をつけて呼び出します。（なお、gradientベクトルは、２×１というふうに転置させる必要はなくて、そのまま１×２のまま使います。～および｜の記号でマージする際には、括弧( )で各項を囲んでマージしないとエラーを引き起こしますので注意が必要です。（これは、～と｜を使う際に、演算記号のある項と同時に使えないというGAUSSの文法からくるものです。）なお、sqpSolveは目的関数が多次の場合に有効な関数です。制約式はリニアであろうとノンリニアであろうと、またそれらが何本あろうと解いてくれます。（制約式が何本もある場合にはQProgのように１行１行積み重ねて行列として制約式のグローバル変数を設定します。以下がその設定の方法のまとめです。

ポイント　制約条件つき最小化問題 sqpSolve

{ x,f,lagr,retcode } = sqpSolve(&**目的関数名**,start);

または、call sqpSolve(&**目的関数名**,start);

Min　目的関数

s.t.　_sqp_A * X = _sqp_B

_sqp_C * X >= _sqp_D
_sqp_EqProc = &非線形等号制約関数;
_sqp_IneqProc = &非線形不等号制約関数;
_sqp_Bounds = { 下限　上限,
.....
下限　上限 }

ここで、xは解のベクトル、fは目的関数のxでの最小値、lagrはラグランジアンのラベルとそのベクトル、recodeはエラーのリターンコード。

ここで、上の線形の等号制約と不等号制約を使った例を、iterationの数や、gradientの収束の有効数字を設定するグローバル変数の使い方とあわせて示しておきましょう。今度は最初の等号制約を形どおり線形の等号制約としてAとｂにパラメーター設定してみます。

Min $h(x_1,x_2)=x_1^3+x_2^3-9x_1x_2+27$

s.t. $x_1=x_2$

$x_1+x_2>=2$

$2x_1+x_2<=4$

$x_1^2+x_2^2<=8$

$0<=x_1<=2,\quad 0<=x_2<=2$

１番目の線形の等号制約は、$x_1-x_2=0$という形にしたうえで、また、３番目の線形の不等号制約では、$-2x_1-x_2<=-4$というふうに等号不等号関係を－1をかけてひっくりかえせる形にします。すなわち、＞＝の向きではなく、＜＝の向きになっている場合には、CとdのCのその部分に相当する要素に－１をかけたものを代入します。

プログラム

```
new; cls;
sqpSolveset;

fn h(x)=x[1]^3+x[2]^3-9*x[1]*x[2]+27;
_sqp_A={1 -1};
_sqp_b={0};
_sqp_C={ 1    1,
         -2   -1};
_sqp_d={ 2,
         -4};
fn ineqc(x)=-x[1]^2-x[2]^2+8;
_sqp_IneqProc=&ineqc;
_sqp_Bounds={ 0 2,
              0 2};
```

```
    _sqp_MaxIters=100;
    _sqp_DirTol=1e-3;
    _sqp_PrintIters=1;

    start={0,0};
    { x,f,lagr,retcode }=sqpSolve(&h,start);
```

画面表示

```
-------------------------------------------------
 iter 1          function =     20.00000000
-------------------------------------------------

       parameter      direction        gradient
               1              1               0
               1              1               0


-------------------------------------------------
 iter 2          function =     15.74074074
-------------------------------------------------

       parameter      direction        gradient
       1.3333333       0.33333333       -6.0000001
       1.3333333       0.33333333       -5.9999998

================================================================================
 SQPSolve Version 3.6.4                                  2/01/2002   2:00 pm
================================================================================

return code =    0
normal convergence

Value of objective function       15.740741

Parameters    Estimates
----------------------------------------
```

```
P01             1.3333
P02             1.3333


Linear Equality Lagrangean Coefficients
       2.2222222

Linear Inequality Lagrangean Coefficients

      0.00000000
       4.4444444

Nonlinear Equality Lagrangean Coefficients
               .

Nonlinear Inequality Lagrangean Coefficients
      0.00000000

Bounds Lagrangean Coefficients

      0.00000000       0.00000000
      0.00000000       0.00000000

Number of iterations        3
Minutes to convergence     0.00000
```

上のように、線形の等号制約と２番目の非線形の不等号制約のgradientが非零です。すなわち、この２つによって決まってきている（２変数２方程式で解けてしまう）わけで、

$x_1 - x_2 = 0$

$2x_1 + x_2 <= 4$

を関係をすべて等号と見たてて連立させると、$x_1 = x_2 = 4/3$ となり、ここでの解のベクトルと一致します。

なお、プログラム後半のグローバル変数の設定は、まず、最大iteration数が

```
_sqp_MaxIters=1e+5;
```

とデフォルトで決まっているものを変更して、100にしてあります。設定は、指数形式でも十進法でもどちらでもかまいません。また、gradientベクトルの収束の際にどれだけの桁数まで０に近づいたら計算をやめるは、

```
_sqp_DirTol=1e-5;
```

とデフォルトで決まっているのを変更して、1e-3にしてあります。また、iterationをすべてのステップで出力するには、

```
_sqp_PrintIters=0;
```

とデフォルトで無効にされているのを、それ以外の数字、ここでは仮に１、とすることですべてのiterationが表示されます。以前と同じように、止まらなくなれば「C」で計算をそこでやめます。なお、リターンコードは０が正常収束ですが、ここにまとめておくと、

リターンコード

0	**正常収束**
1	**強制終了**
2	**最大iteration回数超過**
3	**関数計算失敗**
4	**gradient計算失敗**
5	**Hessian計算失敗**
6	**ラインサーチ失敗**
7	**制約式エラー**

なお、QNewtonもsqpSolveもノンリニアプログラミング関係のために作られた関数ですが、計量のライブラリmaxlikやcmlにたよることなく、これらでノンリニアの計量計算をすませてしまうことも可能です。