5.15 Kernel の基礎(2)　　ver.0.1

引き続き Kernel の基礎についてプログラムします。ここではまず Bivariate の場合の Kernel Density についてです。まず Bivariate Normal をシミュレーションするためにデータの組数ｎを指定してから、Variance Covariance 行列を設定します。この場合の平均は２変数ともに便宜上０にしてあります。ここで、x=rndn(n,2)*chol(vc);とあるのは、２列の標準正規乱数を生成したものに Variance Covariance 行列を Cholesky 分解したものをかけています。Cholesky 分解は行列のルートのようなものですから、これをかけているというのは Variance Covariance 行列のルートしたようなものをかけていると考えてください。ちょうど、σ＝２であれば rndn(n,1)*sigma とすることによって N(0,4)にしたがう乱数が作成できるのと要領は同じです。そこに Covariance が加わっているので Cholesky 分解で行列を変換しているだけです。これによってシミュレートされたデータを plotbikern でグラフ化することにします。なお、この場合のデータは Standardized されていないものを使用してそのまま計算するものとします。

プログラム

```
new; cls;
library pgraph;
graphset;
n=50;
vc={2       0.7,
      0.7    2    };
rndseed 1;
x=rndn(n,2)*chol(vc);
call plotbikern(x,vc);

proc plotbikern(x,vc);
    local point1,point2,points,i,j,dens,kern,x0;
    point1=seqa(minc(minc(x[.,1]),(maxc(x[.,1])-minc(x[.,1]))/98),99);
    point2=seqa(minc(minc(x[.,2]),(maxc(x[.,2])-minc(x[.,2]))/98),99);
    points=point1~point2;
    i=1; dens=zeros(99,99); kern=zeros(99,99);
    do while i<=99;
        j=1;
        do while j<=99;
            x0=points[i,1]~points[j,2];
            dens[i,j]=inv(2*pi)*inv(sqrt(det(vc)))*exp(-0.5*x0*inv(vc)*x0');
```

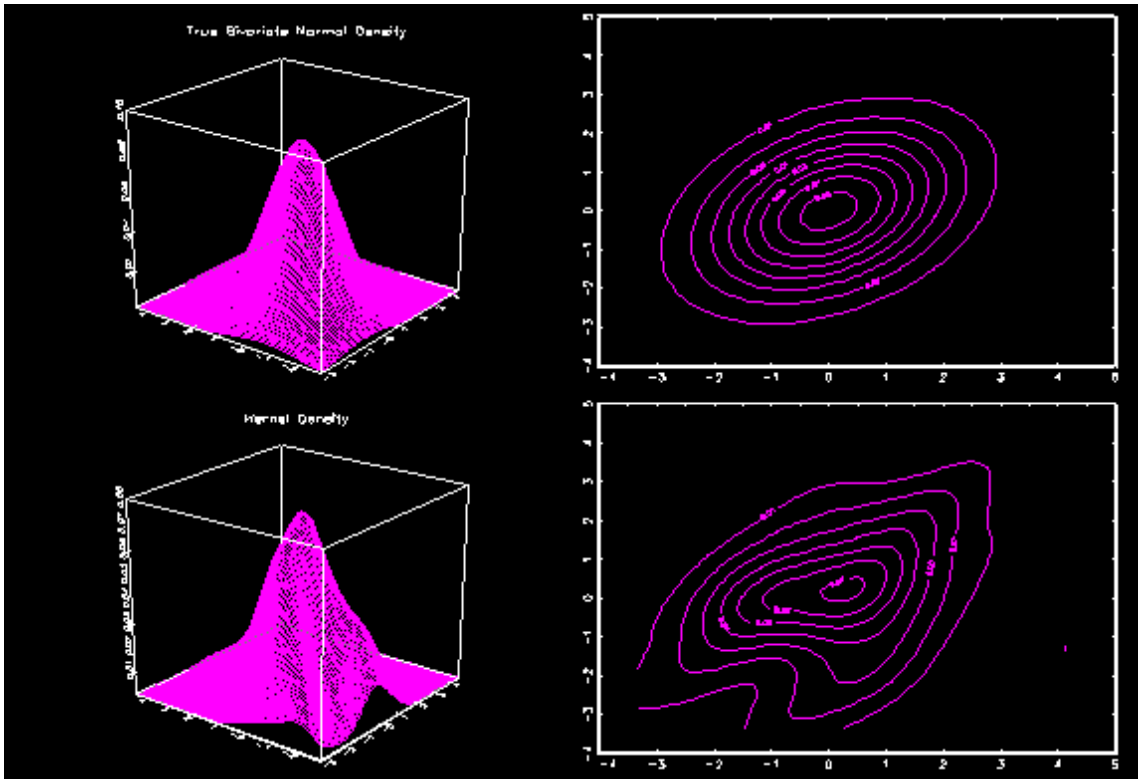
```
            kern[i,j]=kerneldens(x0,x);
            j=j+1;
        endo;
        i=i+1;
    endo;
/* Plot graphs */
    begwind;
    window(2,2,0);
    setwind(1);
        graphset;
        title("True Bivariate Normal Density");
        surface(point1',point2,dens);
    setwind(2);
        graphset;
        contour(point1',point2,dens);
    setwind(3);
        graphset;
        title("Kernel Density");
        surface(point1',point2,kern);
    setwind(4);
        graphset;
        contour(point1',point2,kern);
    endwind;
    retp(kern);
endp;

proc kerneldens(x0,x);
    local n,k,h0,hh,invs,dets,u,kn;
    n=rows(x);
    h0=1; /* ho depends on the type of kernels. */
    hh=h0*n^(-1/6);
    invs=inv(chol(vcx(x)));
    dets=sqrt(det(vcx(x)));
    u=(x0-x)./hh;
    kn=kernels(u*invs,"gauss");
    retp(sumc(kn)/(n*((hh^2)*dets)));
```

```
endp;

proc kernels(u,types);
    local kv;
    if types$=="uniform";
        kv=0.5*(abs(u).<=1);
    elseif types$=="gauss";
        kv=(1/sqrt(2*pi))*exp(-0.5*u^2);
    elseif types$=="triangular";
        kv=(1-abs(u)).*(abs(u).<=1);
    elseif types$=="biweight";
        kv=(15/16)*((1-u^2)^2).*(abs(u).<=1);
    elseif types$=="triweight";
        kv=(35/32)*((1-u^2)^3).*(abs(u).<=1);
    elseif types$=="epanechnikov";
        kv=0.75*(1-u^2).*(abs(u).<=1);
    else;
        errorlog "ERROR:    Type must be either of the following:";
        errorlog "uniform/gauss/triangular/biweight/triweight/epanechnikov";
    endif;
    retp( prodc(kv') );
endp;
```

上のプログラムでは、Univariate のケースとは違って、Kernel を計算する一番下の計算のリターンが kv そのものではなくて prodc(kv')としています。これは２次元であるため、こうなっています。さらに、その上の procedure の計算では、sumc(kn)/(n*((hh^2)*dets))をリターンになっており、ｈｈ＾２と２乗になります。評価点は x0 で Univariate と変更はありません。Kernel の計算を procedure で呼び出すところで kn=kernels(u*invs,"gauss");というふうに invs=inv(chol(vcx(x)));でかけているのは、standardized をここでしていることになります。１変数であればσで割ることに相当します。また、ここではバンド幅を h0 をまず１に止めておいて、hh=h0*n^(-1/6);によって最終的なバンド幅を概算で決定しています。ここで(-1/6)というのは、-0.2 が１変数のときで、この場合２変数ですから５に１が加わって、都合６の(-1/6)となっています。これをもとに、一番上のところで、True Density は dens[i,j]=inv(2*pi)*inv(sqrt(det(vc)))*exp(-0.5*x0*inv(vc)*x0');により計算して、Kernel の計算 kern[i,j]=kerneldens(x0,x);とともに 99×99 のグリッド上の高さを計算していることになります。なお、この 99×99 のグリッドの大きさは任意なのですが、それぞれの軸の最小値と最大値を求め、その間を 98 分割して 99×99 のグリッドを作成しています。なお、

contour グラフは奇数グリッドを必要とするので（偶数ではエラーになります）99 にとどめています。また、ｘ軸の方を列から行に転置することで、ｙ軸とは変えることによって、グリッド座標が自動的に与えられることはグラフィックスのところで行なったのと同じ要領です。そうすると、下のように上段には True Density の３次元グラフとその等高線。下段には、Kernel Density とその等高線が描かれます。

グラフ表示



Kernel Regression(２次元のケース)

ここでは、また Univariate のケースに立ち戻って、それを用いた Regression を考えます。定数項と１独立変数からなるモデルを考えて、まずは通常の OLS で計算できる回帰直線 yhat と Kernel Regression でもとめられた ghat とをプロットするプログラムを考えます。線形モデル $y = \beta_0 + \beta_1 X + u$ に対して、乱数設定を次のようにします。

n	β_0	β_1	X	u
1000	1	4	[0,5]	N(0,1)

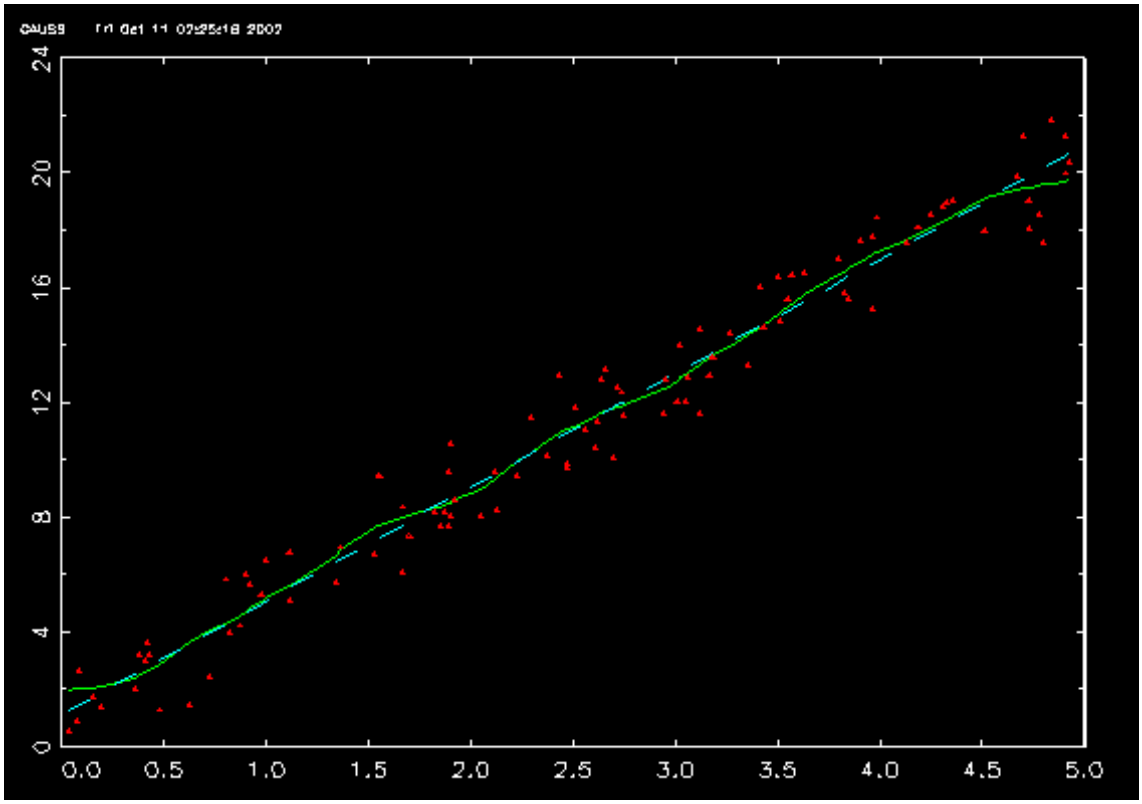
プログラム

```
new; cls;
library pgraph;
graphset;
rndseed 1000;
```

```
/* Simulation */
    n=100;
    x=5*rndu(n,1); u=rndn(n,1); b={1,4};
    y=(ones(n,1)~x)*b+u;
/* Regression */
    ghat=kernelreg(y,x);
    x1=ones(n,1)~x;
    yhat=x1*(inv(x1'x1)*x1'y);
/* Plot */
    data=sortc(x~ghat~yhat~y,1);
    _plctrl={0,0,-1};
    _psymsiz=1;
    xy(data[.,1],data[.,2:4]);

proc kernelreg(y,x);
    local n,k,h0,h,invs,i,u,kv,w,ghat;
    n=rows(x); k=cols(x);
    h0=1; /* h0 depends on the type of kernels. */
    h=h0*n^(-1/(k+4));
    invs=inv(chol(vcx(x)));
    i=1; ghat=zeros(n,1);
    do while i<=n;
        u=(x[i]-x)./h;
        kv=kernels(u*invs,"epanechnikov");
        w=kv./sumc(kv);
        ghat[i]=w'y;
        i=i+1;
    endo;
    retp(ghat);
endp;

proc kernels(u,types);
    local kv;
    if types$=="uniform";
        kv=0.5*(abs(u).<=1);
    elseif types$=="gauss";
```

```
        kv=(1/sqrt(2*pi))*exp(-0.5*u^2);
    elseif types$=="triangular";
        kv=(1-abs(u)).*(abs(u).<=1);
    elseif types$=="biweight";
        kv=(15/16)*((1-u^2)^2).*(abs(u).<=1);
    elseif types$=="triweight";
        kv=(35/32)*((1-u^2)^3).*(abs(u).<=1);
    elseif types$=="epanechnikov";
        kv=0.75*(1-u^2).*(abs(u).<=1);
    else;
        errorlog "ERROR:    Type must be either of the following:";
        errorlog "uniform/gauss/triangular/biweight/triweight/epanechnikov";
    endif;
    retp( kv );
endp;
```

グラフ表示



上のグラフでは、赤い点がシミュレーションによって作成した実際データ。緑の実線がKernelで推定したもの。ブルーの破線がOLSによるものです。直線で推定されるOLSとは違って、残差項の値の影響を受けてその都度曲がっていることがわかります。プログラムにおいて、kv=kernels(u*invs,"epanechnikov");でまず指定されたKernelタイプに対して、

uにinvsをかけたものインプットに使っていますが、この場合１変数（Kernelの場合定数項は関係ない）なのでσで割っても同じことです。ここでは一般化しているだけです。次に、w=kv./sumc(kv);のところでウエイトを計算しています。これを w'y として１からｎまでループを回すことで Kernel Regression になります。

同じことは、非線形のデータについてもあてはめることができます（むしろ、こちらの方が Kernel Regression の適用分野と言えます）。ここでは、Polynomial $y=2X^3-X^2+3X+u$ に対して、乱数設定を次のようにするものとします。

n	X	u
100	[-2,2]	$N(0,2^2)$

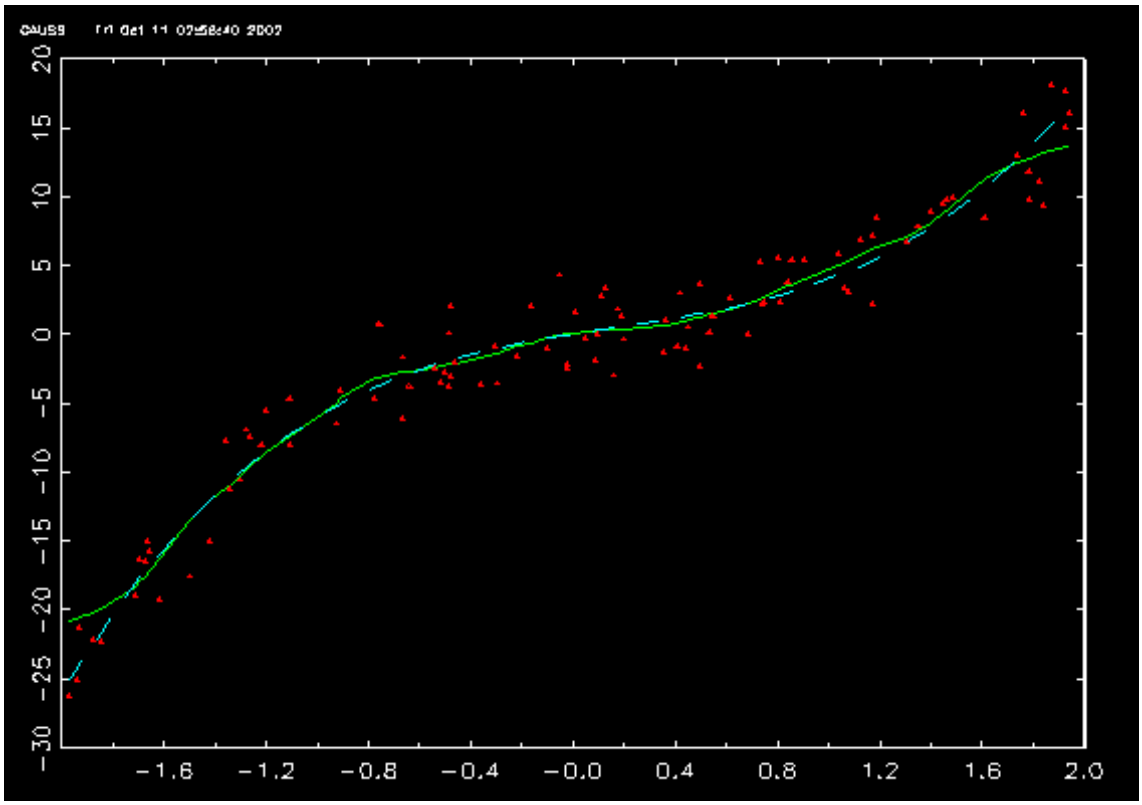
プログラム

```
new; cls;
library pgraph;
graphset;
rndseed 1000;
/* Simulation */
    n=100;
    x=4*rndu(n,1)-2; u=2*rndn(n,1);
    y=2*x^3-x^2+3*x+u;
    yhat=2*x^3-x^2+3*x;
/* Regression */
    ghat=kernelreg(y,x);
/* Plot */
    data=sortc(x~ghat~yhat~y,1);
    _plctrl={0,0,-1};
    _psymsiz=1;
    xy(data[.,1],data[.,2:4]);

proc kernelreg(y,x);
    local n,k,h0,h,invs,i,u,kv,w,ghat;
    n=rows(x); k=cols(x);
    h0=1; /* h0 depends on the type of kernels. */
    h=h0*n^(-1/(k+4));
    invs=inv(chol(vcx(x)));
    i=1; ghat=zeros(n,1);
    do while i<=n;
```

```
        u=(x[i]-x)./h;
        kv=kernels(u*invs,"epanechnikov");
        w=kv./sumc(kv);
        ghat[i]=w'y;
        i=i+1;
    endo;
    retp(ghat);
endp;

proc kernels(u,types);
    local kv;
    if types$=="uniform";
        kv=0.5*(abs(u).<=1);
    elseif types$=="gauss";
        kv=(1/sqrt(2*pi))*exp(-0.5*u^2);
    elseif types$=="triangular";
        kv=(1-abs(u)).*(abs(u).<=1);
    elseif types$=="biweight";
        kv=(15/16)*((1-u^2)^2).*(abs(u).<=1);
    elseif types$=="triweight";
        kv=(35/32)*((1-u^2)^3).*(abs(u).<=1);
    elseif types$=="epanechnikov";
        kv=0.75*(1-u^2).*(abs(u).<=1);
    else;
        errorlog "ERROR:    Type must be either of the following:";
        errorlog "uniform/gauss/triangular/biweight/triweight/epanechnikov";
    endif;
    retp( kv );
endp;
```

プログラムは冒頭の設定部分が違うだけで、procedeure 部分は線形のものと同じです。Kernel Regression は真のモデルがいかなるものであっても、あるいはモデルがそもそもないようのデータに対しても Regression が可能です。なお、上の設定では、Polynomial は直接の推定が困難であるため、便宜上もとのモデルの乱数項 u がないものを yhat としています。そうして、プロットすると次のようになります。

グラフ表示



Kernel Regression(３次元のケース)

独立変数が定数項を勘定に入れないで２つの場合は、Bivariate と同じ手順で推定を行ないます。以下では線形モデル $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + u$ に対して、乱数設定を

n	β_0	β_1	β_2	X1	X2	u
100	2	3	4	[0,5]	[0,5]	N(0,1)

として計算をします。

プログラム

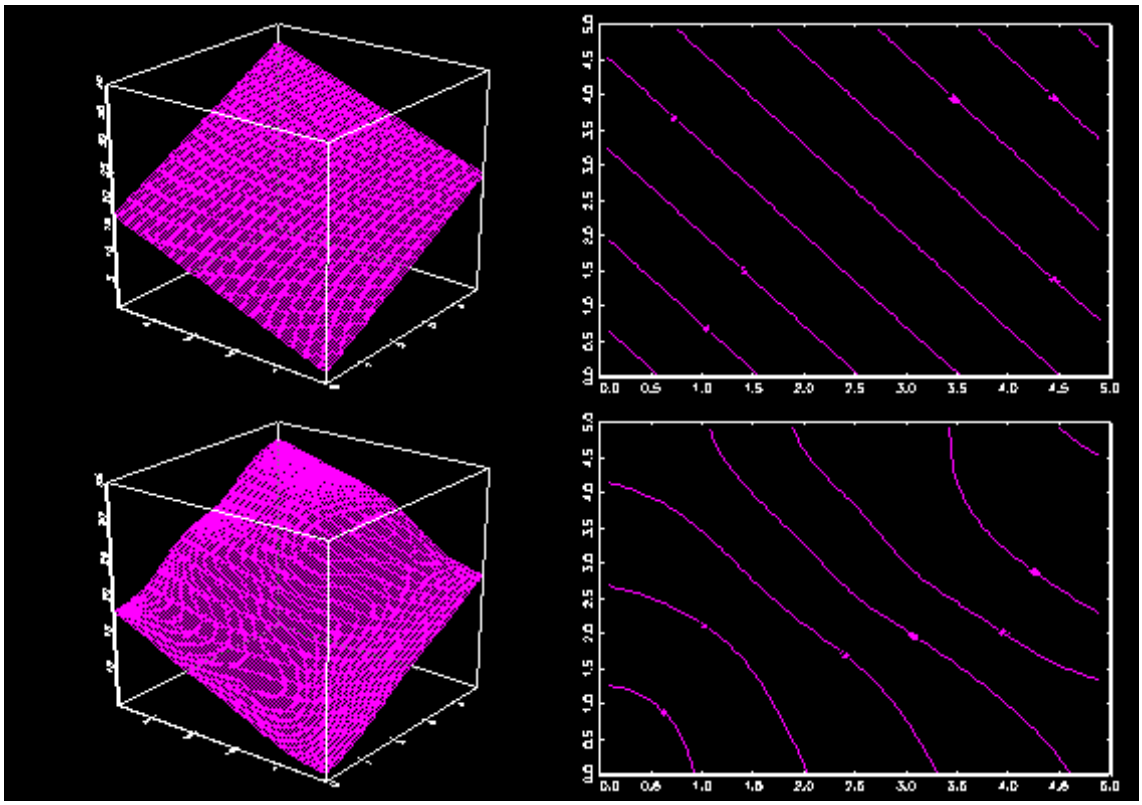
```
new; cls;
library pgraph;
graphset;
rndseed 1000;
/* Simulation */
    n=100;
    x=5*rndu(n,2); u=rndn(n,1); b={2,3,4};
    y=(ones(n,1)~x)*b+u;
/* Regression & Graph */
    ghat=kernelreg(y,x);
```

```
proc kernelreg(y,x);
    local n,k,h0,h,invs,i,u,kv,w,ghat,point1,point2,points,x0,j,x1,bhat,yhat;
    n=rows(x); k=cols(x);
    h0=1; /* h0 depends on the type of kernels. */
    h=h0*n^(-1/(k+4));
    invs=inv(chol(vcx(x)));
    point1=seqa(minc(x[.,1]),(maxc(x[.,1])-minc(x[.,1]))/98,99);
    point2=seqa(minc(x[.,2]),(maxc(x[.,2])-minc(x[.,2]))/98,99);
    points=point1~point2;
    i=1; ghat=zeros(99,99);
    do while i<=99;
        j=1;
        do while j<=99;
            x0=points[i,1]~points[j,2];
            u=(x0-x)./h;
            kv=kernels(u*invs,"gauss");
            w=kv./sumc(kv);
            ghat[i,j]=w'y;
            j=j+1;
        endo;
        i=i+1;
    endo;
    x1=ones(n,1)~x;
    bhat=inv(x1'x1)*x1'y;
    i=1; yhat=zeros(99,99);
    do while i<=99;
        j=1;
        do while j<=99;
            x0=1~points[i,1]~points[j,2];
            yhat[i,j]=x0*bhat;
            j=j+1;
        endo;
        i=i+1;
    endo;
/* Plot panel graphs */
    begwind;
```

```
    window(2,2,0);
    setwind(1);
        surface(point1',point2,yhat);
    setwind(2);
        contour(point1',point2,yhat);
    setwind(3);
        surface(point1',point2,ghat);
    setwind(4);
        contour(point1',point2,ghat);
    endwind;
    retp(ghat);
endp;

proc kernels(u,types);
    local kv;
    if types$=="uniform";
        kv=0.5*(abs(u).<=1);
    elseif types$=="gauss";
        kv=(1/sqrt(2*pi))*exp(-0.5*u^2);
    elseif types$=="triangular";
        kv=(1-abs(u)).*(abs(u).<=1);
    elseif types$=="biweight";
        kv=(15/16)*((1-u^2)^2).*(abs(u).<=1);
    elseif types$=="triweight";
        kv=(35/32)*((1-u^2)^3).*(abs(u).<=1);
    elseif types$=="epanechnikov";
        kv=0.75*(1-u^2).*(abs(u).<=1);
    else;
        errorlog "ERROR:    Type must be either of the following:";
        errorlog "uniform/gauss/triangular/biweight/triweight/epanechnikov";
    endif;
    retp( prodc(kv') );
endp;
```

基本的に冒頭で行なった Bivariate の Kernel 計算と同じです。ここでも w=kv./sumc(kv); としてウエイトを求めて、w'y を計算し Gird の数だけループを回しています。ただし、ここの kv は一番下の Bivariate の procedure のリターン prodc(kv')のことです。

グラフ表示



グラフの上段は OLS による回帰面、下段は Kernel 回帰による面です。当然のことながら線形の OLS による回帰面の等高線は直線になります。

Least Square Jackknife による最適バンド幅の計算

これまでは、バンド幅の概算を求めていました。今度はある統計基準を満たすバンド幅を求めて、よりスムースなデータを反映した Kernel Rgression になるようにしてみましょう。いろいろなやり方がありますが、一番単純な方法として、もとのデータｙとその Kernel 推定値 ghat との差の二乗和を計算するのに、Jackknife を用いて１つずつ取り除いたものを用いて、その Jackknife Least Square を最小にするバンド幅ｈを選ぶ方法があります。

プログラム

```
new; cls;
library pgraph;
graphset;
rndseed 1000;
/* Simulation */
    n=100;
    x=10*rndu(n,1)-5; u=10*rndn(n,1);
    y=x^3+u;
/* Regression */
```

```
    h=linegrid(&f,0,10);
    ghat=kernelreg(y,x,h);
    yhat=x^3;
/* Plot */
    data=sortc(x~ghat~yhat~y,1);
    _plctrl={0,0,-1};
    _psymsiz=1;
    xy(data[.,1],data[.,2:4]);

proc f(h);
    local n,k,h0,h,invs,i,u,kv,w,ghat,x1,y1,index;
    n=rows(x); k=cols(x);
    i=1; ghat=zeros(n,1);
    do while i<=n;
        index=zeros(n,1);
        index[i]=1;
        u=(x[i]-x)./h;
        u=delif(u,index);
        x1=delif(x,index);
        invs=inv(chol(vcx(x1)));
        kv=kernels(u*invs,"triweight");
        w=kv./sumc(kv);
        y1=delif(y,index);
        ghat[i]=w'y1;
        i=i+1;
    endo;
    retp( -(y-ghat)'(y-ghat) );
endp;

proc kernelreg(y,x,h);
    local n,k,h0,h,invs,i,u,kv,w,ghat;
    n=rows(x); k=cols(x);
    invs=inv(chol(vcx(x)));
    i=1; ghat=zeros(n,1);
    do while i<=n;
        u=(x[i]-x)./h;
```

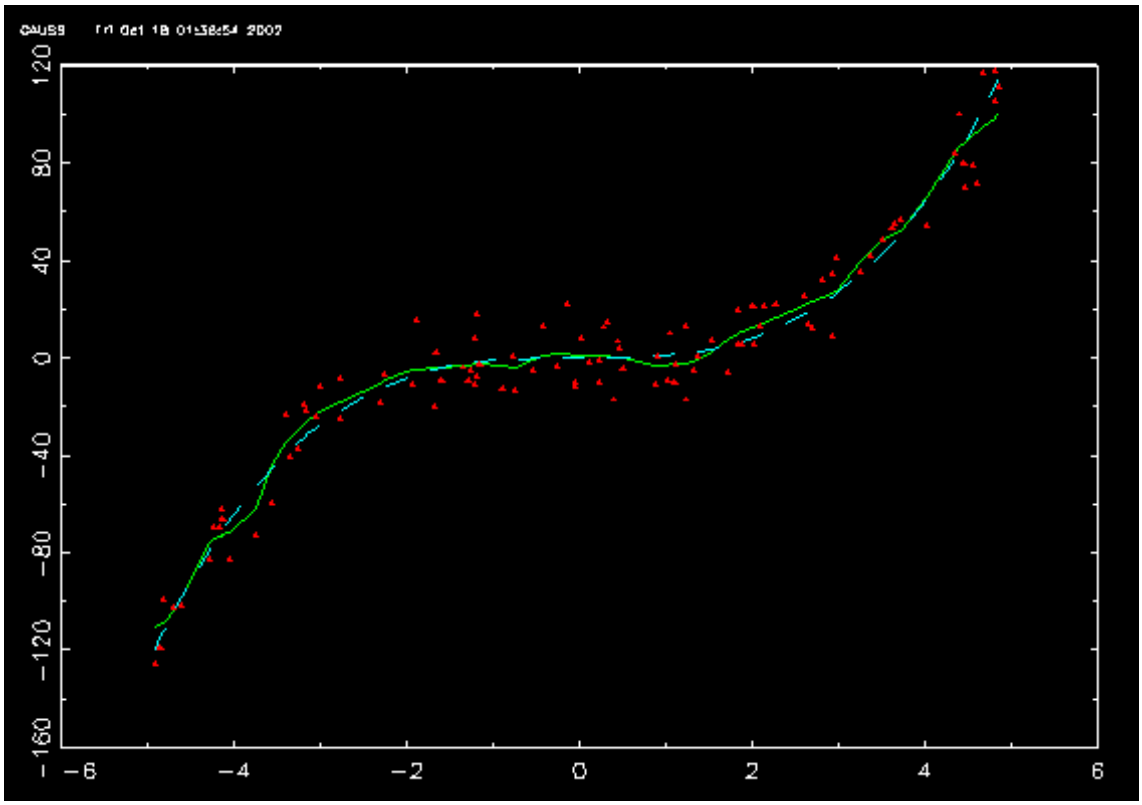
```
        kv=kernels(u*invs,"triweight");
        w=kv./sumc(kv);
        ghat[i]=w'y;
        i=i+1;
    endo;
    retp(ghat);
endp;

proc kernels(u,types);
    local kv;
    if types$=="uniform";
        kv=0.5*(abs(u).<=1);
    elseif types$=="gauss";
        kv=(1/sqrt(2*pi))*exp(-0.5*u^2);
    elseif types$=="triangular";
        kv=(1-abs(u)).*(abs(u).<=1);
    elseif types$=="biweight";
        kv=(15/16)*((1-u^2)^2).*(abs(u).<=1);
    elseif types$=="triweight";
        kv=(35/32)*((1-u^2)^3).*(abs(u).<=1);
    elseif types$=="epanechnikov";
        kv=0.75*(1-u^2).*(abs(u).<=1);
    else;
        errorlog "ERROR:    Type must be either of the following:";
        errorlog "uniform/gauss/triangular/biweight/triweight/epanechnikov";
    endif;
    retp( kv );
endp;

proc linegrid(&ll,start,finish);
    local length,maxiter,tol,i,j,step,grid,b,temp,bmax,ll:proc;
        length=100;
        maxiter=100;
        tol=1e-8;
    bmax=0;
    i=1;
```

```
    do while i<=maxiter;
        step=(finish-start)/length;
        grid=(-1e256).*ones(length+1,1);
        b=start;
        j=1;
        do while j<=length+1;
            grid[j]=ll(b);
            j=j+1;
            b=b+step;
         endo;
         temp=start+(maxindc(grid)-1)*step;
         if abs(bmax-temp)<tol;
            break;
         endif;
         bmax=temp;
         print/rd bmax;
         start=bmax-step; finish=bmax+step;
         i=i+1;
    endo;
    print/rz "# of iterations=" i-1;
    print/rd bmax;
    retp(bmax);
endp;
```

画面表示

```
        0.20000000
        0.25400000
        0.25312000
        0.25312400
        0.25312362
# of iterations=                5
        0.25312362
```

グラフ表示



上のように、自動的にｈは Grid(Line)Search によって Jackknife Least Square f(h)を最小にするｈをまず求めて、それを proc kernelreg(y,x,h)とｈを加えて３インプットに変更して内部のｈの計算部分をなくした procedure にこの計算されたｈを受け渡しています。その他の部分の Kernel Regression はその前のものと同じです。ただし、f(h)を計算するのにｘとｙはインプットとしてではなくてグローバルに通しています。その１値関数 f(h)を以前に作成した Grid(Line)Search のプログラムで関数ｆを＆をつけて呼び出して、０から 10 までの範囲を初期サーチの範囲として最小点を計算しています。中心となる f(h)は、kernelreg をもう一回やったような形になっていますが、実際には、Jackknife された ghat を求めるために、Jackknife のところで行なったと全く同じように、まず０でできたｎ×１の index を毎回作成して、そのｉ番目を１とします。それを論理と考えて、delif(列ベクトル,論理)でｉ番目だけを切り落としています。この場合計算された u と invs を計算するためのｘ、それにｙそのものの３つとも Jackknife で毎回切り落とします。そしてこれまでの kernel regression と同様に１からｎまでループを回して ghat を計算します。これともともとのｙとの差の二乗和を計算して、この場合の Grid(Line)Search の procedure は最大値を求めるものだから、便宜上マイナスをつけたものをリターンにしておきます。なお、Jackknife をしないで Least Square を計算したものを最小にしようとすると、ｈは限りなく０に近くなり結果は全ての点を結ぶ軌跡になります。