Local Polynomial Regression: How the R Package interp estimates partial derivatives for later use in Spline Interpolation

Albrecht Gebhardt
University Klagenfurt

Roger Bivand
Norwegian School of Economics

Abstract

This vignette presents the R package **interp** and focuses on local polynomial regression for estimating partial derivatives.

This is the first of planned three vignettes for this package (not yet finished).

Keywords: local polynomial regression, partial derivatives, R software.

1. Note

Notice: This is a preliminary and not yet complete version of this vignette. Finally three vignettes will be available for this package:

1. this one related to partial derivatives estimation,

2. a next one describing interpolation related stuff

3. and a third one dealing with triangulations and voronoi mosaics.

Currently only the first text is available and not yet finished.

2. Introduction

Altough the main intention of this `R` package is interpolation, it also contains routines for local polynomial regression. The reason is that the spline interpolation implemented by `interp::interp(...,method="akima")` needs estimates of the partial derivatives of the interpolated function up to degree 2.

One approach to get such estimates is to perform a local polynomial regression (see e.g. **?**, p. 19) and get the partial derivatives as a side effect, as explained later. This is also applied in Akimas original code in a special hardcoded way (using a fixed local bandwidth and a uniform kernel). Once this routines had been implemented and used internally in the `interp` ↪ `::interp(...,method="akima")` it was an obvious decision to make these routines also available to end users of package `"interp"`.

3. Kernel Functions

In the next section we will use the notion of kernel functions, so lets start with this definition.

Definition 3.1. *A one-dimensional kernel function $K(x)$ is*

1. *a density function, hence*
   (a) $K(x) \geq 0$
   (b) $\int_{\mathbb{R}} K(x)dx = 1$

   Lets denote the associated stochastic variable with X_K for easier notation, it otherwise carries no meaning.

2. *K has the property $\int_{\mathbb{R}} x \cdot K(x) = 0$ (i.e. $\mathbb{E}X_K = 0$, kernel function is centered at zero) and*

3. *K is assumed to be symmtric $K(-x) = K(x)$ and*

4. *$0 < \int_{\mathbb{R}} x^2 \cdot K(x)dx = \sigma_K^2 < \infty$, i.e. $VarX_K$ exists.*

The kernel functions currently implemented in this library are listed in table **??**.

name	function	support of K (outside: $K(x) = 0$)
gaussian	$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$	$x \in \mathbb{R}$
cosine	$\frac{1}{2}\cos(x)$	$x \in (-\frac{\pi}{2}, \frac{\pi}{2}]$
epanechnikov	$\frac{3}{4}(1-x^2)$	$x \in (-1, 1]$
biweight	$\frac{15}{16}(1-x^2)^2$	$x \in (-1, 1]$
tricube	$\frac{70}{81}(1-\lvert x\rvert^3)^3$	$x \in (-1, 1]$
triweight	$\frac{35}{32}(1-x^2)^3$	$x \in (-1, 1]$
uniform	$\frac{1}{2}$	$x \in (-1, 1]$
triangular	$1-\lvert x\rvert$	$x \in (-1, 1]$

Table 1: kernels

A common approach to create twodimensional kernel functions is to derive them from onedimensional kernels as bivariate densities with independent components:

$$K_{X,Y}(x, y) \quad = \quad K_X(x)K_Y(y)$$

Both K_X and K_Y are choosen from the same kernel function type.

4. Bivariate Local Polynomial Regression

Lets start with a data set $\{(\underline{x}_i, z_i) | i = 1, \dots, n\}$ with vectors $\underline{x}_i = (x_i, y_i)^\top \in \mathbb{R}^2$ and real numbers $z_i \in \mathbb{R}$. Assume a trend model

$$z = m(\underline{x}) + \varepsilon$$

with independent random errors ε and a bivariate polynomial of degree r as setup for m:

$$m(\underline{x}) = m(x,y) = \sum_{i=0}^{r}\sum_{j=0}^{r-i}\beta_{ij}x^i y^j.$$

Note that the sum of exponents i and j in each term of the sum is bounded above by r.

Local regression aims to minimize a weighted sum of squares where the weights are determined by a bivariate kernel function centered at the actual location for prediction $\underline{x}$ which decreases with increasing distance from this centering point:

$$\sum_{k=1}^{n} K_X\left(\frac{x-x_k}{h_x}\right)K_Y\left(\frac{y-y_k}{h_y}\right)\left[z_k - \sum_{i=0}^{r}\sum_{j=0}^{r-i}\beta_{ij}x_k^i y_k^j\right]^2 \rightarrow Min$$

A Taylor expansion of $m(x,y)$ in a location $\underline{x}_0 = (x_0, y_0)$ can be used as a starting point to interpret the estimated parameters:

$$\begin{aligned}
m(x,y) &= \sum_{i=0}^{r-1}\sum_{j=0}^{r-1-i}\frac{\frac{\partial^{i+j}m}{\partial x^i \partial y^j}(x_0)}{i!j!}(x-x_0)^i(y-y_0)^j \\
&= \sum_{i=1}^{r}\sum_{j=1}^{r-i}\underbrace{\frac{\frac{\partial^{i+j}m}{\partial x^{i-1}\partial y^{j-1}}(x_0)}{(i-1)!(j-1)!}}_{=\beta_{ij}}(x-x_0)^{i-1}(y-y_0)^{j-1} \\
&= \sum_{i=1}^{r}\sum_{j=1}^{r-i}\beta_{ij}(x-x_0)^{i-1}(y-y_0)^{j-1}
\end{aligned}$$

With the estimates $\widehat{\beta}_{ij}, i = 1,\ldots,r, j = 1,\ldots,r-i$ for a given location $\underline{x}$ we evaluate this Taylor expansion at $\underline{x} = \underline{x}_0$, which means that all terms $(x-x_0)^i(y-y_0)^j$ with $i > 0$ or $j > 0$ vanish. Only the estimated function and its derivatives at location $\underline{x} = \underline{x}_0$ remain:

$$\begin{aligned}
\widehat{m}(x,y) &= \sum_{i=1}^{r}\sum_{j=1}^{r-i}\widehat{\beta}_{ij}(x-x_0)^{i-1}(y-y_0)^{j-1} && (1)\\
&= \widehat{\beta}_{1,1}y && (2)
\end{aligned}$$

The remaining components of $\widehat{\beta}$ can now be used to estimate the values of the derivatives of m in

$$\frac{\widehat{\partial^{i+j}m}}{\partial x^i \partial y^j}(x_0) = (i-1)!(j-1)!\widehat{\beta}_{ij}, \quad i = 1,\ldots,r, j = 1,\ldots,r-i \qquad (3)$$

5. Implementation details

Function `interp::locpoly()` returns estimated values of the regression function as well as estimated partial derivatives up to order 3 (Akima splines only need derivatives up to order

2). This access to the partial derivatives was the main intent for writing this code as there are already many other local polynomial regression implementations in R. Beside the univariate local estimators `stats::ksmooth()`, `locpol::locPolSmootherC()` and `KernSmooth` ↪ `::locpoly()` (the last two also return univariate derivatives) the packages `locfit` and `sm` provide amongst other things bivariate local regression methods. But to our knowledge currently (spring 2022) no bivariate local regression estimators for partial derivatives exists. But anyhow, to be used from within the C++ implementation of `interp::interp()` we had to implement this estimator directly also in C++ in package interp.

This is a short overview (to be extend in a later version of this document) of the steps that had to be implemented:

- Formulate the normal equations for the above weighted least squares problem.
- Use package `RcppEigen` to perform the numeric solution.
- Package `RcppEigen` provides a sample implementaion `fastLm` to solve ordinary (unweighted) least squares problems. We just used this and extended it for the weighted case.
- `fastLm` has the option to use different solvers provided in `RcppEigen`. Our implementation inherits these options.

6. Application To A Regular Grid

We will test `locpoly()` now with a bicubic polynomial on the unit square on a `ng` by `ng` grid. Later tests using Franke functions will follow.

Set the x - y size of a square data grid to

```
> ng <- 11
```

resulting in 121 grid points.

First let us choose a kernel

```
> knl <- "gaussian"
```

Other Options would have been `"uniform"`, `"cosine"`, `"biweight"`, `"triweight"`, `"tricube"` and `"epanechikov"`, compare section **??**.

Next both a fixed global and a varying local bandwidth is needed:

```
> bwg <- 0.33
> bwl <- 0.11
```

The global bandwidth (=0.33) is interpreted as ratio of the x and y range respective. So in this example the "moving window" of the kernel function covers a rectangular data region of $1/3 \times 1/3 = 1/9$ of the bounding box of the data set.

The local bandwidth indicates the proportion of the data set choosen as local search neighbourhood. Its value 0.11 has been choosen to match the coverage of the global bandwidth above.

Now set the degree of the local polynomial model (maximum supported value is 3)

```
> dg=3
```

and define a bicubic polynomial:

```
> f <- function(x,y) (x-0.5)*(x-0.2)*(y-0.6)*y*(x-1)
```

Now we prepare symbolic derivatives of f both for calculating exact values (via package `Deriv`) and for pretty printing (using package `Ryacas`). The helper functions used for these preparation steps are shown in appendix **??**:

```
> df <- derivs(f,dg)
```

Now build and fill the grid with the theoretical values:

```
> xg <- seq(0,1,length=ng)
> yg <- seq(0,1,length=ng)
> xyg <- expand.grid(xg,yg)
```

and prepare a finer grid for detailed plotting at a larger resolution by increasing the grid density by factor 4 in both axes:

```
> af <- 4
> xfg <- seq(0,1,length=af*ng)
> yfg <- seq(0,1,length=af*ng)
> xyfg <- expand.grid(xfg,yfg)
```

Create coordinate matrices `xx` and `yy` as matching the grid matrix `fg`

```
> nx <- length(xg)
> ny <- length(yg)
> xx <- t(matrix(rep(xg,ny),nx,ny))
> yy <- matrix(rep(yg,nx),ny,nx)
```

Now fill all exact results deriven from symbolic computation into the grid matrices, again one of the helper functions from appendix **??** is used:

```
> ## data for local regression
> fg   <- outer(xg,yg,f)
> ## data for exact plots on fine grid
> ffg <- fgrid(f,xfg,yfg,dg)
```

Now perform the local regression estimation, get both global and local bandwidth results:

```
> ## global bandwidth:
> pdg <- interp::locpoly(xg,yg,fg, input="grid", pd="all", h=c(bwg,bwg), solver="QR", degree
    ↪ =dg,kernel=knl,nx=af*ng,ny=af*ng)
> ## local bandwidth:
> pdl <- interp::locpoly(xg,yg,fg, input="grid", pd="all", h=bwl, solver="QR", degree=dg,
    ↪ kernel=knl,nx=af*ng,ny=af*ng)
```

Now finally generate the plots. Again a collection of helper function is used here to fit all 10 plots and descriptions in a single plot. For interested users they are shown in the appendix.

```
> pf <- gg1image2contours(xfg,yfg,ffg$f,pdg$z,pdl$z,xyg,"f")
> pfx <- gg1image2contours(xfg,yfg,ffg$fx,pdg$zx,pdl$zx,xyg,"f_x")
> pfy <- gg1image2contours(xfg,yfg,ffg$fy,pdg$zy,pdl$zy,xyg,"f_x")
> pfxx <- gg1image2contours(xfg,yfg,ffg$fxx,pdg$zxx,pdl$zxx,xyg,"f_xx")
> pfxy <- gg1image2contours(xfg,yfg,ffg$fxy,pdg$zxy,pdl$zxy,xyg,"f_xy")
> pfyy <- gg1image2contours(xfg,yfg,ffg$fyy,pdg$zyy,pdl$zyy,xyg,"f_yy")
> pfxxx <- gg1image2contours(xfg,yfg,ffg$fxxx,pdg$zxxx,pdl$zxxx,xyg,"f_xxx")
> pfxxy <- gg1image2contours(xfg,yfg,ffg$fxxy,pdg$zxxy,pdl$zxxy,xyg,"f_xxy")
> pfxyy <- gg1image2contours(xfg,yfg,ffg$fxyy,pdg$zxyy,pdl$zxyy,xyg,"f_xyy")
> pfyyy <- gg1image2contours(xfg,yfg,ffg$fyyy,pdg$zyyy,pdl$zyyy,xyg,"f_yyy")
> ## t1 and t3 contain pure texts generated hidden in this Sweave file.
> ## t2 contains aas much of the symbolic computation output as possible:
> t2 <- print_f(f,df,3)
```

Now we use features of the gridExtra package to arrange all texts and plots:

```
> lay<-rbind(c( 1, 2, 3, 3),
           c( 4, 5, 3, 3),
           c( 6, 7, 8, 9),
           c(10,11,12,13))
```

```
> gg <- grid.arrange(grobs=gList(ggplotGrob(pf),t1,t2,ggplotGrob(pfx),ggplotGrob(pfy),
    ↪ ggplotGrob(pfxx),ggplotGrob(pfxy),ggplotGrob(pfyy),t3,ggplotGrob(pfxxx),ggplotGrob(
    ↪ pfxxy),ggplotGrob(pfxyy),ggplotGrob(pfyyy)),layout_matrix = lay)
```

For the resulting plot see figure **??**. They show a colored background image with two (a dashed green and a dotted blue) overlay of isolines. The colored background represents the exact function resp. its exact derivatives. Dashed green isolines are global bandwidth estimators, dotted blue isolines are local nearest neighbour estimates. All three overlays (colors and isolines) share the same step sizes for binning the colors and isoline levels.

Due to the nature of the different used functions only a varying part of the symbolic derivatives can be shown as text in the picture.

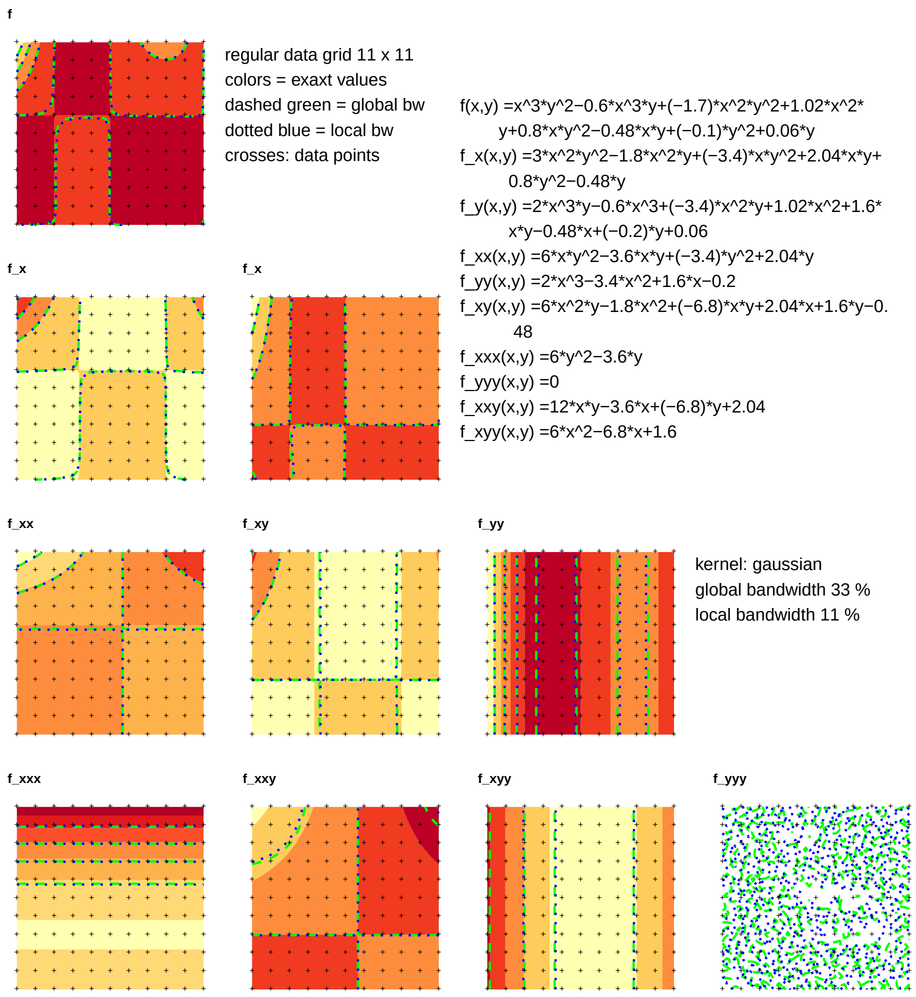


Figure 1: A bicubic polynomial and its derivatives, exact and estimated values, regular grid

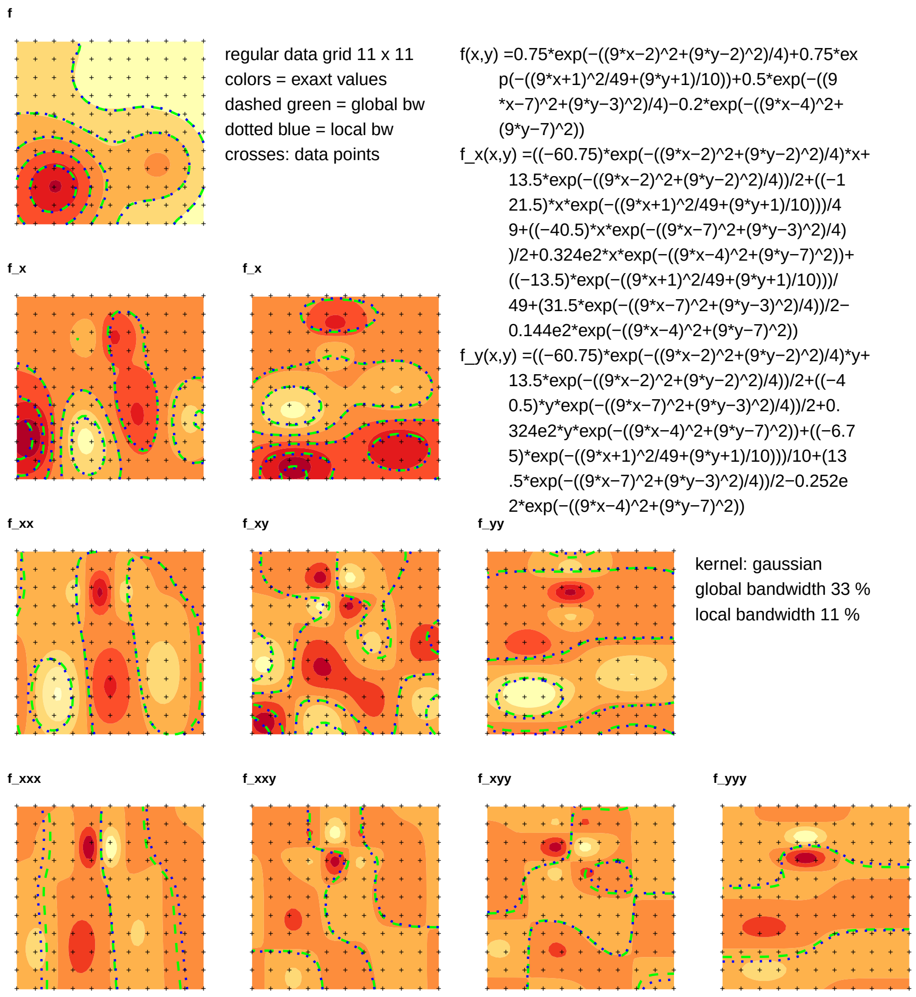
Now the same steps are repeated for Franke function 1:

```
> f <- function(x,y) 0.75*exp(-((9*x-2)^2+(9*y-2)^2)/4)+0.75*exp(-((9*x+1)^2)/49-(9*y+1)/10)
    ↪ +0.5*exp(-((9*x-7)^2+(9*y-3)^2)/4)-0.2*exp(-(9*x-4)^2-(9*y-7)^2)
> fg  <- outer(xg,yg,f)
> ffg <- fgrid(f,xfg,yfg,dg)
> df  <- derivs(f,dg)
```

Again estimate with global and local bandwidth

```
> ## global bw,
> pdg <- interp::locpoly(xg,yg,fg, input="grid", pd="all", h=c(bwg,bwg), solver="QR", degree
    ↪ =dg,kernel=knl,nx=af*ng,ny=af*ng)
> ## local bw:
> pdl <- interp::locpoly(xg,yg,fg, input="grid", pd="all", h=bwl, solver="QR", degree=dg,
    ↪ kernel=knl,nx=af*ng,ny=af*ng)
```

and repeat the plot. Technical details are now hidden and only the plot is shown as the commands above are more or less repeated. Results are shown in figure **??**. The same interpretation for colors and isolines as in the first plot is applied.



```
f(x,y) =0.75*exp(−((9*x−2)^2+(9*y−2)^2)/4)+0.75*ex
      p(−((9*x+1)^2/49+(9*y+1)/10))+0.5*exp(−((9
      *x−7)^2+(9*y−3)^2)/4)−0.2*exp(−((9*x−4)^2+
      (9*y−7)^2))
f_x(x,y) =((−60.75)*exp(−((9*x−2)^2+(9*y−2)^2)/4)*x+
       13.5*exp(−((9*x−2)^2+(9*y−2)^2)/4))/2+((−1
       21.5)*x*exp(−((9*x+1)^2/49+(9*y+1)/10)))/4
       9+((−40.5)*x*exp(−((9*x−7)^2+(9*y−3)^2)/4)
       )/2+0.324e2*x*exp(−((9*x−4)^2+(9*y−7)^2))+
       ((−13.5)*exp(−((9*x+1)^2/49+(9*y+1)/10)))/
       49+(31.5*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2−
       0.144e2*exp(−((9*x−4)^2+(9*y−7)^2))
f_y(x,y) =((−60.75)*exp(−((9*x−2)^2+(9*y−2)^2)/4)*y+
       13.5*exp(−((9*x−2)^2+(9*y−2)^2)/4))/2+((−4
       0.5)*y*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2+0.
       324e2*y*exp(−((9*x−4)^2+(9*y−7)^2))+((−6.7
       5)*exp(−((9*x+1)^2/49+(9*y+1)/10)))/10+(13
       .5*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2−0.252e
       2*exp(−((9*x−4)^2+(9*y−7)^2))
```

Figure 2: Franke function 1 and its derivatives, exact and estimated values, regular grid

7. Application To An Irregular Grid

Next we repeat the estmations with an irregular gridded data set using the same number of $11 \times 11 = 121$ points:

```
> n <- ng*ng
```

Start with the same polynomial as in the last section:

```
> f <- function(x,y) (x-0.5)*(x-0.2)*(y-0.6)*y*(x-1)
```

The kernel settings stay the same (`kernel=`"gaussian", global/local bandwidth 0.33/0.11).

```
> ## random irregular data
> x<-runif(n)
> y<-runif(n)
> xy<-data.frame(Var1=x,Var2=y)
> z <- f(x,y)
```

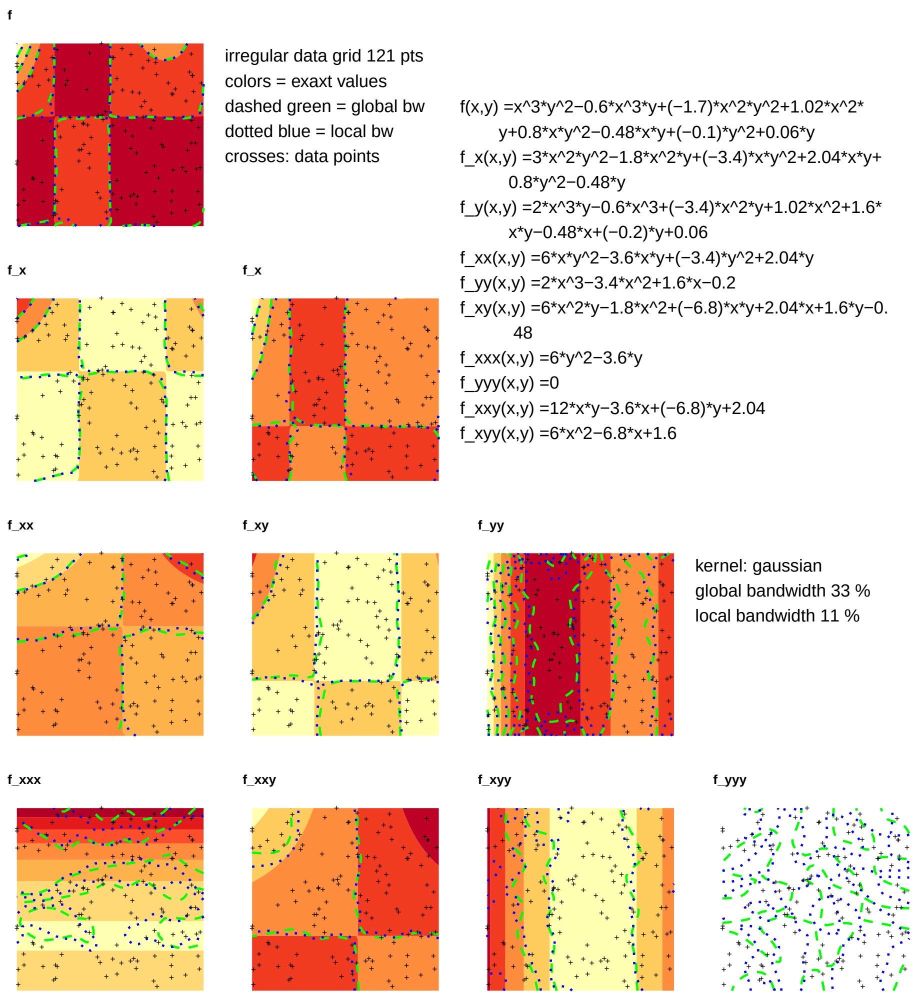
Again fill the grids for plotting the exact values

```
> ffg <- fgrid(f,xfg,yfg,dg)
> df <- derivs(f,dg)
```

and perform the estmation steps:

```
> ## global bandwidth
> pdg <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=c(bwg,bwg), solver="QR", degree=dg,
    ↪ kernel=knl)
> ## local bandwidth:
> pdl <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=bwl, solver="QR", degree=dg,kernel=knl)
```

The remaining steps to generate the plots are again similar to the first plot and therefore hidden. The output for the bicubic polynomial is shown in figure **??**, results for Franke function 1 in figure **??**. The results for Franke function 1 are shown in figure **??**.

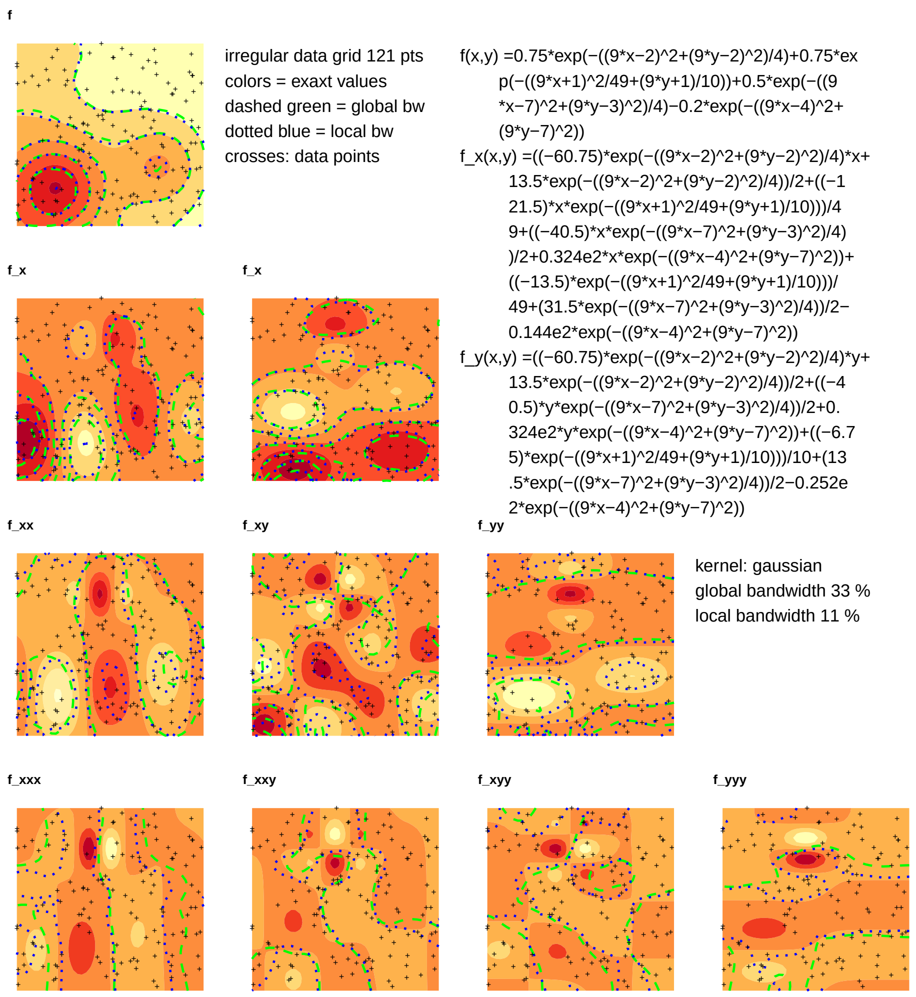


irregular data grid 121 pts
colors = exaxt values
dashed green = global bw
dotted blue = local bw
crosses: data points

```
f(x,y) =x^3*y^2−0.6*x^3*y+(−1.7)*x^2*y^2+1.02*x^2*
      y+0.8*x*y^2−0.48*x*y+(−0.1)*y^2+0.06*y
f_x(x,y) =3*x^2*y^2−1.8*x^2*y+(−3.4)*x*y^2+2.04*x*y+
      0.8*y^2−0.48*y
f_y(x,y) =2*x^3*y−0.6*x^3+(−3.4)*x^2*y+1.02*x^2+1.6*
      x*y−0.48*x+(−0.2)*y+0.06
f_xx(x,y) =6*x*y^2−3.6*x*y+(−3.4)*y^2+2.04*y
f_yy(x,y) =2*x^3−3.4*x^2+1.6*x−0.2
f_xy(x,y) =6*x^2*y−1.8*x^2+(−6.8)*x*y+2.04*x+1.6*y−0.
      48
f_xxx(x,y) =6*y^2−3.6*y
f_yyy(x,y) =0
f_xxy(x,y) =12*x*y−3.6*x+(−6.8)*y+2.04
f_xyy(x,y) =6*x^2−6.8*x+1.6
```

kernel: gaussian
global bandwidth 33 %
local bandwidth 11 %

Figure 3: A bicubic polynomial and its derivatives, exact and estimated, irregular data set



irregular data grid 121 pts
colors = exaxt values
dashed green = global bw
dotted blue = local bw
crosses: data points

```
f(x,y) =0.75*exp(−((9*x−2)^2+(9*y−2)^2)/4)+0.75*ex
    p(−((9*x+1)^2/49+(9*y+1)/10))+0.5*exp(−((9
    *x−7)^2+(9*y−3)^2)/4)−0.2*exp(−((9*x−4)^2+
    (9*y−7)^2))
f_x(x,y) =((−60.75)*exp(−((9*x−2)^2+(9*y−2)^2)/4)*x+
     13.5*exp(−((9*x−2)^2+(9*y−2)^2)/4))/2+((−1
     21.5)*x*exp(−((9*x+1)^2/49+(9*y+1)/10)))/4
     9+((−40.5)*x*exp(−((9*x−7)^2+(9*y−3)^2)/4)
     )/2+0.324e2*x*exp(−((9*x−4)^2+(9*y−7)^2))+
     ((−13.5)*exp(−((9*x+1)^2/49+(9*y+1)/10)))/
     49+(31.5*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2−
     0.144e2*exp(−((9*x−4)^2+(9*y−7)^2))
f_y(x,y) =((−60.75)*exp(−((9*x−2)^2+(9*y−2)^2)/4)*y+
     13.5*exp(−((9*x−2)^2+(9*y−2)^2)/4))/2+((−4
     0.5)*y*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2+0.
     324e2*y*exp(−((9*x−4)^2+(9*y−7)^2))+((−6.7
     5)*exp(−((9*x+1)^2/49+(9*y+1)/10)))/10+(13
     .5*exp(−((9*x−7)^2+(9*y−3)^2)/4))/2−0.252e
     2*exp(−((9*x−4)^2+(9*y−7)^2))
```

kernel: gaussian
global bandwidth 33 %
local bandwidth 11 %

Figure 4: Franke function 1 and its derivatives, exact and estimated, irregular data set

8. Different Kernels

Now we try different kernels. We just continue with Franke function 1 and the irregular gridded data from last section. We show the results of `kernel="uniform"` and `kernel="` ↪ `epanechnikov"` in figures ?? and ??.

```
> ## global bandwidth:
> pdg <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=c(bwg,bwg), solver="QR", degree=dg,
    ↪ kernel="uniform")
> ## local bandwidth:
> pdl <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=bwl, solver="QR", degree=dg,kernel="
    ↪ uniform")
```

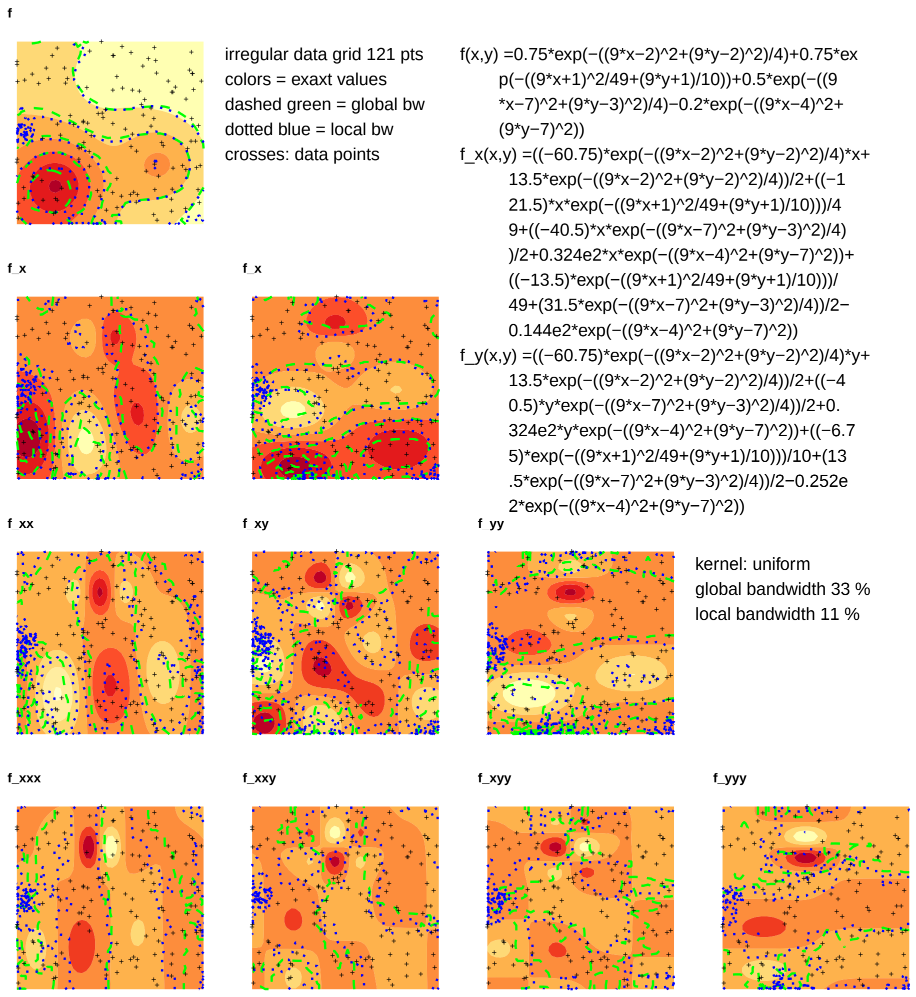


Figure 5: Franke function 1 and its derivatives, uniform kernel

```
> ## global bandwidth:
> pdg <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=c(bwg,bwg), solver="QR", degree=dg,
    ↪ kernel="epanechnikov")
> ## local bandwidth:
> pdl <- interp::locpoly(x,y,z, xfg,yfg, pd="all", h=bwl, solver="QR", degree=dg,kernel="
    ↪ epanechnikov")
```

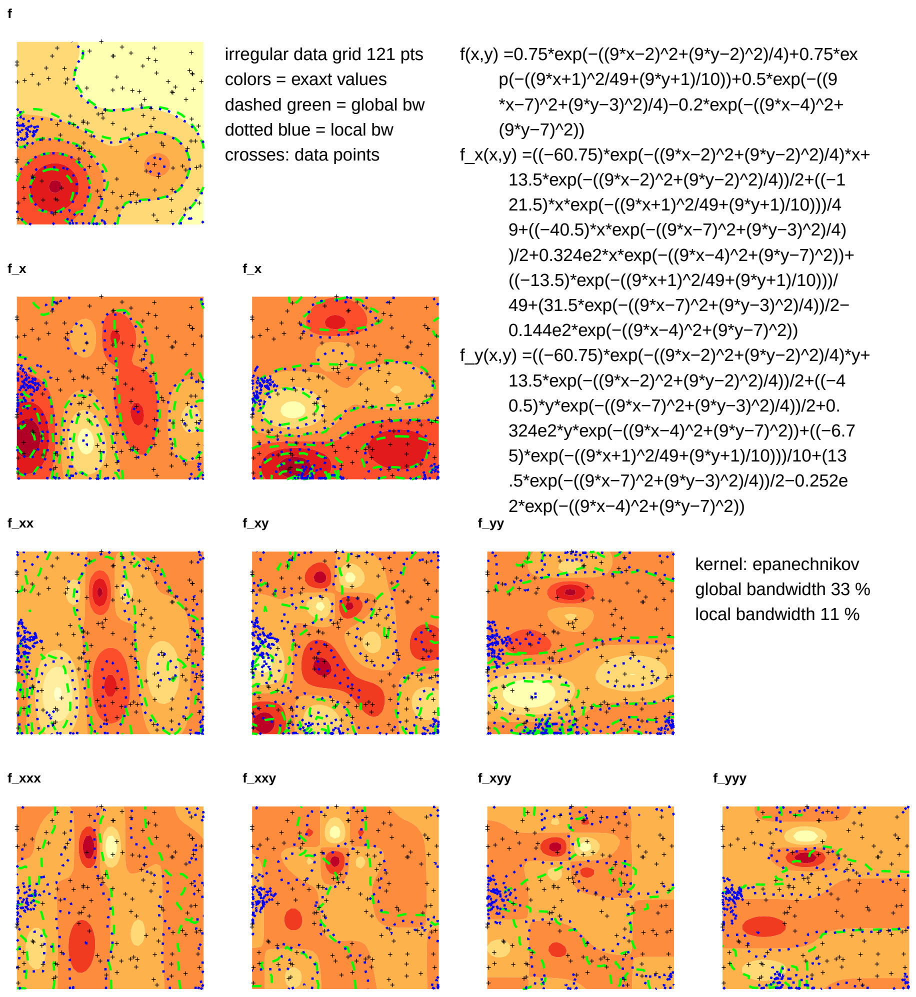


Figure 6: Franke function 1 and its derivatives, epanechnikov kernel

Especially the performance of the uniform kernel with its discontinuous behavior at the borders of its support drops visibly. Globally spoken, the local bandwidth estimators capture more details, across all kernels. But combined with a kernel with bounded support (uniform or epanechnikov in the test) they show problems at the border of the region. So the default setting of a gaussian kernel is well founded.

9. Appendix

These helper functions are needed to convert between R and Yacas:

```
> # helper functions for translation between R and Yacas
> fn_y  <- function(f){
     b <- toString(as.expression(body(f)))
     b <- stringr::str_replace_all(b,"cos","Cos")
     b <- stringr::str_replace_all(b,"sin","Sin")
     b <- stringr::str_replace_all(b,"exp","Exp")
     b <- stringr::str_replace_all(b,"log","Log")
     b <- stringr::str_replace_all(b,"sqrt","Sqrt")
     b
 }
> ys_fn  <- function(f){
     f <- stringr::str_replace_all(f,"Cos","cos")
     f <- stringr::str_replace_all(f,"Sin","sin")
     f <- stringr::str_replace_all(f,"Exp","exp")
     f <- stringr::str_replace_all(f,"Log","log")
     f <- stringr::str_replace_all(f,"Sqrt","sqrt")
     f
 }
```

This function applies symbolic derivatives to a R function, both for later use as R function (via **Deriv**) and for printing (via **Ryacas**).

```
> derivs <- function(f,dg){
     ret<-list(f=f,
               f_str=ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),""),")"))))

     if(dg>0){

         ret$fx <- function(x,y){
             myfx <- Deriv(f,"x");
             tmp <- myfx(x,y);
             if(length(tmp)==1)
                 return(rep(tmp,length(x)))
             else
                 return(tmp)
         }
         ret$fx_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)"),")")))


         ret$fy <- function(x,y){
             myfy <- Deriv(f,"y");
             tmp <- myfy(x,y);
             if(length(tmp)==1)
                 return(rep(tmp,length(x)))
             else
                 return(tmp)
         }
         ret$fy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(y)"),")")))


         if(dg>1){
             ret$fxy <- function(x,y){
                 myfxy <- Deriv(Deriv(f,"y"),"x");
                 tmp <- myfxy(x,y);
                 if(length(tmp)==1)
                     return(rep(tmp,length(x)))
                 else
                     return(tmp)
             }
```

```
        ret$fxy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)D(y)"),")")))

        ret$fxx <- function(x,y){
            myfxx <- Deriv(Deriv(f,"x"),"x");
            tmp <- myfxx(x,y);
            if(length(tmp)==1)
                return(rep(tmp,length(x)))
            else
                return(tmp)
        }
        ret$fxx_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)D(x)"),")")))

        ret$fyy <- function(x,y){
            myfyy <- Deriv(Deriv(f,"y"),"y");
            tmp <- myfyy(x,y);
            if(length(tmp)==1)
                return(rep(tmp,length(x)))
            else
                return(tmp)
        }
        ret$fyy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(y)D(y)"),")")))

        if(dg>2){
            ret$fxxy <- function(x,y){
                myfxxy <- Deriv(Deriv(Deriv(f,"y"),"x"),"x");
                tmp <- myfxxy(x,y);
                if(length(tmp)==1)
                    return(rep(tmp,length(x)))
                else
                    return(tmp)
            }
            ret$fxxy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)D(x)D(y)")
                ↪ ,")")))

            ret$fxyy <- function(x,y){
                myfxyy <- Deriv(Deriv(Deriv(f,"y"),"y"),"x");
                tmp <- myfxyy(x,y);
                if(length(tmp)==1)
                    return(rep(tmp,length(x)))
                else
                    return(tmp)
            }
            ret$fxyy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)D(y)D(y)")
                ↪ ,")")))

            ret$fxxx <- function(x,y){
                myfxxx <- Deriv(Deriv(Deriv(f,"x"),"x"),"x");
                tmp <- myfxxx(x,y);
                if(length(tmp)==1)
                    return(rep(tmp,length(x)))
                else
                    return(tmp)
            }
            ret$fxxx_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(x)D(x)D(x)")
                ↪ ,")")))

            ret$fyyy <- function(x,y){
                myfyyy <- Deriv(Deriv(Deriv(f,"y"),"y"),"y");
                tmp <- myfyyy(x,y);
                if(length(tmp)==1)
                    return(rep(tmp,length(x)))
                else
                    return(tmp)
            }
            ret$fyyy_str  <- ys_fn(yac(paste("Simplify(",y_fn(fn_y(f),"D(y)D(y)D(y)")
                ↪ ,")")))
        }
        }
    }
    ret
}
```

The next function calculates exact values of the given function on a grid and fills it with partial derivatives up to degree `dg`.

```
> # for plots of exact values
> fgrid <- function(f,xg,yg,dg){
   ret <- list(f=outer(xg,yg,f))
   df <- derivs(f,dg)
   if(dg>0){
     ret$fx  <- outer(xg,yg,df$fx)
     ret$fy  <- outer(xg,yg,df$fy)
     if(dg>1){
       ret$fxy <- outer(xg,yg,df$fxy)
       ret$fxx <- outer(xg,yg,df$fxx)
       ret$fyy <- outer(xg,yg,df$fyy)
       if(dg>2){
         ret$fxxy <- outer(xg,yg,df$fxxy)
         ret$fxyy <- outer(xg,yg,df$fxyy)
         ret$fxxx <- outer(xg,yg,df$fxxx)
         ret$fyyy <- outer(xg,yg,df$fyyy)
       }
     }
   }
   ret
 }
```

Another helper function for formatting function expressions in the plots:

```
> split_str <- function(txt,l){
   start <- seq(1, nchar(txt), l)
   stop <- seq(l, nchar(txt)+l, l)[1:length(start)]
   substring(txt, start, stop)
 }
```

The combination of image and contour plots are generated by these functions:

```
> grid2df <- function(x,y,z)
     subset(data.frame(x = rep(x, nrow(z)),
                       y = rep(y, each = ncol(z)),
                       z = as.numeric(z)),
            !is.na(z))
> gg1image2contours <- function(x,y,z1,z2,z3,xyg,ttl=""){
     breaks <- pretty(seq(min(z1,na.rm=T),max(z1,na.rm=T),length=11))
     griddf1 <- grid2df(x,y,z1)
     griddf2 <- grid2df(x,y,z2)
     griddf3 <- grid2df(x,y,z3)
     griddf  <- data.frame(x=griddf1$x,y=griddf1$y,z1=griddf1$z,z2=griddf2$z,z3=griddf3$z)
     ggplot(griddf, aes(x=x, y=y, z = z1)) +
        ggtitle(ttl) +
        theme(plot.title = element_text(size = 6, face = "bold"),
              axis.line=element_blank(),axis.text.x=element_blank(),
              axis.text.y=element_blank(),axis.ticks=element_blank(),
              axis.title.x=element_blank(),
              axis.title.y=element_blank(),legend.position="none",
              panel.background=element_blank(),panel.border=element_blank(),panel.grid.
                  ↪ major=element_blank(),
              panel.grid.minor=element_blank(),plot.background=element_blank()) +
        geom_contour_filled(breaks=breaks) +
        scale_fill_brewer(palette = "YlOrRd") +
        geom_contour(aes(z=z2),breaks=breaks,color="green",lty="dashed",lwd=0.5) +
        geom_contour(aes(z=z3),breaks=breaks,color="blue",lty="dotted",lwd=0.5) +
        theme(legend.position="none") +
        geom_point(data=xyg, aes(x=Var1,y=Var2), inherit.aes = FALSE,size=1,pch="+")
}
```

The expressions for the functions and their derivatives are printed via:

```
> print_deriv <- function(txt,l,at=42){
     ret<-""
     for(t in txt){
         if(stringi::stri_length(t)<at)
             btxt <- t
         else
             btxt <- split_str(t,at)
         ftxt <- rep(paste(rep(" ",stringi::stri_length(l)),sep="",collapse=""),length(btxt)
             ↪ )
         ftxt[1] <- l
         ret <- paste(ret,paste(ftxt,btxt,sep="",collapse = "\n"),sep="",collapse = "\n")
     }
     ret
 }
> print_f <- function(f,df,dg,offset=0.8){
   lns <- c(print_deriv(df$f_str,"f(x,y) ="))
   if(dg>=1)
     lns <- c(lns,
     print_deriv(df$fx_str,"f_x(x,y) ="),
     print_deriv(df$fy_str,"f_y(x,y) ="))
   if(dg>=2)
     lns <- c(lns,
     print_deriv(df$fxx_str,"f_xx(x,y) ="),
     print_deriv(df$fyy_str,"f_yy(x,y) ="),
     print_deriv(df$fxy_str,"f_xy(x,y) ="))
   if(dg>=3)
     lns <- c(lns,
     print_deriv(df$fxxx_str,"f_xxx(x,y) ="),
     print_deriv(df$fyyy_str,"f_yyy(x,y) ="),
     print_deriv(df$fxxy_str,"f_xxy(x,y) ="),
     print_deriv(df$fxyy_str,"f_xyy(x,y) ="))
   txt <- grid.text(paste(lns,
     collapse="\n"),gp=gpar(fontsize=8),
     x=0,y=offset,draw=FALSE,
     just = c("left","top"))
   txt
 }
```

List of Tables

List of Figures

Affiliation:

Albrecht Gebhardt Institut für Statistik
Universität Klagenfurt 9020 Klagenfurt, Austria
E-mail: `albrecht.gebhardt@aau.at`