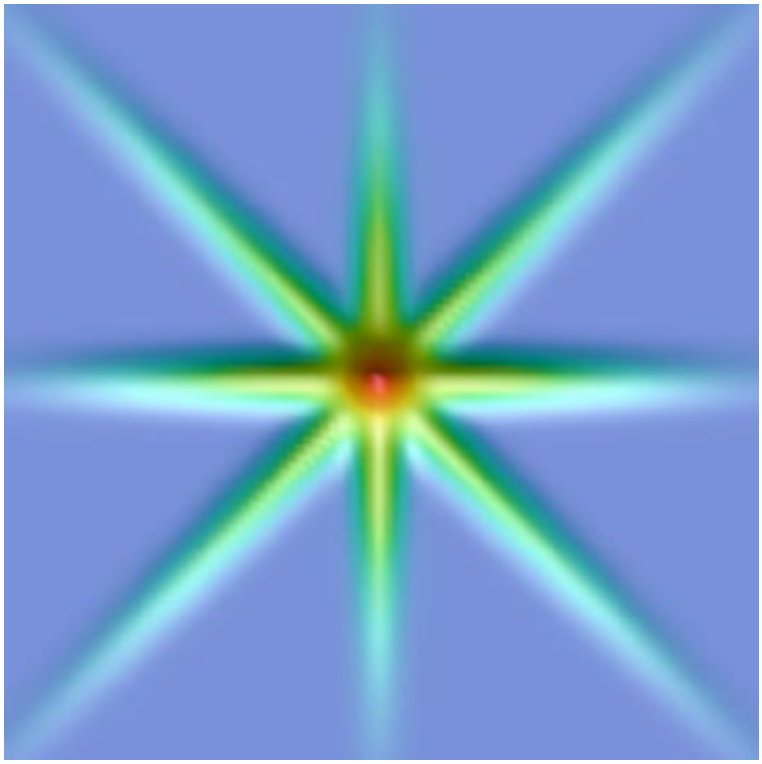
PARDENS - an experimental program for Parzen density fitting

Rafał Wójcik and Paul Torfs

January, 2003



Contents

1 Introduction

This document outlines the operation and the available options of the program PARDENS. The main purpose of the program is to fit a Parzen (or optionally a mixture) probability density model of multivariate normal components to the user-provided data set. Estimation procedure is based on penalized version of log likelihood maximization. The penalty term incorporates the Kullback-Leilbler divergence from the global Gaussian fitted to data (for a technical description of the algorithm see Torfs and Wójcik, 2003). Many other facilities are implemented in the program, that we found to be of use when modelling Parzen densities.

2 Copyright

3 Obtaining the program and seeking-assistance

Copies of PARDENS are available upon request from the authors. Bug reports, code contributions and questions concerning the methodology presented in this document should be mailed to:

`Raf.Wojcik@wur.nl` or `Paul.Torfs@wur.nl`

NOTE: `PAR`zen `DENS`ity fitting remains a difficult business, in particular for high-dimensional data. You may be assured that when treating the program as a black box, you are volnurable to get nonsensical results. Even if you know what you are doing, output produced by the program has to be interpreted very carefully. Now that you are warned, go ahead and have fun!

4 The graphical user interface

While interacting with the program, the user is confronted with a graphical interface as shown in Fig. 1. The interface is written with the `fltk` library (visit `http://www.fltk.org` for more information). Command lines are en-

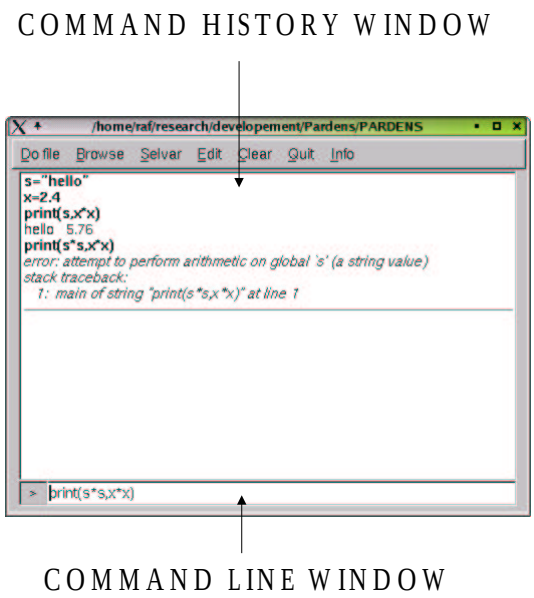


Figure 1: The interface.

tered in the "command line" window. After hitting the Enter-key, these commands, and the responses of the program obtained by executing them, are written to the "command-history" window. The echos of the command lines are written in bold face, the "regular" answers in normal face, the error messages in italic.

4.1 Re-entering commands

Lines from the "command-history" window can be re-entered to the "command-line" window by left-clicking them (they can then be further edited), or by right-clicking, in which case they are immediately executed.

4.2 LUA

The program interpretes the commands through (an extension) of the "LUA" language. At the web site of the LUA-project :

`http://www.tecgraf.puc-rio.br/lua/`

a complete (less then 50 pages) manual can be downloaded. Some simple characteristics of the language are already demonstrated in Fig. 1 :

- variables of simple types (strings, numbers) do not have to be declared, but can immediately be used
- all standard computations are implemented
- when performing un-allowed commands, error messages are produced

All basic libraries of LUA (string manipulation, mathematical functions, I/O facilities, system facilities) have been implemented, so that all standard items of a complete programming language are available. Moreover, there are some functions available which make LUA a useful batch processing program. The most important is the:

`execute(command)`

function. This passes commands to be executed to the operating shell.

4.2.1 Multiline commands

For executing multi-line commands, the command line window is not sufficient, as each line is immediately executed after entering. For that reason, a simple multiline editor is built in. The multiline editor is started by executing the "<" in the command line window. All command lines that are entered afterwards, are not immediately executed, but buffered, and executed as a complete buffer after the ending of the macro by entering the ">" in the command line (see Fig. 2). In the multiline-entering phase, the multiline indicator will light up. The example in Fig. 2 above also shows one of the basic control structures in the LUA language: the for-loop. Figure 3 depicts yet another typical use of the multiline command - that of the definition of a function. In this function, the use of the LUA "if" structure is demonstrated.

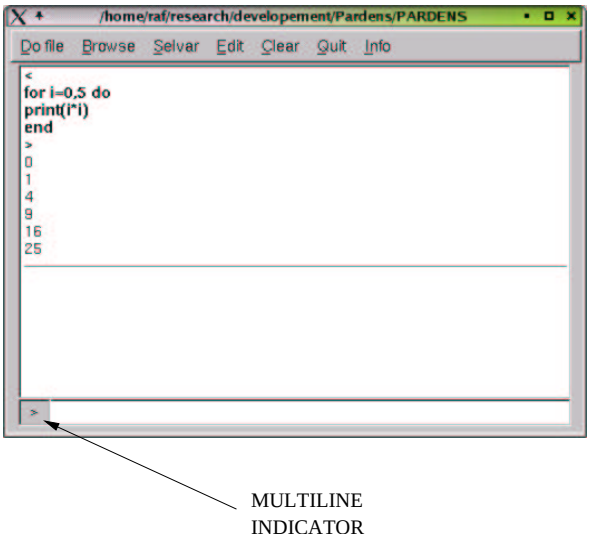


Figure 2: A typical multiline command: a for loop.

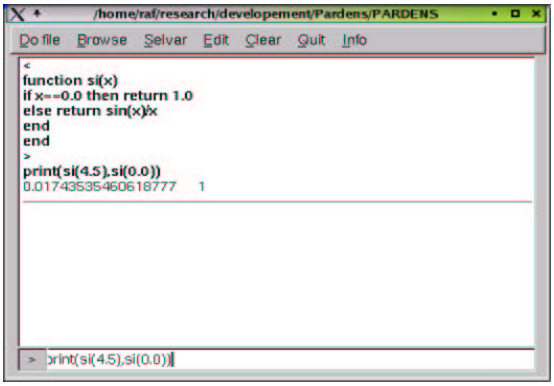


Figure 3: A typical multiline command: the definition of a function.

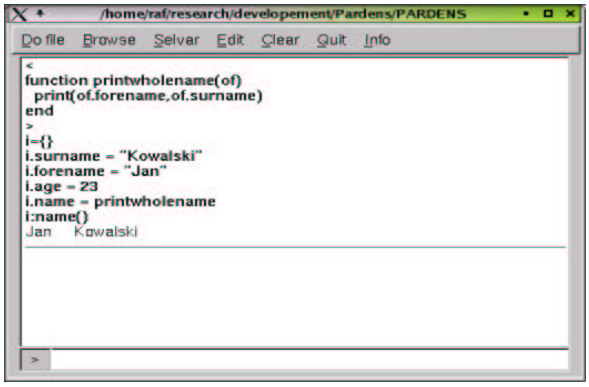


Figure 4: An object oriented example.

4.2.2 LUA and objects

The LUA language has an object oriented flavor. This means that variables can have data fields and methods (or functions acting on those data fields). Figure 4 shows an implementation of an object. First, by using the multiline command facility, a LUA function is defined. Next by the statement `i = {}` a standard LUA object called `TABLE` is created. By `i.surname = ''Kowalski''`, a field with name `surname` and with data being the string `''Kowalski''` is created. An object can have several fields of different types. A field of type "function" is called a method of the object. In the example the object `i` has a method with name `name` and value `printwholename`. The last command illustrates how a method is called. The line `i:name()` is interpreted as `i.name(self)`, thus providing the called function with the object as first argument.

4.2.3 LUA extensions

LUA is a language which can easily be extended by new functions and objects (written in C++). One type of object that is included in the program (and has been added to the LUA core), is the `MACRO` type. A `MACRO` is a collection of (command) lines. A variable is declared to be of the `MACRO` type by:

```
f = MACRO
```

Command lines can be put into the macro by the multiline editing facility, as shown in Fig. 5. Actually, all objects which have a `fread` method can be given values in the same way. Objects of type `MACRO` have the following methods:

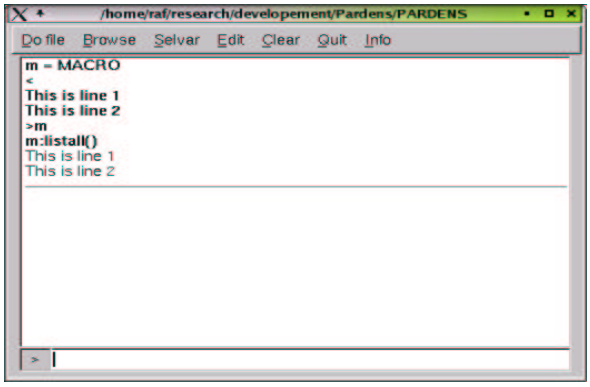


Figure 5: Declaring a macro, filling it with the multiline facility, and calling the method "listall".

`listall()`	lists all lines
`fread(STRING filename)`	reads all lines from a file with name `filename`
`fwrite(STRING filename)`	writes all lines to a file with name `filename`
`exec()`	executes all lines

4.3 The menu

Menu items can be activated by clicking them, or by pressing th `ALT`-key and the corresponding underlined letter.

4.3.1 Do file

Collection of command lines can be stored in files. The "`Dofile`" opens a file selector (see Fig. 6). Once a file is selected, all lines in that file are executed as if they were entered in the command line. By default, the files with extension "`lio`" are shown first.

4.3.2 Browse

If one wants to execute only a few command lines from a file (unlike in the "`Do file`" menu where all the lines are executed), one can use the "`Browser`" option of the menu. After selecting a file with the file selector (see also the "`Do file`" menu) a browser window as in Fig. 7 will pop up. From this window one can select lines to the command line window in the same way as

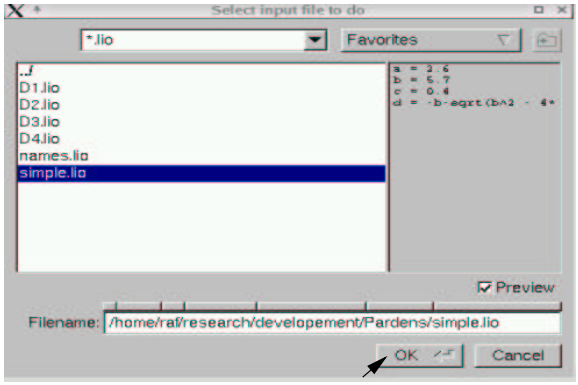


Figure 6: A file selector.

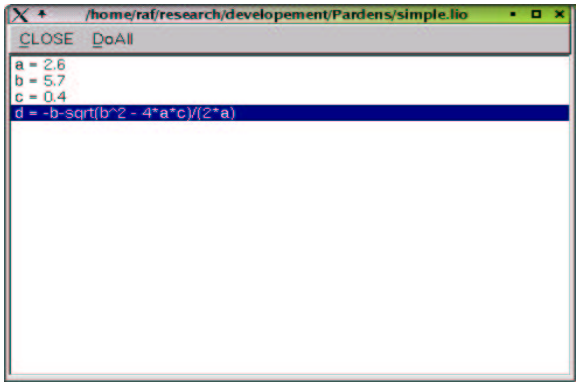


Figure 7: A typical browser window.

from the command history window. The "`Close`" button closes the browser window.

4.3.3 Clear

Clears the command history window.

4.3.4 Selvar

Selvar menu makes it possible for the user to list variables declared in a particular LUA script. Three options to do this listing are available (see Fig. 8): "`all vars`" button displays all the variables currently used by the program, "`value vars`" button displays only variables that are numbers (integers, floating point numbers) and finally "`edit vars`" button displays so-called

edit variables. These variables are objects that have "fread" and "fwrite" as

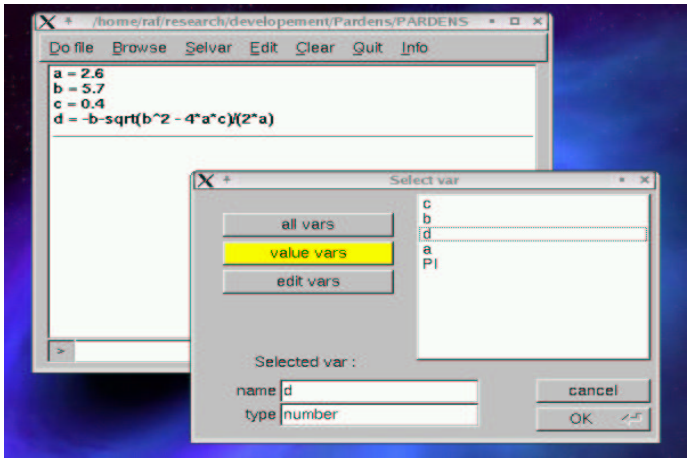


Figure 8: A SelVar menu.

methods:

- `foo:fread(filename)` reads the content of the file into the object
- `foo:fwrite(filename)` writes the content of the file into the object

Clicking a variable of interest in the right-hand side scroll-window shows the name and the type of that variable (two bottom panels in Fig. 8). Selvar menu is particularly helpful in keeping hold on administration of large LUA programs.

4.3.5 Edit

The content of LUA files or edit variables can be altered via Edit menu. Fig 9 shows an edit window which appears after choosing a "file" type option. This edit window has some classical edit facilities so the content of a file can easily be changed and saved. The same kind of operations can be performed on edit variables. Figure 10 shows this for an object of the `MACRO` type (see also Section 4.2.3). In the variable "`sidef`" of type `MACRO`, a function definition is stored by the multiline facility (see Section 4.2.1). Corrections (e.g. changing the "*" in the function definition) however are not possible in the multiline environment. In this case the Edit menu should be used. After selecting "`sidef`" variable to be edited, and clicking Search and then Replace options in edit window, the "*" can be replaced by "\". By pressing Save menu item the (changed) content of the edit window is read into the "`sidef`" variable (by the `fread` method).

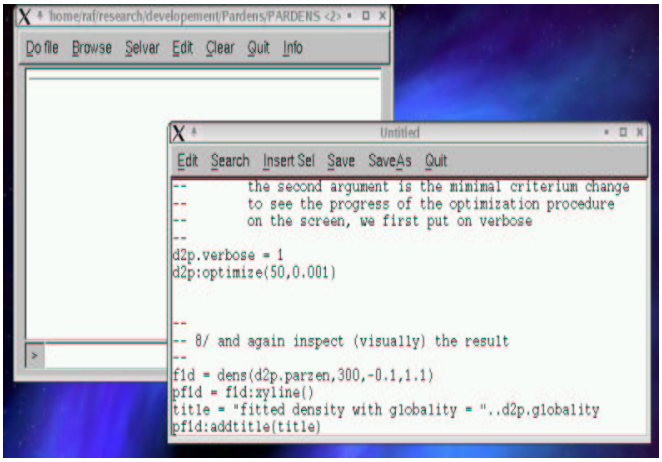


Figure 9: An edit window.

4.3.6 Quit

The quit button quits the program.

4.3.7 Info

The info button displays the information about the program.

5 The interface to GNUPLOT

A remark

The material presented here (and further in Section 6.1.3) is optional. The interface to GNUPLOT plotting package described below was rather developed to get a poor men's look at the results produced by PARDENS during long debugging sessions than to obtain sophisticated graphical representation of data. Since it is easy to store output from PARDENS in text ASCII files, we strongly encourage the user unfamiliar with or annoyed by GNUPLOT to visualize the output with any plotting software she/he feels comfortable with.

To provide a simple plotting facility, an object type `GNUMACRO` was developed. The content of a variable of this type is just a number of lines (as in the case

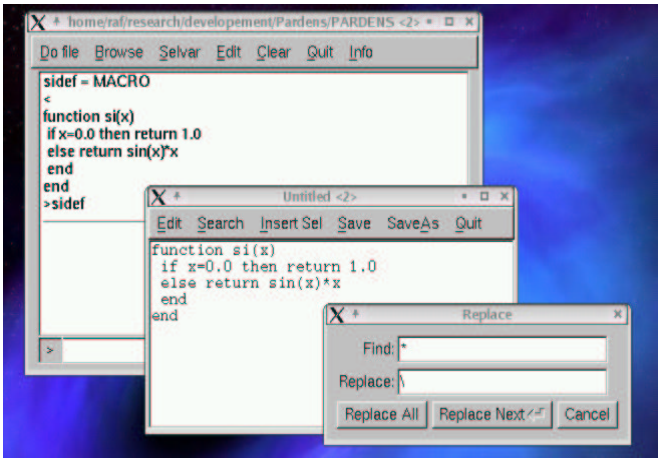


Figure 10: Editing a macro.

of more general `MACRO` type of section 4.2.3). Those lines are then send by various methods of this object to the GNUPLOT interpreter. GNUPLOT is a general plotting program, available in the public domain. From the website `http://www.gnuplot.info`, executables, manuals and source code can be downloaded.

In order to make `GNUMACRO`s function properly, the `gnuplot` program for UNIX-based systems, or the `wgnuplot` program for WINDOWS-based systems, should be callable by the interface.

For the `GNUMACRO` type there are several available

- methods:

`listall()`	lists all lines
`fread(STRING filename)`	reads all lines from `filename`
`fwrite(STRING filename)`	writes all lines to `filename`
`show()`	shows the plotting results in a window
`exec()`	the same as `show()`
`eps(STRING filename)`	plots in eps-format to `filename.eps`
`ps(STRING filename)`	plots in ps-format to `filename.ps`
`fig(STRING filename)`	plots in fig-format to `filename.fig` [1]

[1] The fig-format is the format used by an excellent UNIX-drawing program Xfig (see `http://www.xfig.org` for more information).

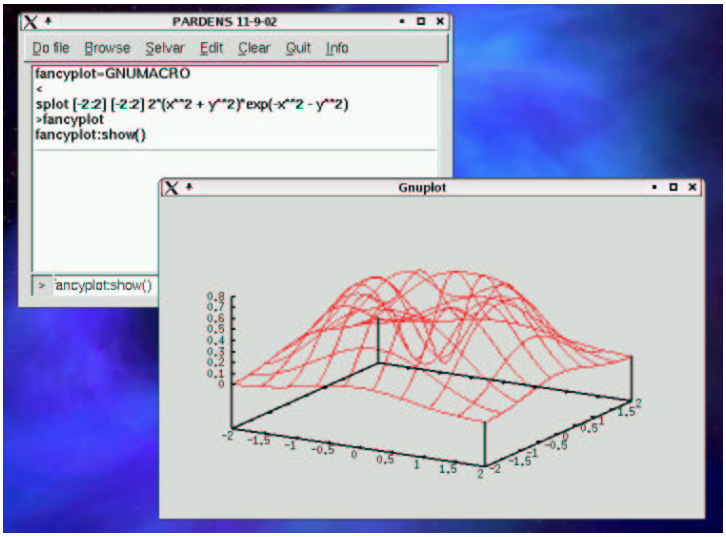


Figure 11: A typical plot made by a `GNUMACRO`.

Figure 11 illustrates a simple use of this object. The methods `fread` and `fwrite` make it also possible to use the `Edit` facility of the menu (see Section 4.3.5).

6 Data types

6.1 Vectors and matrices

For mathematical convenience, the object types `VECTOR` and `MATRIX` have been added to the program. The `VECTOR` class has the following interface:

- setfields:

`length`	`INT` length of a vector

- methods:

`fread(STRING filename)`	reads all elements of a vector from a file with name `filename`
`fwrite(STRING filename)`	writes all elements of a vector to a file with name `filename`
`listall()`	lists all elements of a vector

`listfromto(INT e1,INT e2)`	lists elements of a vector with indices ranging from `e1` to `e2`.
`showmode()`	shows mode of a vector

The interface to the `MATRIX` class reads:

- setfields:

`nrows`	`INT` number of rows of a matrix
`ncols`	`INT` number of columns of a matrix

- methods:

`fread(STRING filename)`	reads all elements of a matrix from a file with name `filename`
`fwrite(STRING filename)`	writes all elements of a matrix to a file with name `filename`
`listall()`	lists all elements of a matrix
`listfromto(INT r1, INT r2, INT c1, INT c2)`	lists elements of sub-matrix of a matrix; the sub-matrix ranges from row `r1` to `r2` and column `c1` to `c2` of the original matrix
`showmode()`	shows mode of a matrix

It is important to note that fields of both `VECTOR` and `MATRIX` classes are so-called *setfields*. This means that the user must specify values of those fields in order to use a class properly. Sometimes the values of fields are automatically found by a class itself. Therefore, the user cannot explicitely set those values and is only able to get them by i.e. printing them out. Such fields are said to be *getfields* (see Section 6.2.1 and 6.2.2 for examples of classes with *getfields*).

6.1.1 Defining vectors and matrices

The code below (file vecmat.lio) creates a vector of length 3 and a matrix with 3 rows and 2 columns, and stores some values in those objects.

```
1   -- define a vector
2
3   v = VECTOR
```

```
4   v.length = 3
5   print("length of v =",v.length)
6   v[0] = 2.3
7   v[1] = 3.4
8   v[2] = 5.6
9
10  -- define a matrix
11
12  m = MATRIX
13  m.numrows = 3
14  m.numcols = 2
15  print("size of m  =",m.numrows," by ",m.numcols)
16  m[0][0] = 2.3
```

Note that for a vector indices must be in the range `0,...,N-1` where `N` is the length of the vector and for a matrix row's and column's indices must be in range `0,...,Nr-1` and `0,...,Nc-1` respectively where `Nr` is the number of rows and `Nc` is the number of columns. Both for vectors and matrices the default value of all the elements is 0. Note also that the statement m[1] produces a vector of length m.numcols

As mentioned earlier both the `VECTOR` and the `MATRIX` objects have the `fread` and `fwrite` methods. This makes the multiline option of Section 4.2.1 and the edit options of Section 4.3.5 available. Note that the `fread` method tries to read as many entries as possible. The methods `listall` and `listfromto` are used to show the content of a vector or of a matrix in the history window.

6.1.2 Modes

Sometimes, it does not make sense to change the dimensions of a matrix or vector (especially when they depend on some other parameters). In that case, they are said to be in `EDIT` mode. This is e.g. the case for the vector `m[r]` of a matrix `m`. The length of this vector should remain equal to the number of rows of the matrix. Sometimes, it does not make sense to change the values of a vector or a matrix. In that case, they are said to be in `VIEW` mode. When both the dimensions and the values may change, the vector or matrix is said to be in `ALL` mode. The method `showmode()` shows the mode of a vector or matrix.

6.1.3 Plotting through GNUPLOT

When `v` is a `VECTOR`, a command "`picture = v:plot()`" assigns to `picture` a `GNUMACRO` (see Section 5) containing a plot of the values of `v`. If needed,

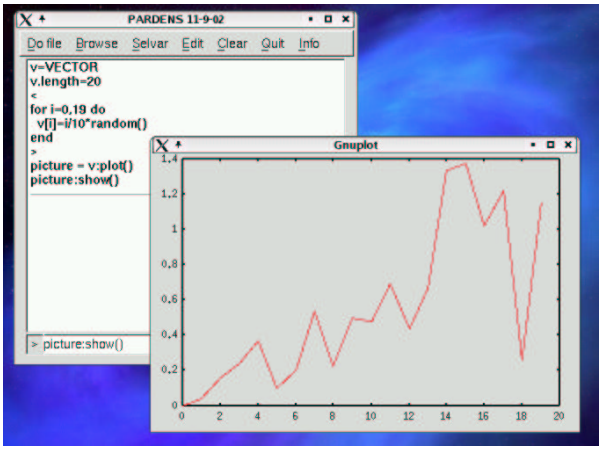


Figure 12: A plot of a vector.

the variable `picture` can further be edited (see Section 4.3.5) to add some other GNUPLOT commands. With the help of those commands the user can e.g. change the form of the plot, add titles etc.

For a matrix, the following plotting commands (see also Fig. 13) are available:

1. `surfplot()`: Interpretes the values of the matrix as z-values. A surface is plotted with row numbers and column numbers on the x and y-axis.

2. `xysurfplot(VECTOR x, VECTOR y)`: Same as above, but now with the values of the x-vector on the x-axis and the values of the y-vector on the y-axis.

3. `contplot()`: Interpretes the values of the matrix as z-values. Contours are plotted with row numbers and column numbers on the x and y-axis.

4. `xycontplot(VECTOR x, VECTOR y)`: Same as above, but now with the values of the x-vector on the x-axis and the values of the y-vector on the y-axis.

5. `colplot()`: The different columns are plotted against the row number.

6. `xyline()`: The first column is interpreted as x-values. The other columns are interpreted as y-values and are plotted against the x-values, connected by lines.

7. `xyscatter()`: Same as above, except the points are not connected by lines, but represented with various symbols.

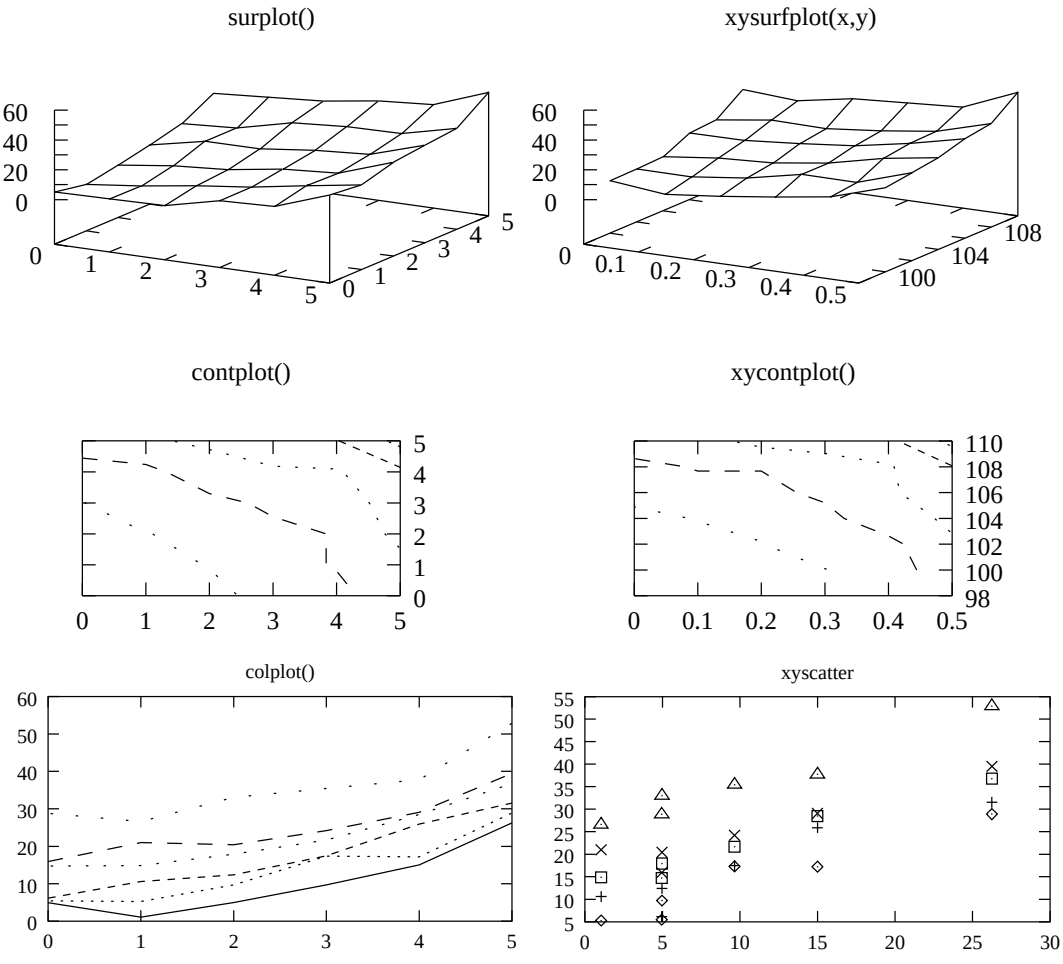


Figure 13: Different plots of a matrix.

6.2 Parzen density classes

In this section we describe three classes for dealing with the Parzen densities: PARZEN, D2P and XYPARZEN. For the sake of clarity it is useful here to

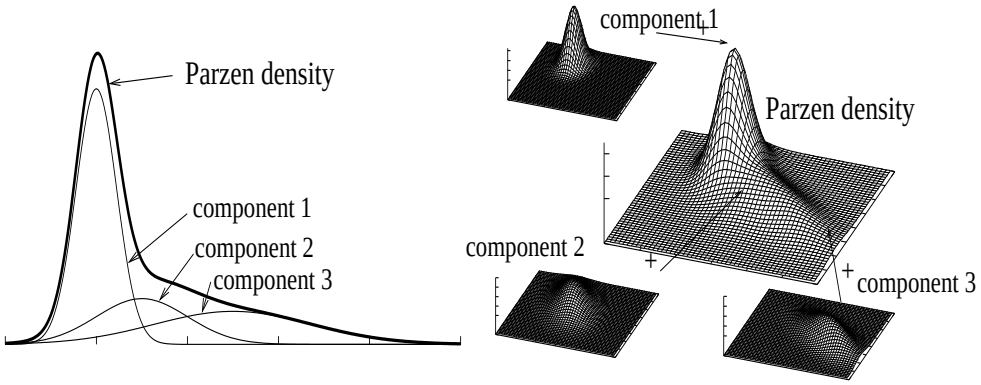


Figure 14: A one and two dimensional example of a Parzen density. In both cases the density is the sum of three Gaussian components.

introduce some basic theory of Parzen models. In a Parzen model (Parzen, 1962), a probability density function is expressed as a linear combination of component densities or simply components (see Fig.14). A model with Nc components is written in the form:

$$f_{\mathcal{P}}(\mathbf{x}) = \sum_{n=1}^{N_c \leqslant N} w_n \, f(\mathbf{x}|n) \tag{1}$$

where $w_1, \ldots, w_{N_c}$ are called the weights and the parameters of the components $f(\mathbf{x}|n)$ typically vary with n. Note that in statistical literature (see e.g. McLachlan and Peel, 2000) (1) is sometimes referred to as the mixture density model if N_c is smaller than the number of data points N to which the model is to be fitted. By constraining the weights:

$$\sum_{n=1}^{N_c \leqslant N} w_n = 1 \tag{2}$$

$$0.0 \leq w_n \leq 1.0 \tag{3}$$

and choosing normalized components:

$$\int f(\mathbf{x}|n)dx = 1 \tag{4}$$

guarantees that (1) does represent the density function.

Having $f_{\mathcal{P}}$ defined it only remains to decide on the form of the components. Apart from numerous other possibilities (see e.g. Epanechnikov components in Silverman, 1986) in PARDENS there is only one option available, which is a Gaussian density with full covariance matrix:

$$f(\mathbf{x}|n) = f_{\mathcal{N}(\mu_n,\mathcal{C}_n)}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N \, |\mathcal{C}_n|}} \; \exp\left(\frac{-||\mathbf{x}-\mu_n||^2_{\mathcal{C}_n}}{2}\right) \tag{5}$$

where

$$||\mathbf{x}-\mu_n||^2_{\mathcal{C}_n} = (\mathbf{x}-\mu_n)\,{\mathcal{C}_n}^{-1}\,(\mathbf{x}-\mu_n)^T \tag{6}$$

and μ_n is given mean vector and $\mathcal{C}_n$ is given covariance matrix.

6.2.1 PARZEN class

PARZEN class was designed to construct Parzen density models out of given parameters (number of component densities, their weights, means and covariances). So this class does not implement any fitting procedure and as such cannot be used to fit a Parzen density to the user-provided data sets. The interface to the PARZEN class reads:

- getfields:

  `dim` — `INT` dimension of the Parzen density function

  `numcomponents` — `INT` number of component density functions

- methods:

  `simulate(INT numsim)` — samples `numsim` points from a Parzen density; the result is `dim` × `numsim` `MATRIX`

  `density(VECTOR x)` — evaluates a Parzen density function at a given point `x`; the result is a `DOUBLE` number

  `addcomponent(VECTOR mean, MATRIX covariance, DOUBLE weight)` — adds a component parameterized by the `mean`, `covariance` and the unnormalized `weight` to a Parzen density

`moments()`	calculates first and second order moments of a Parzen density; the result is a pair consisting of mean `VECTOR` and the covariance `MATRIX`
`listall()`	lists the parameters of all component densities of a Parzen density
`save(STRING filename)`	stores parameters (i.e. component weights, means and covariances) of a Parzen density in a file with `filename`
`load (STRING filename)`	reads parameters (i.e. component weights, means and covariances) of a Parzen density from a file with `filename`

Example 1

To explain the concepts of this object, the meaning of the fields and methods, we will work out an example. In this example program (file star.lio) we will construct a 2-dimensional star-shaped mixture density (which is a logo of our program), visualize it and then sample 3000 points from it. In lines `1-42` we define a simple LUA function that computes of Parzen density

```
1   -- first a helper function that produces 2-d density plots
    -- and writes a density to a file

    <
5   function densplot(p,N,xmin,xmax,ymin,ymax)
      local dens
      local xaxis
      local yaxis
      local i
10    local j
      local arg
      dens = MATRIX
      dens.numrows = N
      dens.numcols = N
15    xaxis = VECTOR
      xaxis.length = N
      yaxis = VECTOR
      yaxis.length = N
```

```
      for i=0,N-1 do
20      xaxis[i] = xmin+(i/N)*(xmax-xmin)
      end
      for i=0,N-1 do
        yaxis[i] = ymin+(i/N)*(ymax-ymin)
      end
25    arg = VECTOR
      arg.length = 2
      writeto("star.dat")
      for i=0,N-1 do
        for j=0,N-1 do
30        arg[0] = xaxis[i]
          arg[1] = yaxis[j]
          dens[i][j]= p:density(arg)
          write("\n",arg[0],"  ",arg[1],"  ",dens[i][j])
         end
35    end
      writeto()
      gsurf = dens:xysurfplot(xaxis,yaxis)
      gcont = dens:xycontplot(xaxis,yaxis)
      gcont:addcommand("set cntrparam levels 15")
40    return gsurf,gcont
    end
    >
```

on a regular mesh of values in x- and y-directions and prepares it for plotting with `GNUMACRO` (see Section 5 and 6.1.3) facility. Additionally the computed density is stored in an ASCII text file called ("star.dat") in line 27. The function takes several arguments: density object `p` and the number `N` that controls the amount of plotting positions over the region `[xmin,xmax]`$\times$`[ymin,ymax]`. The returned `Edit` variables `gsurf` and `gcont` can easily be visualized (as will be demonstrated further) via `show()` method to produce surface and contour plots of the density `p` respectively.

Next, in line 48 of the listing below we declare a PARZEN object `p`. In this example `p` is comprised of four uniformly weighted components.

```
    ------------------------------------------------
    -- Set up star-shaped mixture density to sample from
45  ------------------------------------------------

    -- allocate PARZEN object
```

```
    p = PARZEN
    -- define the mean
50  m = VECTOR
    m.length = 2
    m[0] = 0.0
    m[1] = 0.0
    -- and the covariance of first component density
55  c = MATRIX
    c.numrows = 2
    c.numcols = 2
    sx = 2.0
    sy = 0.2
60  rho = 0.0
    c[0][0]  = sx*sx
    c[0][1]  = rho*sx*sy
    c[1][0]  = rho*sy*sx
    c[1][1]  = sy*sy
65  -- add the first component density to a mixture density p
    p:addcomponent(m,c,1.0)
```

The way to build such a mixture density in PARDENS is to add the components one by one to the object `p`. Lines 50-64 define mean vector and the covariance matrix of the first component. This component is then sort of "pushed" onto the mixture density `p` in line 66. Note that the (unit in this case) weight in `addcomponent` method is an unnormalized weight designated as $\tilde{w_n}$. The unnormalized weight is further automatically translated by our program into the normalized weight in (1) by:

$$w_n = \frac{\tilde{w_n}}{\sum_{n=1}^{Nc} \tilde{w_n}} \tag{7}$$

In a similar manner we add the remaining three component densities (lines 70-93):

```
    -- and repeat the same operations for
    -- the next three component densities

70  sx = 0.2
    sy = 2.0
    rho = 0.0
    c[0][0]  = sx*sx
    c[0][1]  = rho*sx*sy
75  c[1][0]  = rho*sy*sx
```

```
    c[1][1]  = sy*sy
    p:addcomponent(m,c,1.0)
    sx = 2.0
    sy = 2.0
    rho = 0.99
    c[0][0]  = sx*sx
    c[0][1]  = rho*sx*sy
    c[1][0]  = rho*sy*sx
    c[1][1]  = sy*sy
    p:addcomponent(m,c,1.0)
    sx = 2.0
    sy = 2.0
    rho = -0.99
    c[0][0]  = sx*sx
    c[0][1]  = rho*sx*sy
    c[1][0]  = rho*sy*sx
    c[1][1]  = sy*sy
    p:addcomponent(m,c,1.0)

    -- calculate moments of the mixture density p
    -- and print them to the screen

    pm,pc = p:moments()
    print("numcomponents = "..p.numcomponents)
    print("mean = ["..pm[0]..","..pm[1].."]")
    print("covariance : ")
    pc:listall()
```

The last few lines of the code above show the use of `moments` method by printing out the mean vector $\mu_{\mathcal{P}}$ and the covariance matrix $\mathcal{C}_{\mathcal{P}}$ of `p`. The ith element of the mean vector is calculated as:

$$\begin{aligned} \mu_{\mathcal{P}}[i] &= \int d\mathbf{x}\; x[i]\; f_{\mathcal{P}}(\mathbf{x}) \\ &= \sum_{n=1}^{N_c} w_n\; \mu_n[i] \end{aligned} \tag{8}$$

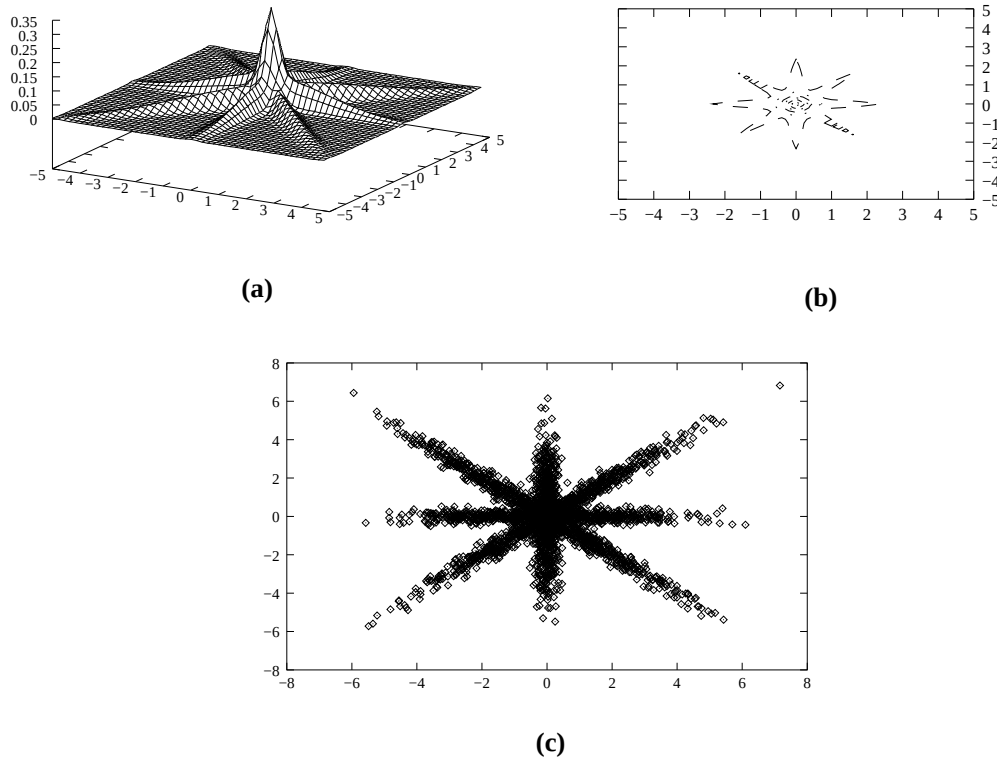


Figure 15: Example of a mixture density produced by the demonstration script `star.lio`. The density is represented as (**a**) surface plot, (**b**) contour plot and (**c**) data points drawn from it.

and the i,jth element of the covariance matrix as:

$$\begin{aligned} \mathcal{C}_{\mathcal{P}}[i,j] &= \int d\mathbf{x}\ (x[i]-\mu_{\mathcal{P}}[i])\ (x[j]-\mu_{\mathcal{P}}[j])\ f_{\mathcal{P}}(\mathbf{x}) \\ &= \sum_{n=1}^{N_c} w_n\ (\mu_n[i]-\mu_{\mathcal{P}}[i])\ (\mu_n[j]-\mu_{\mathcal{P}}[j]) \\ &\quad + \sum_{n=1}^{N_c} w_n\ \mathcal{C}_n[i][j] \end{aligned} \tag{9}$$

where indices i and j are in the range $0,\ldots,$ `dim`$(\mathbf{x})-1$. Finally, the following statements produce surface and contour plot of the density `p` (lines 106-111), sample some points from it via `simulate` method (line 116) and visualize those sampled points by executing `xyscatter()` command on lines 121-122. The resulting plots are depicted in Fig.15.

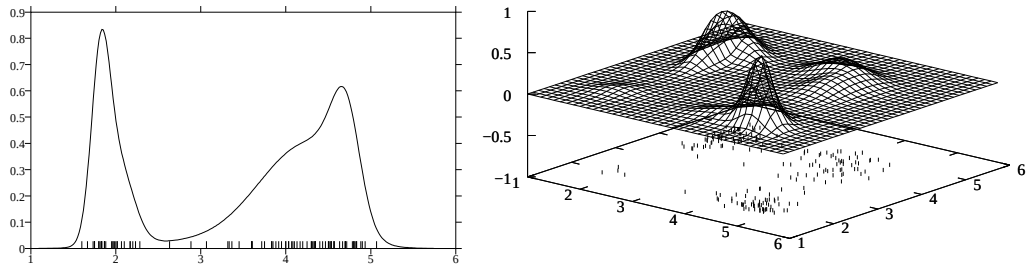


Figure 16: Parzen density fitted to 1-D (left panel) and 2-D (right panel) toy data sets. Data points are marked by small vertical lines.

```
    -- show surface and contour plots of p
105
    gsurf,gcont = densplot(p,50,-5,5,-5,5)
    gsurf:addtitle("The star density")
    gsurf:show()
    pause()
110 gcont:addtitle("The star density")
    gcont:show()

    pause()

115 -- sample 3000 points from p

    simp = p:simulate(3000)

    -- and plot them
120
    simp_plot = simp:xyscatter()
    simp_plot:show()
```

6.2.2 D2P class

While the class described in Section 6.2.1 is useful for modelling Parzen densities, it does not provide the way in which those densities can be fitted to data. For that purpose we developed another class called D2P (Data-To-Parzen). Within this class all the tools necessary to fit Parzen density models, i.e. to find the weights, means and covariances of the components to the user-provided data sets (see Fig.16 for a depiction of this concept), are implemented. To begin a technical description of D2P we present the

interface to the class in question:

- setfields:

`calibdata`	`MATRIX` with calibration data set
`validdata`	`MATRIX` with validation data set
`globality`	`DOUBLE` globality constant γ
`verbose`	verbose level: 0 makes the fitting procedure run silently (even without warning messages); 1 (default value) displays the fitting criterion values during the iteration process

- getfields:

`parzen`	fitted Parzen density object of type `PARZEN`
`numcomponents`	`INT` number of component density functions
`numcalibdata`	`INT` number of points in calibration set
`numvaliddata`	`INT` number of points in validation set
`spacedim`	`INT` dimension of a Parzen density function
`calibcrit`	`DOUBLE` value of the maximized fitting criterion on the calibration set for a given globality constant γ
`validcrit`	the same as above but for validation data set

- methods:

`initialize(INT numcomponents)`	initializes parameters(weights, means and covariances) of a Parzen density with `numcomponents` components; this method must be executed before actual fitting procedure starts.

`optimize(INT maxnumiter, DOUBLE mincritchange)`
starts the iterative fitting procedure; the procedure terminates if either the maximum number of iterations `maxnumiter` or minimum change in fitting criterion function `mincritchange` between two consecutive iterations is reached

Preparing the data

The data to be processed by the D2P class should be read from an input file to a matrix. An input file is a text ASCII file that contains the data set to be analyzed. The data is listed as a data point on each line, with each data point consisting of one or more variables separated by one or more space(s) or tab(s). An example file `stdex.dat` consisting of 5 data points each with 3 variables is shown below:

0.45	8.97	5.00
2.42	-1.88	4.89
-3.34	4.79	6.14
3.38	1.76	2.34
5.65	0.04	1.21

The following piece of code reads the file into PARDENS:

```
data = MATRIX
data:fread("stdex.dat")
```

By convention every row in the `data` matrix represents one data point as in the original input file.

Fitting procedure

The fitting criterion implemented in D2P class is based on maximum log likelihood approach (as in McLachlan and Krishnan, 1997, Section 2.7). The new twist proposed by Torfs and Wójcik (2003) is incorporation of an extra regularization term - the average Kullback-Leibler distance (Kullback and Leibler, 1951) between the components of the Parzen density and the global Gaussian fitted to data. Imposing such a penalty avoids all numerical problems, as e.g. those with singular covariance matrices, that have frequently been encountered when fitting mixture densities with classical log likelihood

maximization. Formally, the fitting procedure can be represented as the following optimization problem:

Given:

- a globality constant γ
- a data sample $\mathbf{x}_1, \ldots, \mathbf{x}_{N_d}$, with the following moments:

$$\mu_{\mathcal{S}}[j] = \frac{1}{N_d} \sum_{k=1}^{N_d} x_k[j] \tag{10}$$

$$\mathcal{C}_{\mathcal{S}}[j_1][j_2] = \frac{1}{N_d} \sum_{k=1}^{N_d} (x_k[j_1] - \mu_{\mathcal{S}}[j_1]) \, (x_k[j_2] - \mu_{\mathcal{S}}[j_2]) \tag{11}$$

- a number of components $N_c \leq N_d$

Find:

a Parzen density $f_{\mathcal{P}}(\mathbf{x})$ as defined in (1) with components' weights $w_1, \ldots, w_{N_c}$, means and covariances $\mu_1, \ldots, \mu_{N_c}$, $\mathcal{C}_1, \ldots, \mathcal{C}_{N_c}$ such that

$$\Lambda = \mathcal{L} - h(\gamma) \left[\frac{1}{N_c} \sum_{n=1}^{N_c} \int d\mathbf{x} \; f_{\mathcal{N}(\mu_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})}(\mathbf{x}) \log \left(\frac{f_{\mathcal{N}(\mu_n, \mathcal{C}_n)}(\mathbf{x})}{f_{\mathcal{N}(\mu_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})}(\mathbf{x})} \right) \right] \tag{12}$$

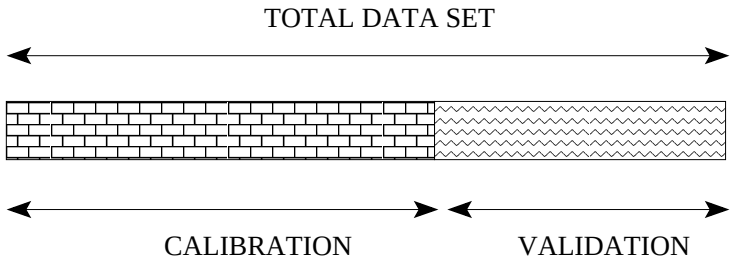
is maximal **under the condition** that $\sum_{n=1}^{N_c} w_n = 1$. The term $\mathcal{L}$ stands for the log likelihood function, $h(\gamma) \stackrel{\circ}{=} \sqrt{1/(1-\gamma^2)} - 1$ and $f_{\mathcal{N}(\mu_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})}(\cdot)$ stands for the global Gaussian fitted to data.

It is clear from the description above that the user has to choose two parameters before starting the maximization: the number of components N_c and the globality constant γ. To see that this choice is non-trivial let us consider two "limiting" cases:

- If $\gamma = 0$ then $h(\gamma) = 0$ and criterion (12) will become pure log likelihood criterion which gives high preference to very local density functions. Therefore, setting $N_c = N_d$ would result in Dirac density centered on each data point. To avoid such degeneration $N_c \ll N_d$. Several heuristic ways of choosing N_c are described in McLachlan and Peel (2000). None of them, however, guarantees numerical stability of the fitting procedure unless extra assumptions are made with regard to the form of covariance matrices of the components.

- If $\gamma \longrightarrow 1$ then $h(\gamma) \longrightarrow \infty$ and one essentially fits a global Gaussian to data. In this case the choice of N_c is irrelevant as all components are equal.

Intuitively it follows that the higher the value of γ the smaller the N_c should be since a density function that gets closer to the Gaussian will anyway not be able to capture geometric details potentially present in data space. However, due to the Kullback-Leibler penalty term used in (12) it is numerically safe, for all $\gamma \in (0; 1)$, to simply set $N_c = N_d$ although it increases computing time. With this choice there is only one parameter left to be optimized: γ. In PARDENS we adapt the approach by Hastie et al. (2001) p. 196 in which the total data set is divided randomly (unless there are good reasons to do the division deterministicly) into two parts: a *calibration set* and a *validation set*. The calibration set is used to fit the density models for a given range of γ while the validation set is used to estimate the criterion (12) for model selection. It is difficult to give a general rule on how to choose the number of observations in each of the two parts, as it depends on the sample size and signal-to-noise ratio in the data. A typical split might be 60 % for calibration and 40% for validation:



The model selection runs according to the following algorithm:

set N_c equal to the number of points in the calibration set
set $0 < \gamma \ll 1$
for a number of γ values $\in (0; 1)$ **do**
    for a given γ MAXIMIZE Λ in (12) on the calibration set
    STORE the parameters of Parzen density
    ESTIMATE Λ on the validation set using the above parameters
    STORE this Λ
end
Optimal γ or optimal density model is one that corresponds to the largest of the STORED Λ values.

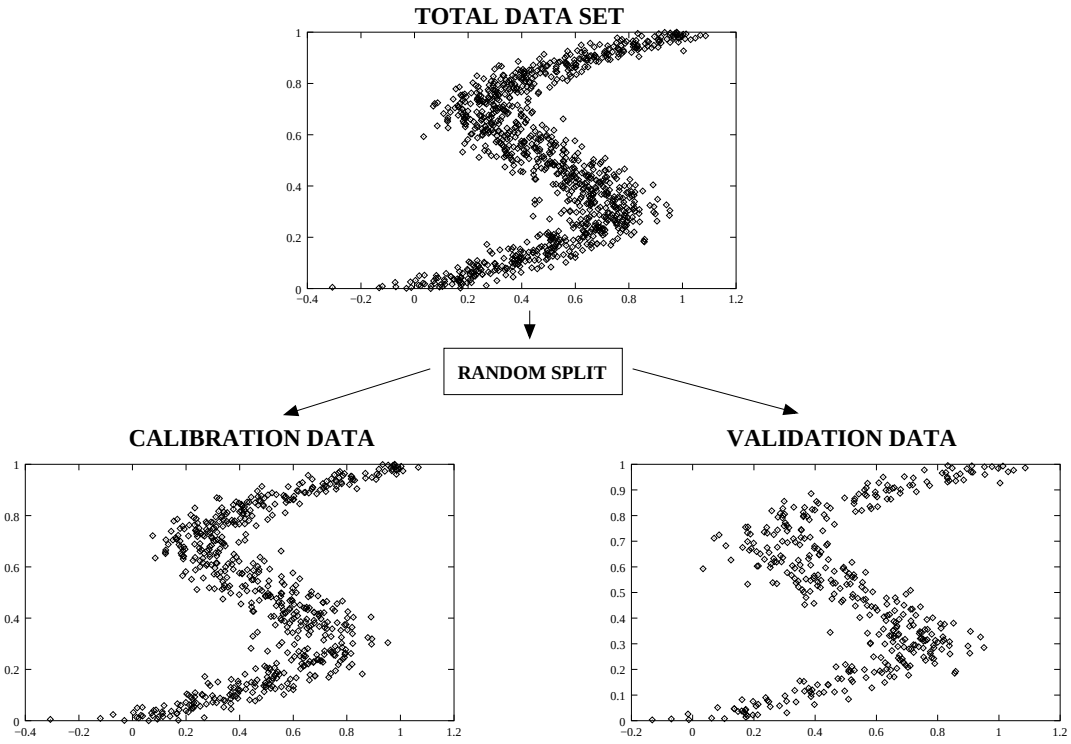


Figure 17: Data used in demonstration script `onetomany.lio`. The total data set (1000 points) was randomly divided into calibration part (600 points) and validation part (400 points).

Example 2

For the example of this Section, we will use the 2-D data set shown in Fig. 17. The actual numbers of this example were obtained from non-linear one-to-many mapping (for more details see Example 3 in Section 6.2.3) inspired by Bishop (1995) p.204 and were stored in file `mulvalmap.dat`. To illustrate Parzen density fitting to this data set we developed a demonstration script `onetomany.lio`. For clarity, we start the listing after omitting the definition of density plotting function (see Example 1 in Section 6.2.1):

```
   -- a helper function to randomize rows of a matrix

   <
40 function randomizerows(m)
     local temptosswap
     for i=0, m.numrows-1 do
```

```
        j = floor((i+1)*random())
        for k=0, m.numcols-1 do
          temptoswap = m[i][k]
          m[i][k] = m[j][k]
          m[j][k] = temptoswap
        end
      end
end
>
```

A helper function above is designed to shuffle rows of a matrix `m` in a random manner. This matrix can represent a particular data set. The basic idea here is to randomly draw new data set without replacement from the original data set. The new data set has exactly the same size as the original one. Such a procedure is generally known as the *bootstrap* (see Efron and Tibshirani, 1993).

```
--  main
--  read in the data

onetomany = MATRIX
onetomany:fread("mulvalmap.dat")
print("numdata read = ",onetomany.numrows)

--randomize rows of onetomany MATRIX

randomseed(7866023)
randomizerows(onetomany)

--  show the scatter plot

onetomany:xyscatter():show()
onetomany:xyscatter():fig("totaldata")
pause()

--  split the data into calibration and validation part

caldata = MATRIX
caldata.numrows = 600
caldata.numcols = 2
valdata = MATRIX
```

```
    valdata.numrows = 400
    valdata.numcols = 2

    <
80  for i=0,caldata.numrows-1 do
      caldata[i][0] = onetomany[i][0]
      caldata[i][1] = onetomany[i][1]
    end
    >
85  <
    for i=0,valdata.numrows-1 do
      valdata[i][0] = onetomany[i+caldata.numrows][0]
      valdata[i][1] = onetomany[i+caldata.numrows][1]
    end
90  >

    -- show the scatter plots of both parts

    caldata:xyscatter():show()
95  caldata:xyscatter():fig("caldata")
    pause()

    valdata:xyscatter():show()
    valdata:xyscatter():fig("valdata")
100 pause()
```

In lines 55-57 we read the data set of this example into `onetomany` matrix and print the total number of data points (1000 in this case). After fixing the seed of pseudo-random number generator in line 61 (to get reproducible results) we randomize rows of the `onetomany` matrix in line 62. This operation is needed to split the total data set into the calibration and the validation part by lines 70-90. The total data set, the calibration part and the validation part are visualized in form of scatter plots in lines 66,94 and 98 respectively and saved as `Xfig` files in lines 67,95 and 99 respectively. Those plots are shown in Fig.17.

Once the data have been prepared we can start the actual fitting procedure:

```
-- fitting a Parzen density by increasing the gamma
```

```
    fit = D2P
105 fit.calibdata = caldata
    fit.validdata = valdata

    -- initilization

110 fit.globality = 0.01
    fit:initialize(600)
    fit:optimize(50,0.001)

    -- define a matrix to store criterion values
115 -- on the calibration and validation set

    crits = MATRIX
    crits.numrows = 101
    crits.numcols = 2
120 crits[0][0] = fit.globality
    crits[0][1] = fit.validcrit

    Numglobs = crits.numrows
    startglob = 0.02
125 endglob   = 0.99

    -- main loop

    <
130 for i=0,Numglobs-1 do
      fit.globality = startglob + (endglob-startglob)*i/Numglobs
      fit:optimize(50,0.001)
      crits[i][0] = fit.globality
      crits[i][1] = fit.validcrit
135   if (mod(i,10) == 0) then
         print(fit.globality)
         filename="otm_1"..i..".fig"
         gsurf,gcont = densplot(fit.parzen,50,-0.5,1.5,-0.5,1.5)
         gsurf:addcommand("set view 20,10,1")
140      gsurf:fig(filename)
      end
    end
    >
    pause()
```

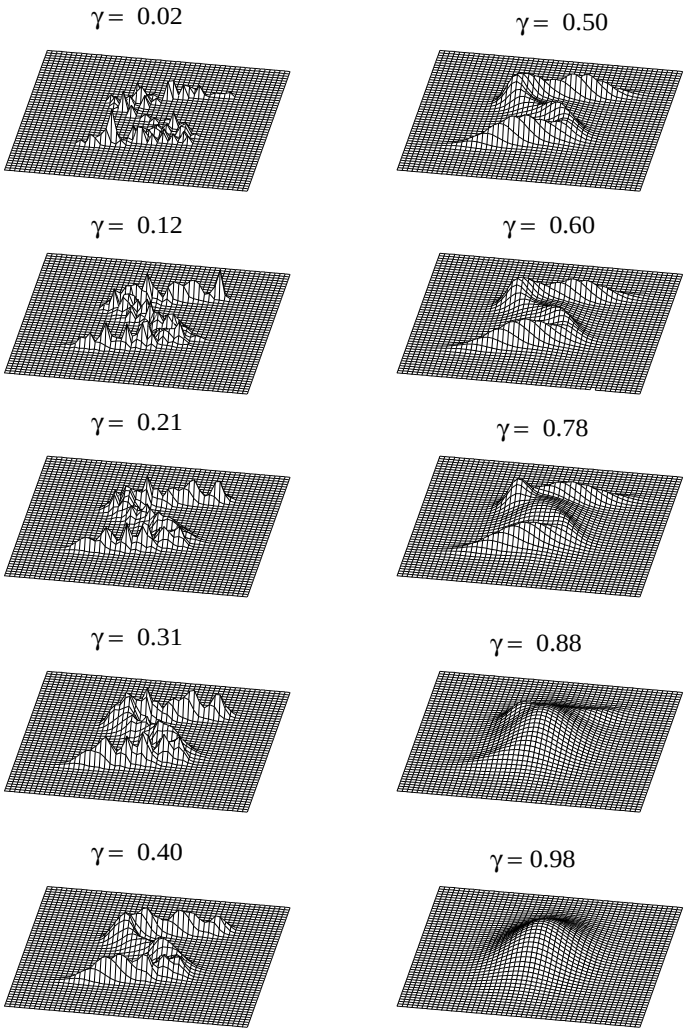


Figure 18: A sequence of Parzen densities fitted to the calibration data from the bottom left panel of Fig. 17.

In lines 104-106 an object `fit` of D2P class type is defined and two fields of that object - the calibration and the validation data - are filled in. Next (lines 110-111) we define the starting value of globality constant as $\gamma = 0.01$ and fix the number of components $N_c = 600$ which is exactly the number of points in the calibration set. Line 112 initializes parameters of Parzen density (for technical details of the initialization see Torfs and Wojcik, 2003) and maximizes Λ in (12) for those initial parameters. The result of that optimization is very local Parzen density fitted to the calibration data. The globality constant γ and the corresponding value of Λ for the validation set is stored as 0th row of the `crit` matrix in lines 120-121. This matrix has 101 rows which means that we are going to re-estimate Parzen density for

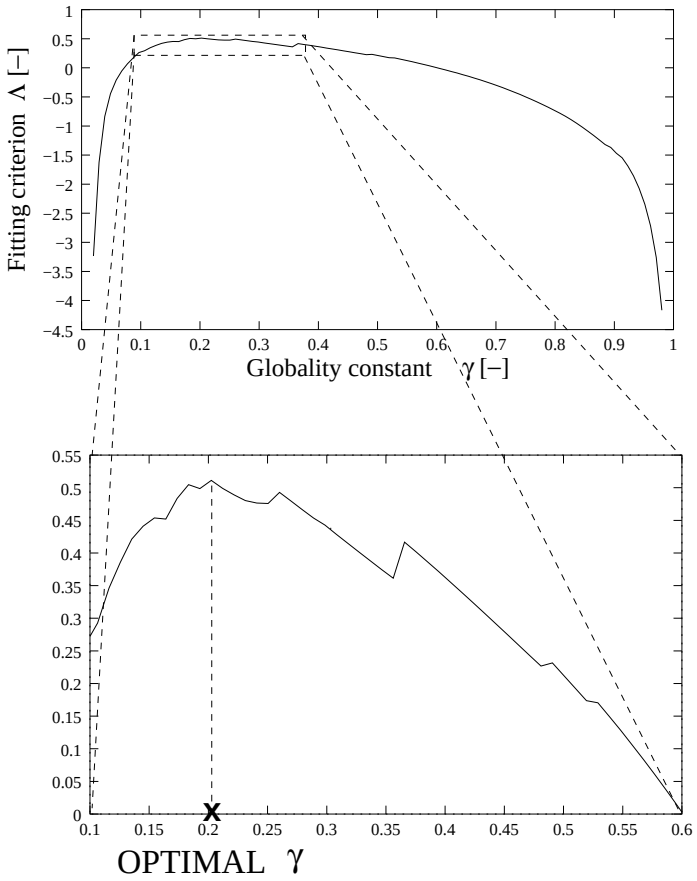


Figure 19: Fitting criterion plot for the validation set from Fig.17. The selected optimal value of $\gamma \backsimeq 0.21$ is marked as a cross.

101 different values of γ (line 123) ranging from `startglob` to `endglob` (lines 124-125). From those values we hope to infer the optimal one.

In the main loop of the fitting procedure (lines 129-143) we proceed according to the algorithm presented on page 28. Lines 135-140 execute some density plotting commands for every 10th value of γ. The resulting plots are depicted in Fig.18. Note the smooth evolution of the density function from very local one (upper left panel) to one that is almost global Gaussian (lower right panel). The next lines of the code:

```
145
      --  show the criterion plot for the validation set
```

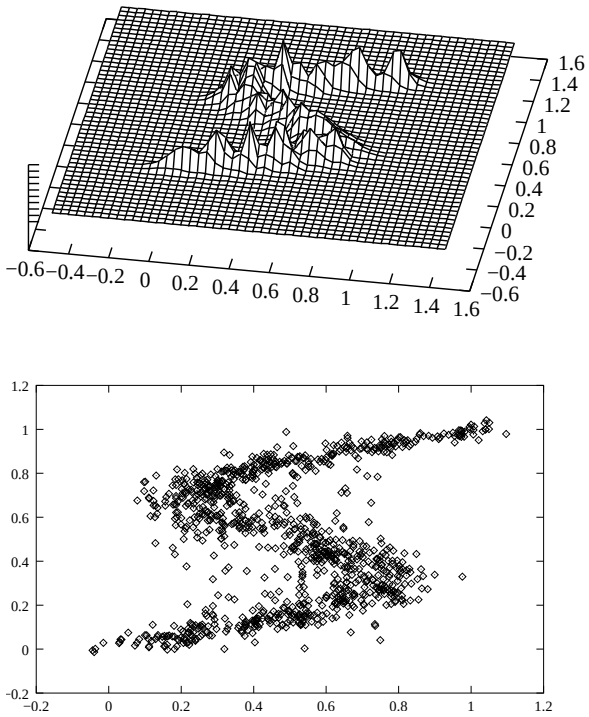


Figure 20: A Parzen density obtained with an optimal value of $\gamma = 0.21$. Upper panel: surface plot; lower panel: a sample o 1000 points drawn from the density.

```
    critsplot = crits:xyline()
    critsplot:addcommand("set xlabel \"globality\"")
150 critsplot:addtitle("The criterium values")
    critsplot:show()
    critsplot:fig("criteriaotm")
    pause()

155 --  enlarge the criterion plot in the region
    --  where the global maximum is identified to occur

    critsplot = crits:somexyline(0,1)
    critsplot:addcommand("set grid")
160 critsplot:addcommand("set xrange [0.1:0.6]")
    critsplot:show()
    critsplot:fig("critplotenl")
    pause()
```

produce the criterion plots for selecting the optimal value of γ. Figure 18 shows those plots. It is easy to see that the optimal γ which corresponds to the largest value of the criterion on the validation set is about 0.21. In what follows we re-estimate the density model for that optimal globality constant:

```
    --  The density model for the optimal
165 --  globality constant 0.21

    fit_optimal = D2P
    fit_optimal.calibdata = caldata
    fit_optimal.globality = 0.01
170 fit_optimal:initialize(caldata.numrows)
    fit_optimal.globality = 0.21
    fit_optimal:optimize(500,0.001)

    -- show the surface plot
175
    gsurf,gcont = densplot(fit_optimal.parzen,50,-0.5,1.5,-0.5,1.5)
    gsurf:show()

    -- store the model to a file
180
    fit_optimal.parzen:save("onetomany.parzen")

    --  just for fun simulate 1000 points from it and
    --  show the scatter plot
185
    sim_opt = fit_optimal.parzen:simulate(1000)
    sim_opt:xyscatter():show()
    sim_opt:xyscatter():fig("simulated")
```

plot it (lines 176-177), write it to the file `onetomany.parzen` (line 181) and simulate 1000 points from it (line 186). The surface plot of the optimal density and the simulated points are shown in Fig.20.

6.2.3 XYPARZEN class

The main idea behind XYPARZEN class is to apply Parzen densities that have already been fitted to data, to regression function estimation, marginal density estimation and conditional simulation. That application requires the user to make a clear distinction between two types of variables e.g., "`x`" and

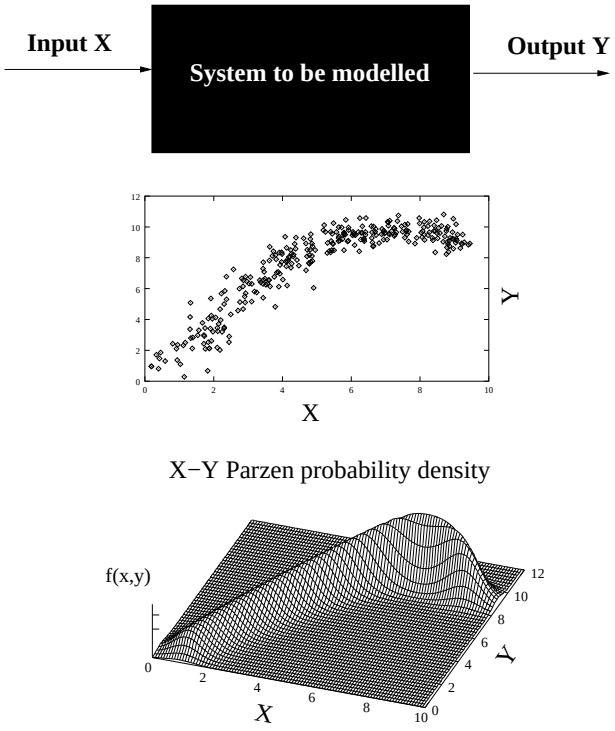


Figure 21: Parzen densities re-cast into systems' modeling framework. The data points are the input-output pairs which characterize the system to be modelled. To those data a joint (X-Y) Parzen density is fitted. This fitting is the first step in black-box modeling of the system.

"y" variables in data underlying a Parzen density. For instance, in systems' modeling "x" could represent a set of inputs and "y" could represent the set of outputs. An example of this for a model with one input and one output is shown in Fig.21.

The above framework requires some reformulation and complementation of the theory introduced in the beginning of Section 6.2. Let $\mathbf{X}$ and $\mathbf{Y}$ be two multivariate stochastic variables and $\mathbf{x}$ and $\mathbf{y}$ their realizations, respectively. Those variables are said to have a joint Parzen density if their density function can be written as:

$$f_{\mathcal{P}_{\mathbf{XY}}}(\mathbf{x},\mathbf{y}) = \sum_{n=1}^{N} w_n \; f_{\mathbf{XY}}(\mathbf{x},\mathbf{y}|n) \tag{13}$$

or, more compactly by defining $\mathbf{Z} = [\mathbf{X}; \mathbf{Y}]^T$ as:

$$f_{\mathcal{P}_\mathbf{Z}}(\mathbf{z}) = \sum_{n=1}^{N} w_n \; f_\mathbf{Z}(\mathbf{z}|n) \tag{14}$$

where nth component $f_\mathbf{Z}(\mathbf{z}|n)$ is a Gaussian function defined analogously to (5) with parameters :

$$\mu_\mathbf{n} = \begin{bmatrix} \mu_\mathbf{X}^{(\mathbf{n})} \\ \mu_\mathbf{Y}^{(\mathbf{n})} \end{bmatrix} \tag{15}$$

$$\mathcal{C}_\mathbf{n} = \begin{bmatrix} \mathcal{C}_\mathbf{X}^{(\mathbf{n})} & \mathcal{C}_\mathbf{XY}^{(\mathbf{n})} \\ \mathcal{C}_\mathbf{YX}^{(\mathbf{n})} & \mathcal{C}_\mathbf{Y}^{(\mathbf{n})} \end{bmatrix} \tag{16}$$

where μ_n is the joint mean vector and $\mathcal{C}_n$ is the joint covariance matrix (for details of the covariance matrix partitioning in (16) see e.g Johnson and Wichern, 1988, p.58). Note, that because $\mathbf{X}$ and $\mathbf{Y}$ are jointly Parzen distributed then each marginal density is also Parzen. For example:

$$f_{\mathcal{P}_\mathbf{X}}(\mathbf{x}) = \sum_{n=1}^{N} w_n \; f_\mathbf{X}(\mathbf{x}|n) \tag{17}$$

where $f_{\mathcal{P}_\mathbf{X}}(\mathbf{x}|n)$ has parameters $\mu_\mathbf{X}^{(\mathbf{n})}$ and $\mathcal{C}_\mathbf{X}^{(\mathbf{n})}$. Moreover, the conditional density of $\mathbf{Y}$ given $\mathbf{X}$ is expressed as:

$$\begin{aligned} f_{\mathcal{P}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}}(\mathbf{y}) &= \frac{f_{\mathcal{P}_\mathbf{XY}}(\mathbf{x}, \mathbf{y})}{f_{\mathcal{P}_\mathbf{X}}(\mathbf{x})} \\ &= \frac{\sum_n w_n \; f_\mathbf{XY}(\mathbf{x}, \mathbf{y}|n)}{\sum_n w_n \; f_\mathbf{X}(\mathbf{x}|n)} \\ &= \frac{\sum_n w_n \; f_\mathbf{X}(\mathbf{x}|n) \; f_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}(\mathbf{y}|n)}{\sum_n w_n \; f_\mathbf{X}(\mathbf{x}|n)} \end{aligned} \tag{18}$$

As $f_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}(\mathbf{y}|n)$ is a Gaussian component and the conditional of the Gaussian is also Gaussian (see e.g. Johnson and Wichern, 1988, p.127), the conditional density in (13) is again a Parzen density, with "new" weights $w_n \; f_\mathbf{X}(\mathbf{x}|n)$. The conditional mean also called the *generalized regression curve* is given by:

$$\begin{aligned} E[\mathbf{Y}|\mathbf{X} = \mathbf{x}] &= \widehat{\mathbf{y}}(\mathbf{x}) = \int \mathbf{y} f_{\mathcal{P}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}}(\mathbf{y}) d\mathbf{y} \\ &= \frac{\sum_n w_n \; f_\mathbf{X}(\mathbf{x}|n) \; \mu_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}^{(n)}}{\sum_n w_n \; f_\mathbf{X}(\mathbf{x}|n)} \end{aligned} \tag{19}$$

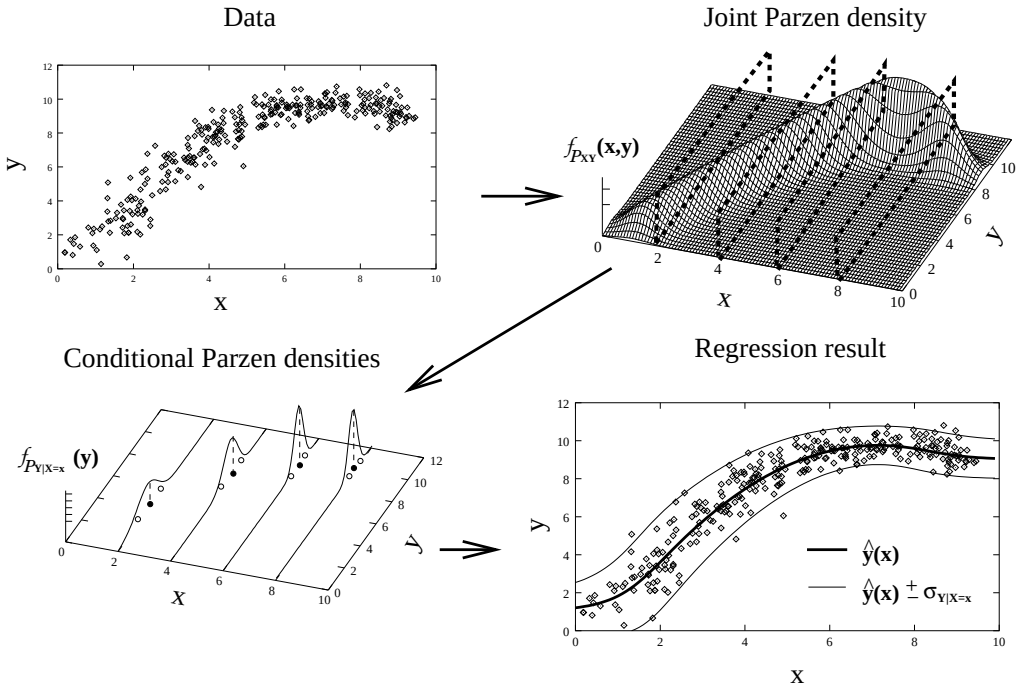


Figure 22: A principle of generalized regression based on Parzen densities.

where:

$$\mu^{(n)}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} = \mu^{(\mathbf{n})}_{\mathbf{Y}} + \mathcal{C}^{(\mathbf{n})}_{\mathbf{XY}} \mathcal{C}^{(\mathbf{n})^{-1}}_{\mathbf{X}} (\mathbf{x} - \mu^{(\mathbf{n})}_{\mathbf{X}}) \tag{20}$$

is the local conditional expectation. To assess the error in the generalized regression curve we need to estimate the conditional covariance:

$$\begin{aligned} \mathcal{C}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} &= E[(\mathbf{Y} - \widehat{\mathbf{y}}(\mathbf{x}))(\mathbf{Y} - \widehat{\mathbf{y}}(\mathbf{x}))^T | \mathbf{X} = \mathbf{x}] \\ &= \int (\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{x}))(\mathbf{y} - \widehat{\mathbf{y}}(\mathbf{x}))^T f_{\mathcal{P}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}}(\mathbf{y}) d\mathbf{y} \qquad (21) \\ &= \frac{\sum_n w_n \; f_{\mathbf{X}}(\mathbf{x}|n); \{\mathcal{C}^{(n)}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} + (\mu^{(n)}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} - \widehat{\mathbf{y}}(\mathbf{x}))(\mu^{(n)}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} - \widehat{\mathbf{y}}(\mathbf{x}))^T\}}{\sum_n w_n \; f_{\mathbf{X}}(\mathbf{x}|n)} \end{aligned}$$

where:

$$\mathcal{C}^{(n)}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}} = \mathcal{C}^{(\mathbf{n})}_{\mathbf{Y}} - \mathcal{C}^{(\mathbf{n})}_{\mathbf{XY}} \mathcal{C}^{(\mathbf{n})^{-1}}_{\mathbf{X}} \mathcal{C}^{(\mathbf{n})}_{\mathbf{YX}} \tag{22}$$

is the local conditional covariance. As symmetric and unit-preserving measure of this error, we simply calculate the standard deviation:

$$\sigma_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}[i] = \pm\sqrt{\mathcal{C}_{\mathbf{Y}|\mathbf{X}=\mathbf{x}}[i,i]} \tag{23}$$

Figure 22 shows construction of the generalized regression curve based on Parzen densities. The first step is to estimate a joint (X-Y) Parzen density $f_{\mathcal{P}_{\mathbf{XY}}}(\mathbf{x},\mathbf{y})$ for data shown in the upper left panel of the figure. This density is plotted in the upper right panel. Afterwards, we need to calculate the conditional densities in (18) for many selected values of $\mathbf{x}$ in order to obtain estimates of conditional expectation in (19) and conditional standard deviation in (23). This is done by (scaled) projecting $f_{\mathcal{P}_{\mathbf{XY}}}(\mathbf{x},\mathbf{y})$ onto "$f_{\mathcal{P}_{\mathbf{XY}}}(\mathbf{x},\mathbf{y}) - \mathbf{y}$" planes associated with each selected value of $\mathbf{x}$. Four examples of such planes are represented by dashed lines in the upper right panel of Fig.22. The conditional densities in those planes together with the conditional means (filled dots) and standard deviations (empty dots) are depicted in the lower-left panel. The entire regression curve with standard deviation error bands is shown in the lower right panel.

The interface to the XYPARZEN class reads:

- getfields :

`margx`	marginal Parzen density object of type `PARZEN` for `x` variable(s)
`condy`	conditional (`y` **given** `x`) Parzen density object of type `PARZEN`
`dimx`	`INT` dimension of `x` space
`dimy`	`INT` dimension of `y` space

- methods :

`construct(PARZEN fromparzen, INT dimx, INT dimy)`	constructs `XYPARZEN` density object by specifying how many `x` variables and `y` variables are present in `fromparzen` density. Note that the order of variables is important: `x`'s always go first.
`setxcondition(VECTOR x)`	sets `x` condition. This is needed for conditional Parzen density estimation.

Example 3

To illustrate the use of XYPARZEN class, the demonstration script `regress.lio` uses calibration data from section 6.2.2 (see Fig.17) with one input ($\mathbf{x}$ variable) and one output ($\mathbf{y}$ variable). This allows us to plot regression result together with the full conditional densities and show how they fit the data. It is important to note that the generation function for this example is defined by:

$$\mathbf{x} = \mathbf{y} + 0.45 sin(2\pi \mathbf{y}) + 0.1\varepsilon \tag{24}$$

where ε is a random variable with a $\mathcal{N}(0,1)$ distribution. Therefore $\mathbf{x}$ is a proper function of $\mathbf{y}$: this data models applications where the "forward" problem (mapping $\mathbf{y}$ to $\mathbf{x}$) is a single-valued (one-to-one) mapping while the "inverse" problem (mapping $\mathbf{x}$ to $\mathbf{y}$) is multivalued (one-to-many) mapping. For $\mathbf{x} \in [0.1; 0.9]$ the mapping has three branches as shown in Fig.17.

For clarity the listing below starts after skipping definitions of helper functions (some aspects of those functions will be discussed further):

```
--read the calibration data

onetomany = MATRIX
onetomany:fread("mulvalmap.dat")
print("numdata read = ",onetomany.numrows)

randomseed(7866023)
randomizerows(onetomany)


caldata = MATRIX
caldata.numrows = 600
caldata.numcols = 2

<
for i=0,caldata.numrows-1 do
  caldata[i][0] = onetomany[i][0]
  caldata[i][1] = onetomany[i][1]
end
>

-- load "the optimal" Parzen density for those data
```

```
240   p=PARZEN
      p:load("onetomany.parzen")
      gsurf,gcont = densplot(p,50,-0.5,1.5,-0.5,1.5)
      gsurf:show()
      pause()
```

Lines 219-236 reconstruct the calibration data from file `mulvalmap.dat`. Note that in line 223 we use exactly the same seed of random number generator as in the script `onetomany.lio` from the previous Section. Next, in line 241 we load the Parzen density that was found to be optimal for the analyzed data set (see Fig.20). The surface plot of that density is produced anew by lines 242-243.

```
245
      -- construct XYPARZEN out of p object

      pxy = XYPARZEN
      pxy:construct(p,1,1)
250
      -- show the regression plot

      regs = standardregdataplot(caldata,-0.5,1.5,300)
      regs:show()
255   pause()
```

The `pxy` object of type XYPARZEN is declared and constructed by executing lines 248-249. Function `standardregdataplot(data,xmin,xmax,numxs)` in line 253 calculates, given XYPARZEN density object `p` for a number of plotting positions `numxs` on `x` axis in the range `[xmin;xmax]`, a regression curve together with standard deviation error bounds, and underlying `data`. The essential lines of that calculation, without going into details of the function body, are listed below:

```
165
        preds = MATRIX
        preds.numrows = numxs
        preds.numcols = 4
        x = VECTOR
170     x.length = 1
        y = VECTOR
        y.length = 1
        for i=0, numxs-1 do
```

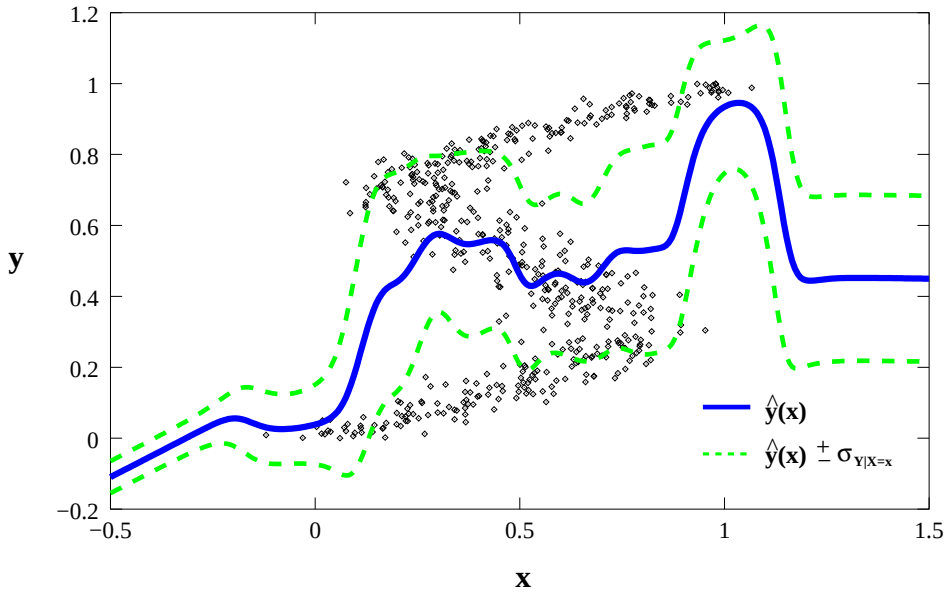


Figure 23: A generalized regression curve (blue solid line) with standard deviation error bands (green dashed lines) for the calibration data (black dots) of Fig.17

```
        x[0] =  xmin+(i/(numxs-1))*(xmax-xmin)
175     pxy:setxcondition(x)
        y,c = pxy.condy:moments()
        stdev = sqrt(c[0][0])
        preds[i][0] = x[0]
        preds[i][1] = y[0]
180     preds[i][2] = y[0]-stdev
        preds[i][3] = y[0]+stdev
    end
```

Note that line 176 is an implementation of (19) and (21) and line 177 of (23). The resulting plot is presented in Fig.23. It is clear from the figure that the conditional mean gives a poor fit to the function underlying the data. This is due to multi-branched nature of the function. However, in Parzen regression there is more information available than just conditional mean. We can, e.g. easily plot graphs of the full conditional densities:

```
-- show the regression plot together with full conditional
-- densities
```

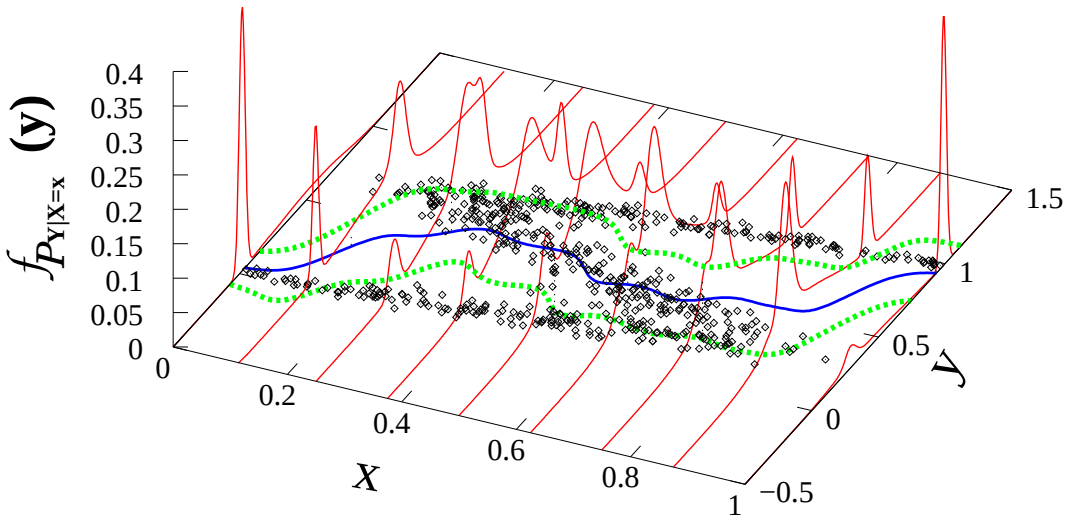


Figure 24: Conditional densities (red solid line) together with generalized regression curve (blue solid line) and standard deviation error bands (green dashed lines) for the calibration data (black dots) of Fig.17. Note that due to multi-branched nature of the data the conditional densities are sometimes two or three modal.

```
260   reg3d = reg3dplot(0,1,300,-0.5,1.5,300, 8)
      reg3d:show()
      pause()
```

with the helper function:

```
reg3dplot(xmin,xmax,numxs,ymin,ymax,numys,numdens)
```

The first three arguments are defined the same way as for the function `standardregdataplot`. The arguments `ymin` and `ymax` control the range of `y` axis and `numdens` determines the number of conditional densities to be plotted within `[xmin;xmax]` interval. The number of plotting position for each of those densities is controlled by `numys`. Figure 24 shows the graph of several conditional densities. Clearly the multimodal structure of the underlying data is well represented. The conditional mean and standard deviation should be than interpreted through the modes of the conditional densities.

Another way of expressing the above conditional information is to represent the conditional densities by points drawn from them. This can be achieved by so-called *conditional simulation*:

```
-- perform conditional simulation
```

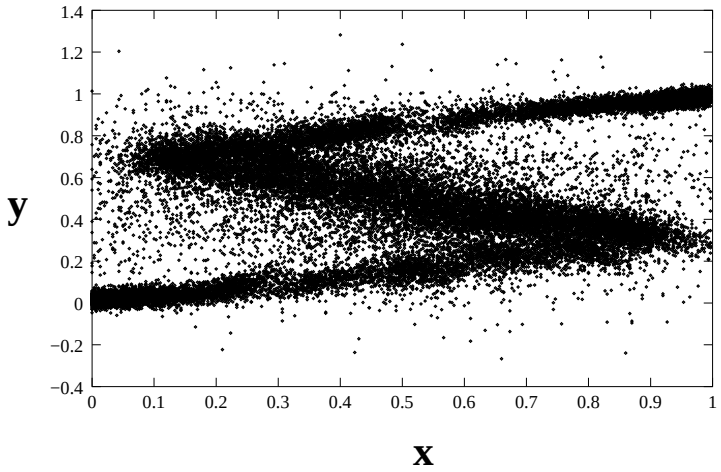


Figure 25: Conditional simulation.

```
265
      xmin = 0
      xmax = 1
      ymin = -0.5
      ymax = 1.5
270   numxs = 300
      numsimys = 100

      xc = VECTOR
      xc.length = 1
275   ys = MATRIX
      ys.numrows = 1
      ys.numcols = 1

      simulateddata = MATRIX
280   simulateddata.numrows = numxs*numsimys
      simulateddata.numcols = 2


      z=0
285   <
      for i=0,numxs-1 do
          xc[0] =  xmin+(i/numxs)*(xmax-xmin)
          pxy:setxcondition(xc)
```

```
        for j=0,numsimys-1 do
290       ys = pxy.condy:simulate(1)
          simulateddata[z][0]=xc[0]
          simulateddata[z][1]=ys[0][0]
          z=z+1
        end
295   end
    >
    simulateddata:xyscatter():show()
```

The code above calculates 300 conditional densities (for 300 $\mathbf{x}$ conditions) in line 288 and simulates 100 points from each density (lines 289-294). The simulated points are shown in Fig.25. It is easy to see that again multi-branched nature of the data is correctly reproduced. Morover, conditional simulation compares favorably with unconditional simulation depicted in lower panel of Fig.20.

References

Bishop, M. C. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press, New York.

Efron, B. and Tibshirani, R. (1993). *An Introduction to the Bootstrap.* Chapman and Hall, London.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Verlag, London.

Johnson, R. A. and Wichern, D. (1988). *Applied Multivariate Statistical Analysis.* Prentice Hall International Inc., New Jersey.

Kullback, S. and Leibler, R. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86.

McLachlan, G. and Krishnan, T. (1997). *The EM Algorithm and Extensions.* Wiley Iterscience, New York.

McLachlan, G. and Peel, D. A. (2000). *Finite Mixture Models.* Wiley Iterscience, New York.

Parzen, E. (1962). On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33:1065–1076.

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis.* Chapman and Hall, New York.

Torfs, P. and Wójcik, R. (2003). Fitting multidimensional Parzen densities with the use of a Kullback-Leibler penalty. *Submitted to Journal of the American Statistical Association.*

Acknowledgments

This work was sponsored by the Netherlands Institute of Applied Geoscience TNO within the framework of the SAMCARDS project.