Package 'aplpack'

July 29, 2019

Title Another Plot Package: 'Bagplots', 'Iconplots', 'Summaryplots', Slider Functions and Others

Version 1.3.3

Date 2019-07-26

Author Hans Peter Wolf [aut, cre]

Maintainer Hans Peter Wolf <pwolf@wiwi.uni-bielefeld.de>

Depends R (>= 3.0.0)

Suggests tkrplot, jpeg, png, splines, utils, tcltk

Description Some functions for drawing some special plots:
The function 'bagplot' plots a bagplot,
'faces' plots chernoff faces,
'iconplot' plots a representation of a frequency table or a data matrix,
'plothulls' plots hulls of a bivariate data set,
'plotsummary' plots a graphical summary of a data set,
'puticon' adds icons to a plot,
'skyline.hist' combines several histograms of a one dimensional data set in one plot,
'slider' functions supports some interactive graphics,
'spin3R' helps an inspection of a 3-dim point cloud,
'stem.leaf' plots a stem and leaf plot,
'stem.leaf.backback' plots back-to-back versions of stem and leaf plot.

License GPL (>= 2)

URL http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/wolf_aplpack

NeedsCompilation no

Repository CRAN

Date/Publication 2019-07-29 07:50:12 UTC

R topics documented:

`bagplot` *bagplot, a bivariate boxplot*

Description

`compute.bagplot()` computes an object describing a bagplot of a bivariate data set. `plot.bagplot()` plots a bagplot object. `bagplot()` computes and plots a bagplot.

Usage

```
bagplot(x, y, factor = 3, na.rm = FALSE, approx.limit = 300,
       show.outlier = TRUE, show.whiskers = TRUE,
       show.looppoints = TRUE, show.bagpoints = TRUE,
       show.loophull = TRUE, show.baghull = TRUE,
       create.plot = TRUE, add = FALSE, pch = 16, cex = 0.4,
       dkmethod = 2, precision = 1, verbose = FALSE,
       debug.plots = "no",   col.loophull="#aaccff",
       col.looppoints="#3355ff", col.baghull="#7799ff",
       col.bagpoints="#000088", transparency=FALSE,
       show.center = TRUE, ...
)
compute.bagplot(x, y, factor = 3, na.rm = FALSE, approx.limit = 300,
       dkmethod=2,precision=1,verbose=FALSE,debug.plots="no")
## S3 method for class 'bagplot'
plot(x,
       show.outlier = TRUE, show.whiskers = TRUE,
       show.looppoints = TRUE, show.bagpoints = TRUE,
       show.loophull = TRUE, show.baghull = TRUE,
```

```
        add = FALSE, pch = 16, cex = 0.4, verbose = FALSE,
        col.loophull="#aaccff", col.looppoints="#3355ff",
        col.baghull="#7799ff", col.bagpoints="#000088",
        transparency=FALSE,
        show.center = TRUE, ...)
```

Arguments

x	x values of a data set; in `bagplot`: an object of class `bagplot` computed by `compute.bagplot`
y	y values of the data set
factor	factor defining the loop
na.rm	if TRUE 'NA' values are removed otherwise exchanged by median
approx.limit	if the number of data points exceeds `approx.limit` a sample is used to compute some of the quantities; default: 300
show.outlier	if TRUE outlier are shown
show.whiskers	if TRUE whiskers are shown
show.looppoints	if TRUE loop points are plottet
show.bagpoints	if TRUE bag points are plottet
show.loophull	if TRUE the loop is plotted
show.baghull	if TRUE the bag is plotted
create.plot	if FALSE no plot is created
add	if TRUE the bagplot is added to an existing plot
pch	sets the plotting character
cex	sets characters size
dkmethod	1 or 2, there are two method of approximating the bag, method 1 is very rough (only based on observations
precision	precision of approximation, default: 1
verbose	automatic commenting of calculations
debug.plots	if TRUE additional plots describing intermediate results are constructed
col.loophull	color of loop hull
col.looppoints	color of the points of the loop
col.baghull	color of bag hull
col.bagpoints	color of the points of the bag
transparency	see section details
show.center	if TRUE the center is shown
...	additional graphical parameters

Details

A bagplot is a bivariate generalization of the well known boxplot. It has been proposed by Rousseeuw, Ruts, and Tukey. In the bivariate case the box of the boxplot changes to a convex polygon, the bag of bagplot. In the bag are 50 percent of all points. The fence separates points within the fence from points outside. It is computed by increasing the the bag. The loop is defined as the convex hull containing all points inside the fence. If all points are on a straight line you get a classical boxplot. `bagplot()` plots bagplots that are very similar to the one described in Rousseeuw et al. Remarks: The two dimensional median is approximated. For large data sets the error will be very small. On the other hand it is not very wise to make a (graphical) summary of e.g. 10 bivariate data points. In case you want to plot multiple (overlapping) bagplots, you may want plots that are semi-transparent. For this you can use the `transparency` flag. If `transparency==TRUE` the alpha layer is set to '99' (hex). This causes the bagplots to appear semi-transparent, but ONLY if the output device is PDF and opened using: `pdf(file="filename.pdf",version="1.4")`. For this reason, the default is `transparency==FALSE`. This feature as well as the arguments to specify different colors has been proposed by Wouter Meuleman.

Value

`compute.bagplot` returns an object of class `bagplot` that could be plotted by `plot.bagplot()`. An object of the bagplot class is a list with the following elements: `center` is a two dimensional vector with the coordinates of the center. `hull.center` is a two column matrix, the rows are the coordinates of the corners of the center region. `hull.bag` and `hull.loop` contain the coordinates of the hull of the bag and the hull of the loop. `pxy.bag` shows you the coordinates of the points of the bag. `pxy.outer` is the two column matrix of the points that are within the fence. `pxy.outlier` represent the outliers. The vector `hdepths` shows the depths of data points. `is.one.dim` is `TRUE` if the data set is (nearly) one dimensional. The dimensionality is decided by analysing the result of `prcomp` which is stored in the element `prdata`. `xy` shows you the data that are used for the bagplot. In the case of very large data sets subsets of the data are used for constructing the bagplot. A data set is very large if there are more data points than `approx.limit`. `xydata` are the input data structured in a two column matrix.

Note

Version of bagplot: 10/2012

Author(s)

Peter Wolf

References

P. J. Rousseeuw, I. Ruts, J. W. Tukey (1999): The bagplot: a bivariate boxplot, The American Statistician, vol. 53, no. 4, 382–387

See Also

`boxplot`

Examples

```
# example: 100 random points and one outlier
dat<-cbind(rnorm(100)+100,rnorm(100)+300)
dat<-rbind(dat,c(105,295))
bagplot(dat,factor=2.5,create.plot=TRUE,approx.limit=300,
   show.outlier=TRUE,show.looppoints=TRUE,
   show.bagpoints=TRUE,dkmethod=2,
   show.whiskers=TRUE,show.loophull=TRUE,
   show.baghull=TRUE,verbose=FALSE)
# example of Rousseeuw et al., see R-package rpart
cardata <- structure(as.integer( c(2560,2345,1845,2260,2440,
 2285, 2275, 2350, 2295, 1900, 2390, 2075, 2330, 3320, 2885,
 3310, 2695, 2170, 2710, 2775, 2840, 2485, 2670, 2640, 2655,
 3065, 2750, 2920, 2780, 2745, 3110, 2920, 2645, 2575, 2935,
 2920, 2985, 3265, 2880, 2975, 3450, 3145, 3190, 3610, 2885,
 3480, 3200, 2765, 3220, 3480, 3325, 3855, 3850, 3195, 3735,
 3665, 3735, 3415, 3185, 3690, 97, 114, 81, 91, 113, 97, 97,
 98, 109, 73, 97, 89, 109, 305, 153, 302, 133, 97, 125, 146,
 107, 109, 121, 151, 133, 181, 141, 132, 133, 122, 181, 146,
 151, 116, 135, 122, 141, 163, 151, 153, 202, 180, 182, 232,
 143, 180, 180, 151, 189, 180, 231, 305, 302, 151, 202, 182,
 181, 143, 146, 146)), .Dim = as.integer(c(60, 2)),
 .Dimnames = list(NULL, c("Weight", "Disp.")))
bagplot(cardata,factor=3,show.baghull=TRUE,
  show.loophull=TRUE,precision=1,dkmethod=2)
title("car data Chambers/Hastie 1992")
# points of y=x*x
bagplot(x=1:30,y=(1:30)^2,verbose=FALSE,dkmethod=2)
# one dimensional subspace
bagplot(x=1:100,y=1:100)
```

`bagplot.pairs` `pairs` *plot with bagplots*

Description

`bagplot.pairs` calls `pairs` and use bagplot() as panel function. It can be used for the inspection of data matrices.

Usage

```
bagplot.pairs(dm, trim = 0.0, main, numeric.only = TRUE,
              factor = 3, approx.limit = 300, pch = 16,
              cex = 0.8, precision = 1, col.loophull = "#aaccff",
              col.looppoints = "#3355ff", col.baghull = "#7799ff",
              col.bagpoints = "#000088", ...)
```

Arguments

dm	datamatrix, columns contain values of the variables
trim	fraction or vector of fractions of data points that should be removed from the variables before computing
main	title of the plot
numeric.only	if TRUE only numerical variables will be used. Otherwise an transformation to numeric will be performed.
factor	see help of bagplot
approx.limit	see help of bagplot
pch	see help of bagplot
cex	see help of bagplot
precision	see help of bagplot
col.loophull	see help of bagplot
col.looppoints	see help of bagplot
col.baghull	see help of bagplot
col.bagpoints	see help of bagplot
...	further arguments to be passed to pairs

Details

bagplot.pairs is a cover function which calls pairs and uses bagplot to display the data.

Value

The data which has been used for the plot.

Note

Feel free to have a look inside of bagplot.pairs and to improve it according to your ideas.

Author(s)

Peter Wolf

See Also

bagplot, pairs

Examples

```
# bagplot.pairs(freeny)
 # bagplot.pairs(trees,col.baghull="green", col.loophull="lightgreen")
```

`boxplot2D` *Boxplot of projection of two dimensional data*

Description

boxplot2D computes summary statistics of a one dimensional projection of a two dimensional data set and plots a sloped boxplot of the statistics into the scatterplot of the two dimensional data set.

Usage

```
boxplot2D(xy, add.to.plot = TRUE, box.size = 10, box.shift = 0,
angle = 0, angle.type = "0", tukey.style = TRUE, coef.out = 1.5,
coef.h.out = 3, design = "sl", na.rm=FALSE, ...)
```

Arguments

`xy`	`(nx2)`-matrix, two dimensional data set
`add.to.plot`	if `TRUE` the boxplot is added to the actual plot of the graphics device
`box.size`	height of the box (of the boxplot)
`box.shift`	shift of boxplot perpendicular to the projection direction
`angle`	direction of projection in units defined by angle.type
`angle.type`	`"0"`: angle in (0,2*pi), `"1"`: clock-like: angle.typ.0==2*pi*angle.typ.1/12, `"2"`: degrees: angle.typ.0==2*pi*angle.typ.2/360, `"3"`: by fraction: delta.y/delta.x
`tukey.style`	if `TRUE` outliers are defined as described in Tukey (1977)
`coef.out`	outliers are values that are more than `coef.out*boxwidth` away from the box, default: `coef.out=1.5`
`coef.h.out`	heavy outliers are values that are more than `coef.h.out*boxwidth` away from the box, default: `coef.h.out=3`
`design`	if `sl` then parallelogram else box
`na.rm`	if TRUE 'NA' values are removed otherwise exchanged by mean
`...`	additional graphical parameters

Note

version 08/2003

Author(s)

Peter Wolf

References

Tukey, J. *Exploratory Data Analysis.* Addison-Wesley, 1977.

See Also

boxplot

Examples

```
xy<-cbind(1:100, (1:100)+rnorm(100,,5))
par(pty="s")
plot(xy,xlim=c(-50,150),ylim=c(-50,150))
boxplot2D(xy,box.shift=-30,angle=3,angle.typ=1)
boxplot2D(xy,box.shift=20,angle=1,angle.typ=1)
boxplot2D(xy,box.shift=50,angle=5,angle.typ=1)
par(pty="m")
```

---

faces *Chernoff Faces*

Description

faces represent the rows of a data matrix by faces. plot.faces plots faces into a scatterplot.

Usage

```
faces(xy, which.row, fill = FALSE, face.type = 1, nrow.plot, ncol.plot,
    scale = TRUE, byrow = FALSE, main, labels, print.info = TRUE,
    na.rm = FALSE, ncolors = 20, col.nose = rainbow(ncolors),
    col.eyes = rainbow(ncolors, start = 0.6, end = 0.85),
    col.hair = terrain.colors(ncolors), col.face = heat.colors(ncolors),
    col.lips = rainbow(ncolors, start = 0, end = 0.2),
    col.ears = rainbow(ncolors, start = 0, end = 0.2), plot.faces = TRUE, cex = 2)
## S3 method for class 'faces'
plot(x, x.pos, y.pos, face.type = 1, width = 1, height = 1, labels,
        ncolors = 20, col.nose = rainbow(ncolors), col.eyes = rainbow(ncolors,
        start = 0.6, end = 0.85), col.hair = terrain.colors(ncolors),
        col.face = heat.colors(ncolors), col.lips = rainbow(ncolors,
        start = 0, end = 0.2), col.ears = rainbow(ncolors, start = 0,
        end = 0.2), cex = 2, ...)
```

Arguments

xy	xy data matrix, rows represent individuals and columns variables
which.row	defines a permutation of the rows of the input matrix
fill	if(fill==TRUE), only the first nc attributes of the faces are transformed, nc is the number of columns of xy
face.type	an integer between 0 and 2 with the meanings: 0 = line drawing faces, 1 = the elements of the faces are painted, 2 = Santa Claus faces are drawn
nrow.plot	number of columns of faces on graphics device

ncol.plot	number of rows of faces
scale	if(scale==TRUE), variables will be normalized
byrow	if(byrow==TRUE), xy will be transposed
main	title
labels	character strings to use as names for the faces
print.info	if TRUE information about usage of variables for face elements are printed
na.rm	if TRUE 'NA' values are removed otherwise exchanged by mean of data
plot.faces	if FALSE no face is plotted
cex	size of labels of faces
x	an object of class faces computed by faces
x.pos	x coordinates of positions of faces
y.pos	y coordinates of positions of faces
width	width of the faces
height	height of the faces
ncolors	number of colors in the palettes for painting the elements of the faces
col.nose	palette of colors for painting the nose
col.eyes	palette of colors for painting the eyes
col.hair	palette of colors for painting the hair
col.face	palette of colors for painting the face
col.lips	palette of colors for painting the lips
col.ears	palette of colors for painting the ears
...	additional graphical arguments

Details

Explanation of parameters: 1-height of face, 2-width of face, 3-shape of face, 4-height of mouth, 5-width of mouth, 6-curve of smile, 7-height of eyes, 8-width of eyes, 9-height of hair, 10-width of hair, 11-styling of hair, 12-height of nose, 13-width of nose, 14-width of ears, 15-height of ears.

For painting elements of a face the colors of are found by averaging of sets of variables: (7,8)-eyes:iris, (1,2,3)-lips, (14,15)-ears, (12,13)-nose, (9,10,11)-hair, (1,2)-face.

Further details can be found in the literate program of faces.

Value

list of two elements: The first element out$faces is a list of standardized faces of class faces, this object could be plotted by plot.faces; a plot of faces is created on the graphics device if plot.faces=TRUE. The second list is short description of the effects of the variables.

Note

version 01/2009

Author(s)

H. P. Wolf

References

Chernoff, H. (1973): The use of faces to represent statistiscal assoziation, JASA, 68, pp 361–368. The smooth curves are computed by an algorithm found in Ralston, A. and Rabinowitz, P. (1985): A first course in numerical analysis, McGraw-Hill, pp 76ff. http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/wolf_aplpack

See Also

—

Examples

```
faces()
faces(face.type=1)

faces(rbind(1:3,5:3,3:5,5:7))

data(longley)
faces(longley[1:9,],face.type=0)
faces(longley[1:9,],face.type=1)

plot(longley[1:16,2:3],bty="n")
a<-faces(longley[1:16,],plot=FALSE)
plot.faces(a,longley[1:16,2],longley[1:16,3],width=35,height=30)

set.seed(17)
faces(matrix(sample(1:1000,128,),16,8),main="random faces")

a<-faces(rbind(1:3,5:3,3:5,5:7),plot.faces=FALSE)
plot(0:5,0:5,type="n")
plot(a,x.pos=1:4,y.pos=1:4,1.5,0.7)
# during Christmastime
faces(face.type=2)
```

---

hdepth *hdepth of points*

Description

hdepth() computes the h-depths of points.

Usage

```
hdepth(tp, data, number.of.directions=181)
```

Arguments

tp	two column matrix of the coordinates of points which h-depths are needed
data	two column matrix of the coordinates of the points of a data set
number.of.directions	number of directions to be checked

Details

The function hdepth computes the h-depths of the points tp relative to data set data. If data is missing tp will also be taken as data set.

Value

the h-depths of the test points

Note

Version of bagplot: 12/2012

Author(s)

Peter Wolf

See Also

bagplot

Examples

```
# computation of h-depths
data <- cbind(rnorm(40), rnorm(40))
xy <- cbind(runif(50,-2,2),runif(50,-2,2))
bagplot(data); text(xy, as.character(hdepth(xy,data)))
```

---

iconplot *Icon Plots for Visualization of Contingency Tables*

Description

An icon plot is a graphical representation of a contingency table. iconplot(computes a icon plot of a data matrix (matrix or data frame) or of an object of class table. Based on argument grp.xy the data set is split into groups. Similarly the graphics region is divided into panels. Then the elements of the groups are visualized within the associated panels.

Usage

```
 iconplot(data
,  grp.xy = 2 ~ 1
,  grp.color = NULL
,  grp.icon = NULL
,  colors
,  icons
,  vars.to.factors
,  panel.reverse.y = FALSE
,  panel.space.factor = 0.05
,  panel.prop.to.size = c(FALSE, FALSE)
,  panel.margin = 0.03
,  panel.frame = TRUE
,  panel.adjust = c(0.5, 0.5)
,  icon.horizontal = TRUE
,  icon.stack.type = c("lt", "lb", "rt", "rb")[1]
,  icon.cex = NA
,  icon.aspect = 1
,  icon.stack.len = NA
,  icon.space.factor = 0.3
,  icon.grey.levels = 2
,  icon.frame = TRUE
,  icon.draft = TRUE
,  lab.side = c("bl", "br", "tl", "tr")[1]
,  lab.parallel = c(TRUE, TRUE)
,  lab.cex = 1
,  lab.boxes = 2
,  lab.color = c("#CCCCCC", "white")
,  lab.type = c("expanded", "compact")[2]
,  lab.n.max = c(20, 30)
,  lab.legend = c("cols","rows","skewed","horizontal","vertical")[2]
,  packer = c("icons", "numbers", "panel.legend", "stars")[1]
,  panel.text =  NULL
,  mar = rep(1, 4)
,  main
,  verbose = !TRUE
,  ...)
```

Arguments

`data`	a data matrix, a data frame or an object of class `table`. Note: If the column or dimension names are elements of the following set of reserved names: `.sign,.fraction,.color,.icon,.job.no,.x0,.x1,.y0,.y1` strange results may occur; therefore, avoid these variable names.
`grp.xy`	a formula specifying how the data set is divided into groups and defines in which panel an element of the data is represented. The formula `y1 ~ x1` means that the data set is split according to the levels of the variables `y1` and `x1` into groups. If `n.level.y1` and `n.level.x1` are the numbers of levels of the two variables

the plotting region is divided like a chessboard into `n.level.y1` rows and `n.level.x1` columns. In this way we get `n.level.y1 * n.level.x1` fields that are called panels. In each of the panels the elements of the associated group are represented by pictogram elements or icons.

If the argument `grp.xy` hasn't been set by default the first variable of the data set defines the grouping of the data along the x-axis and the second one the grouping along the y-axis.

Instead of variable names the indices of the variables can be used.

The definition of recursive groupings is allowed and is expressed by operator `"+"`: `y ~ 3 + 4` means that the horizontal range of the graphical region is split twice: At first the segmentation of the region is computed according to variable 3, in the second step the subranges of step 1 will be divided as a function of the levels of variable 4.

A `"0"` on one side of the ~ character indicates that no splitting of the correspondent region is desired.

`grp.color` defines how the data are grouped with respect to coloring. The name of the variable used for coloring the icons (or pictogram elements) has to be assigned to `grp.color`. `colors[i]` defines the color of the icon belonging to the level `i` of the variable fixed by `grp.color`.

`grp.icon` defines how the data are grouped with respect the associated icon. The name of the variable used for selecting symbols or icons has to be fixed by argument `icons`. The symbol (icon) representing an observation depends on its level of the variable specified by `grp.icon`.
If additional variables – separated by a + character – are found the values of these variables will be used in the call of a icon generating function. For details see paragraph 'Details'.

`colors` set of colors used for pictogram elements.

`icons` defines the icons or the set of icons used by `iconplot`. If `icons` is a vector `icons[i]` is used to represent the observations whose level of the variable fixed by `grp.icon` is `i`. There are some alternatives to define the icons or pictogram elements:
* default: the default symbol is a rectangle.
* vector of numbers: numbers specify plotting characters of the graphics system similar to `points(...,pch = 13)`.
* list of raster images: the images are used as icons.
* character vector: `icons[i]` with an extension indicating a `pnm`, `ppm`, `jpg` or `png` image file: `iconplot` tries to use the image of the file as icon.
Otherwise `icons[i]` is interpreted as the name of an internal icon generating function.
* list of functions: `icons[i]` is interpreted as an icon generating function and is called to compute the icon for level `i`.
* list of icon descriptions. For details see paragraph "Details".
Note: If an image file is defined by an internet link it is temporarily downloaded using `tempfile()` and `download.file()`.
Note: Mixtures of these alternative definition don't work usually. Therefore, it is recommended to use one type of definition only.

vars.to.factors
: controls the transformation of variables to factors. If missing it is set to TRUE for each of the relevant variables. If vars.to.factors is a vector and if its elements don't have names the variables 1:length(vars.to.factors) are transformed.
If vars.to.factors consists of named elements the names indicate the variables to be transformed.
If vars.to.factors[i] == FALSE variable i will not be transformed.
If vars.to.factors[i] == 1 variable i is transformed to a factor.
If vars.to.factors[i] < 1 the range of variable i is cut into groups in a way that we approximately get round(1/vars.to.factors[i]) groups and each of the groups approximately contain 100 * vars.to.factors[i] percent of the data. If vars.to.factors[i] > 1 the range of variable i will be cut into floor(vars.to.factors[i]) subranges of equal size and you get a factor variable with floor(vars.to.factors[i]) levels.

panel.reverse.y
: logical, if TRUE the vertical axis is reversed.

panel.space.factor
: relative space inserted between the panels.

panel.prop.to.size
: a vector containing two elements which controls the sizes of the panels. The first entry determines the widths of the panels and the second one their heights. panel.prop.to.size[1] == 0 means all panels are of the same width. If panel.prop.to.size[2] == 0 the panels are of the same height. A value of 1 indicates that sizes should be computed proportional to the frequencies of the levels. Otherwise the sizes of the panels are fixed proportional to: frequencies^panel.prop.to.size.

panel.margin
: controls the margins around the regions of the panels. If this argument is a vector of length four the elements refer to the four sides of the plot: bottom, left, top, and right. If this argument is set to c(0,0.1,0.5,0) we get no additional margin below the panels and on the right-hand side. However, there will be an upper margin of size 100 * 0.5 percent of the height of the area containing the panels and a margin of size code100 * 0.1 percent of the width on the left-hand is provided.

panel.frame
: logical, if TRUE a border line is drawn around each of the panels.

panel.adjust
: controls the adjustment of the panels within their regions. This argument modifies the internal coordinates and do usually not change the appearance of the plot.

panel.text
: vector of strings. The text panel.text[i] is written into panel[i]. The texts can be used for short describitions of the contents of the panels. To get an idea of the numbering of the panels you can set panel.text = 1:20.

icon.horizontal
: logical, if TRUE the stacks of icons or pictogram elements are plotted horizontally. This argument effects the way how icons are put into the panels.

icon.stack.type
: defines the method of plotting the stacks of icons: "r" or "l" are shortcuts for "right" or "left". "t", "b" correspond to "top" and "bottom", respectively. Note:

Fractional parts of frequencies are represented by smaller icons. Adding the letter "s" (as a abbreviation for "shrinkage") to the argument icon.stack.type both dimensions of the icons are reduced. If icon.stack.type is a vector its elements define the different types of stacking for the panels.

icon.cex — size of icons; this argument is similar to cex of points().

icon.aspect — aspect ratio of icons: width / height.

icon.stack.len — maximal number of icons gathered to build a stack. If this length is decreased the number of stacks (rows or columns of icons) will increase.

icon.space.factor — relative space between two icons.

icon.grey.levels — controls the coloring of icons of class raster or images. If icon.grey.levels is of length 1 and if it is an integer the argument defines the number of color- or grey-levels of the icons. If icon.grey.levels consists of two or more values between 0 and 1 they are interpreted as grey limits. Level 0 represents "black", and 1 indicates the maximal brightness. The default setting generates one color only (besides black and white).

icon.frame — logical, if TRUE a border is drawn around each of the pictograms.

icon.draft — logical, if TRUE raster images are generated by calling rasterImage() with the setting interpolate = TRUE.

mar — this argument is delivered to the graphics device via par() and manipulates the margins of the plot.

main — defines the title of the plot.

lab.side — defines one or two sides that are used for margin information: "l" indicates the "left" side, "b" identifies the "bottom" as well as "r" the "right" and "t" the "top" side.

lab.parallel — logical, if FALSE margin labels are perpendicularly constructed to the axes. If lab.parallel is a vector the first element is used for controlling the labels of the bottom or top side and the second one specifies the orientation of the y-labels. If one elements is set to 0.5 the labels of the last xy grouping variable are printed perpendicularly only.

lab.legend — a character string indicating the kind of legend out of the vector c("cols","rows","skewed","horizontal","vertical"). Assigning a number of the set 1:5 to the argument is interpreted as an index of the set of the five types of legends.
"cols": vertical legends, side by side at the bottom side of the plot.
"rows": horizontal legends, line by line at the bottom side of the plot.
"skewed": horizontal legends, line by line and the level names are rotated.
"horizontal": horizontal legends, side by side at the bottom side of the plot.
"vertical": vertical legends, line by line at the right side of the plot.

lab.cex — sets the size of the characters of the labels and the legends.

lab.boxes — defines the types of boxes around the margin labels: lab.boxes == 0: no boxes are drawn.

	`lab.boxes >= 1`: small boxes around the labels are drawn. `lab.boxes >= 2`: big boxes around the labels are drawn. `lab.boxes %% 1`: defines the size of the separation line between the names of the variables and the names of the levels.
`lab.color`	The first element defines the color of the box containing the names of variables or levels in the margins. The second element sets the color of the separation line between the variable names and the level names within the margins.
`lab.type`	defines the design style of margin labeling: "c" or "e" are shortcuts for "compact" or "expanded".
`lab.n.max`	is an integer vector consisting of three elements. The first element sets the number of characters during printing the labels of the levels. The second element defines the maximal number of level names to be plotted in the margins. `lab.n.max[3]` limits the number of labels of the color- or icon-legend.
`packer`	defines the packer(s) which are used to fill the panels. If "icons" is an element of `packer` the observations will be represented by icons, pictogram elements or symbols. If the character string "numbers" is found in `packer` in each of the panels the numbers of its observations will be printed into the areas of the panels. The packer "panel.legend" plots the level combinations into the panels. This may be a useful feature as long as the number of the panels is small. Otherwise the texts of level combinations will overlap each other. The argument `cex` controls the size of the text strings.
`verbose`	logical, if `TRUE` internal information is printed during the computation.
`...`	arguments that will be passed to the graphics functions and suitable ones to the icon generating functions.

Details

`iconplot()` constructs an icon plot of a data matrix and a contingency table. In an icon plot each observation of the data set is represented by a small symbol or an image called pictogram or icon. A cell of a contingency table is visualized by a set of icons. The icons of a cell are plotted within a rectangular region which we call panel and an icon plot consists of a lot of panels containing the icons of the cells.

Similar to the layout of contingency tables the set of panels are arranged in a grid-like manner. Considering a high dimensional contingency table you can concentrate on some of the variables and can construct suitable margin tables. Equivalently you can build a lot of icon plots to emphasize your viewpoint. By varying the actual arguments of `iconplot()` a huge set of appearances of plots results and the nicest one for your purpose can be choosen. `table`, `matrix` or `data frames` can be used as data input of `iconplot()`. Tables are allowed to have fractional or negative entries; these cases may occur by computing the difference of two tables or by changing the units of counting. Internally a table will be expanded to a data matrix. Fractional numbers are coded in a data matrix by the additional column or variable `.fraction`, to handle negative numbers the new variable `.sign` is added.

The argument `grp.xy` of `iconplot` defines the variables used for grouping and splitting the data dependent on the levels of the specified variables. Each group is represented within a panel as stated above. Let's have a look at an example: Consider you have a 2x3 contigency table and

would like to represent it by an icon plot. So a plot to be constructed should have 2x3 panels and the number of icons of the panels should be given by the cell entries. To get an icon plot with desired panel structure you define the xy-grouping by `grp.xy = 1 ~ 2`. This means: The data set has to be split according to the two levels of the first variable and the y-range of the plot has to be divided in two rows of panels. On the other side the second variable defines the grouping concerning the the x-range and three columns of panels appear. As a result a icon plot is generated that consists of six panels arranged in two rows and three columns. The panels of a fixed level of the first variable are placed side by side, whereas the panels of a fixed level of the second variable are stacked one upon the other and a layout known from a chessboard results. As an example try: `x <-as.table(matrix(1:6,2,3)); iconplot(x,grp.xy = 1 ~ 2) grp.xy = 0 ~ 1 + 2` leads a double grouping on the x-axis and no vertical grouping. `grp.xy = 1 + 2 ~ 3 + 4` presums four or more variables and splits the graphics region twice along the x- and twice along the y-direction.

Within a panel the entry of one cell is represented. Several arguments control the way how the icons are placed in a panel. The absolute size of the icons can be defined by `icon.cex`. `icon.aspect` fixes the aspect ratio of the pictograms (width / height). The elements in a panel are assembled into stacks; the maximal length of these stacks can be set by `icon.stack.len`; horizontal stacks are plotted if `icon.horizontal` is `TRUE`. Framing icons and spacing between them is controlled by the arguments `icon.frame` and `icon.space.factor`.

The icons or pictogram elements may be colored dependent on the levels of a variable. The variable has to be established by argument `grp.color`. A set of colors can be defined by argument `colors`. Accordingly, the symbols or images are determined by `grp.icon` and `icons`.

An icon or pictogram element can be generated by an icon generating function. The result of an icon generating function describes a standardized icon by a set of segments, polygons, splines and texts which are combined in a list. `segments`: `segments` are defined by a matrix or a data frame of 5 or 6 columns: Columns 1 to 4 keep the coordinates of the starting and ending points of the segments: `x.0,y.0,x.1,y.1`.
The 5th column contains the widths of the segments. The coordinates and the widths have to be choosen in a way that the icon fits pretty well into a plotting field of size 100mm x 100mm assuming the coordinates of the world window defined by: `usr = c(0,100,0,100)`.
If the 6th column is available it defines the coloring of the segments. A value of "0" codes the color "white" and the other values are interpreted as usually: "1" means "black" and any other color is processed as `col` in `points`, for example. An `NA` value instead of a color instructs `iconplot()` to color the segment dependent on the associated level of the variable fixed by `grp.color`. Segment objects must have the class attribute `"segments"`.

`polygon`: Polygons are defined by a matrix or data frame of 2 or 3 columns. Colums 1 and 2 store the coordinates of the vertices of the polygon. A third column fixes the coloring of the polygon. The class attribute of this kind of element has to be set to `"polygon"`.

`spline`: Splines are defined by a matrix or data frame of 3 or 4 columns. Colums 1 and 2 store the coordinates of the points which form the basis of the spline. The third column keeps the line width of the curve. The optional fourth column shows how to color the spline. Splines are identified by class attribute `"spline"`.

`text`: Text elements of a generated icon are defined by a data frame of 3, 4 or 5 columns. The first two columns of the object store the coordinates of the positions of the text(s). The third element stores the text information and the fourth is used to set the size of the characters. The fifth fixes the coloring of the text. The class attribute of a text element is `"text"`. There are some internal icon generating functions. Here is a list of them:

BI,TL,cross.simple,cross,circle.simple,circle,car.simple,car,nabla, walkman,smiley.blueeye,smiley.normal,smiley,smiley.sad,mazz.man,bike, bike2,heart,bend.sign,fir.tree,comet,coor.system.

Value

iconplot() returns a list consisting of three elements. The first element is the matrix jobs whose lines show some attributes of the panels. In a row of this matrix you find the number of the panel .job.no and the location of the panel (in user coordinates: xmins,xmaxs,ymins,ymaxs). The second element is a copy of the modified data matrix which is used for the construction of the icon plot: Besides the data delivered by the user there are columns showing the colors, icons and coordinates of the pictogram elements. The third element contains the output of par() and describes the graphics device during the computation; this list differs from the parameter settings after leaving iconplot() because the state of graphics parameter before calling iconplot() is restored. These three lists may be helpful if you want to add further graphical elements to the plot.

Note

Remark: the version of iconplot of this package is an experimental version. Therefore, in the future some of the features may be changed and it is not sure that the function works as described on all types of graphics devices. In case of errors feel free to write a mail. Additional information and examples are found on the web page
http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/wolf_aplpack.

Author(s)

Hans Peter Wolf

See Also

mosaicplot, pairs, puticon

Examples

```
# HairEyeColor data, grouping by color
iconplot(HairEyeColor,
         grp.color         = 1,
         grp.xy            = NULL,
         colors            = c("black", "brown", "red", "gold"),
         icon.space.factor = 0,
         icon.aspect       = 2,
         main = "grouping by color")
# HairEyeColor data, grouping by color and symbols
iconplot(HairEyeColor,
         grp.icon          = "Sex",
         grp.color         = "Hair",
         grp.xy            = NULL,
         colors            = c("black", "brown", "red", "gold"),
         icons             = 18:17,
```

```
          icon.frame        = FALSE,
          lab.cex           = 0.8,
          icon.space.factor = 0,
          lab.parallel      = !FALSE,
          main = "grouping by color and icons")
# HairEyeColor data, grouping by x and color
iconplot(HairEyeColor,
          grp.xy            = "0 ~ 1",
          grp.color         = 2,
          colors            = c("black", "brown", "red", "gold"),
          icon.stack.type   = "tr",
          icon.space.factor = c(0, 0.4),
          lab.cex           =0.7,
          main = "grouping by x and by colors")
# 2-dim, 1 split in y, 1 split in x, grouping by color
iconplot(HairEyeColor,
          grp.xy          = "1 ~ 3",
          grp.color       = 2,
          colors          = c("brown", "blue", "brown3", "green"),
          panel.frame     = FALSE,
          icon.stack.type = "bl",
          lab.cex         = 0.7,
          main = "grouping by x and y and by colors")
# 3-dim, 2 splits in x, 1 split in x, margin labs on the right
iconplot(HairEyeColor,
          grp.xy             = "2 ~ 1 + 3 ",
          grp.color          = 2,
          panel.space.factor = c(0, .1),
          panel.margin       = c(.05,.03,.03,.01),
          icon.stack.type    = "lb",
          icon.stack.len     = 7,
          icon.frame         = FALSE,
          icon.space.factor  = .0,
          lab.parallel       = c(TRUE, FALSE),
          lab.color          = c("lightblue","green"),
          lab.side           = "br",
          lab.boxes          = 0.2,
          lab.type           = "compact",
          lab.cex            = 0.8,
          main = "grouping: 2~1+3 and by color, margin labs variations")
# 3-dim, 3 splits in y, icon.aspect = NA
iconplot(HairEyeColor,
          grp.xy             = "3 + 2 ~ 1",
          grp.color          = 3,
          panel.margin       = 0,
          panel.space.factor = 0.1,
          icon.stack.type    = "lb",
          icon.horizontal    = TRUE,
          icon.stack.len     = 5,
          icon.space.factor  = c(.1, .3),
          icon.aspect        = NA,
          icon.frame         = FALSE,
          lab.boxes          = 0.3,
```

```
         lab.color          = "grey",
         lab.side           = "tl",
         lab.parallel       = TRUE,
         lab.cex            = 0.7,
         lab.type           = "compact",
         main = "grouping: 3 + 2 ~ 1 and by color")
# 3-dim, plotting characters as icons
data <- as.table(array(0:23, 2:4))
iconplot(data,
         grp.xy          = 1 + 2 ~ 3,
         grp.color       = 3,
         grp.icon        = 2,
         icon.aspect     = 2,
         icon.horizontal = TRUE,
         icons           = 15:18,
         icon.stack.type = c("lb", "lt", "rb","rt")[3],
         icon.frame      = FALSE,
         lab.cex         = .6,
         lab.type        = "compact",
         main = "1+2 ~ 3")
# 3-dim contingency table: panels of different sizes, 1 split in y, 2 in x
# packer numbers
  ## because of computation time
iconplot(Titanic,
         grp.xy             = 1~3+2,
         grp.color          = 1,
         packer             = c("icons", "numbers"),
         panel.prop.to.size = 0.5,
         panel.frame        = !TRUE,
         panel.margin       = .01,
         icon.aspect        = 0.15,
         icon.stack.type    = "lt",
         icon.space.factor  = 0.0,
         icon.frame         = FALSE,
         lab.side           = c("bl","br","tl","tr")[1],
         lab.type           = "compact",
         lab.cex            = 0.8,
         lab.boxes          = 1.1,
         lab.color          = "lightgreen",
         lab.parallel       = TRUE,
         main = "different sizes of panels")

# 3-dim contingency table: panels of different sizes, 3 splits in y
  ## because of computation time
iconplot(Titanic,
         grp.xy             = "4 + 3 + 1 ~ 0" ,
         grp.color          = 4,
         colors             = c("green", "red"),
         packer             = c("icons", "numbers"),
         panel.frame        = FALSE,
         panel.margin       = .01,
         panel.prop.to.size = .3,
         panel.space.factor = 0.05,
```

```
          panel.reverse.y    = TRUE,
          icon.space.factor  = 0.5,
          lab.side           = "l",
          lab.type           = "compact",
          lab.parallel       = c(FALSE, TRUE),
          lab.cex            = 0.7,
          main = "Titanic data, different sizes of panels")

#  3-dim contingency table: panels of different sizes
  ## because of computation time
iconplot(Titanic,
          grp.xy             = "0 ~ 4 + 3 + 1 " ,
          grp.color          = 4,
          colors             = c("green", "red"),
          panel.frame        = FALSE,
          panel.margin       = .01,
          panel.prop.to.size = .2,
          panel.space.factor = 0.05,
          panel.reverse.y    = TRUE,
          icon.space.factor  = 0.5,
          lab.side           = "b",
          lab.type           = "compact",
          lab.boxes          = 0.2,
          lab.parallel       = c(FALSE, TRUE),
          lab.cex            = 0.6,
          lab.color          = c("lightblue"),
          main = "Titanic data, different widths of panels")

# 3-dim contingency table: panels of different sizes, 3 splits in x
  ## because of computation time
iconplot(Titanic,
          grp.xy             = 3 + 2 ~ 1,
          grp.color          = 2,
          panel.prop.to.size = 0.66,
          icon.space.factor  = 0.4,
          panel.space.factor = 0.1,
          lab.type           = "c",
          lab.cex            = 0.7,
          lab.boxes          = 1.2,
          lab.color          = c("lightblue"),
          main = "Titanic: panel.prop.to.size = 0.66")

# comparing iconplot and mosaic plot
# par(mfrow = 2:1)
iconplot(HairEyeColor,
          grp.xy             = 2 ~ 1 + 3 ,
          lab.parallel       = c(TRUE, TRUE),
          colors             = "red",
          panel.reverse.y    = TRUE,
          panel.prop.to.size = TRUE,
          icon.space.factor  = 0.5,
          icon.aspect        = 2,
          lab.cex            = .6,
```

```
         lab.boxes          = 1,
         lab.color          = "grey",
         # lab.side           = "lt",
         panel.margin       = c(0.00,.035,0.0,.050),
         main = 'HairEyeColor: grp.xy = 2 ~ 1 + 3')
mosaicplot(HairEyeColor)
# par(mfrow = c(1,1))
# relative frequences
data <- as.table(Titanic / max(Titanic))
iconplot(data,
         grp.xy             = 1 ~ 2 + 3,
         grp.color          = 4,
         panel.frame        = FALSE,
         panel.space.factor = 0.05,
         icon.horizontal    = !TRUE,
         icon.space.factor  = 0.103,
         icon.stack.type    = "b",
         icon.aspect        = 0.5,
         main = "Titanic: relative frequencies", colors = c("black", "green"))
# negative and fractional cell entries
  ## because of computation time
data <- HairEyeColor; Exp <- margin.table(data, 1)
for( d in 2:length(dim(data)) ){
  Exp <- outer( Exp, margin.table(data, d) ) / sum(data)
}
Diff <- Exp - data
cat("observed:\n"); print(data)
cat("expected:\n"); print(round(Exp, 3))
cat("deviation: expected - observed:\n"); print(round(Diff,3))
iconplot(Diff,
         grp.xy          = 1 + .sign ~ 2 + 3,
         grp.color       = ".sign",
         colors          = c( "red", "green"),
         panel.reverse.y = TRUE,
         panel.frame     = FALSE,
         icon.stack.type = c("t","b"),
         lab.boxes       = 1.2,
         lab.color       = "lightgreen",
         main = "deviations from expectation: HairEyeColor")

# relative differences of expectations, split according sign
data <- margin.table(Titanic, c(2,1,4)); pT <- prop.table(data)
eT <- outer(outer(margin.table(pT,1), margin.table(pT,2)), margin.table(pT,3))
data <- as.table(pT - eT); data <- data / max(data)
iconplot(data,
         grp.xy = Survived + Sex + .sign ~ Class,
         grp.color          = ".sign",
         panel.frame        = FALSE,
         panel.reverse.y    = TRUE,
         panel.space.factor = 0.05,
         icon.horizontal    = !TRUE,
         icon.stack.type    = rep(c("t","b"), 2),
         icon.aspect        = 2,
```

```
         icon.space.factor  = 0.1,
         lab.boxes          = 1.2,
         lab.color          = "lightgrey",
         main = "Titanic: difference to expectation")
# using a foto as icon, rentals of flats in Goettingen 2015/12
rentels <-
 structure(list(Rooms = c(2, 3, 2, 2, 3, 2, 2, 3, 2, NA, 2, 2,
 3, 4, 4, NA, 3, 2, 3, 2, 4, 2, 1, 2), qm = c(43.13, 86, 48, 66.62,
 76, 49, 59, 97, 45, 87, 46.39, 71, 65, 100, 75, 178, 94.07, 56,
 97, 70, 132, 43, 24, 48), Eur = c(365, 480, 480, 660, 500, 410,
 440, 1200, 450, 696, 420, 710, 747.5, 1300, 450, 990, 900, 520,
 1020, 1005, 924, 610, 375, 420)), class = "data.frame",
 row.names = c(NA, 24L))
fname <- system.file("src", "tm1.jpg", package="aplpack") # fname <- "tm1.jpg"
print(fname)
iconplot(rentels,
         grp.xy              = Eur ~ qm,
         vars.to.factors     = c(1, .5, .3),
         panel.frame         = FALSE,
         panel.space.factor  = 0.2,
         panel.prop.to.size  = 0.7,
         icons               = fname,
         icon.frame          = FALSE,
         icon.space.factor   = 0.05,
         lab.parallel        = c(TRUE, TRUE),
         lab.legend          = "cols",
         main = "rentels of flats in Goettingen 2015/12")
# size by .fractions, color by rooms
data <- cbind(rentels, .fraction = (rentels[,3] / max(rentels[,3]))^.5)
iconplot(data,
         grp.xy             = Eur ~ qm,
         grp.color          = Rooms,
         vars.to.factors    = c(1,.5, .3),
         panel.frame        = FALSE,
         panel.space.factor = 0.1,
         panel.prop.to.size = 0.7,
         icons              = fname,
         icon.stack.type    = "s",
         icon.frame         = FALSE,
         icon.space.factor  = 0.05,
         lab.cex            = 0.8,
         main               = "size fby .fractions, color by rooms")
# jpg files as icons
  ## because of computation time
data <- as.table(Titanic[2:3,,,,drop=FALSE]) / 10
fname1 <- system.file("src", "walkman-r.jpg", package="aplpack") # fname1 <- "walkman-r.jpg"
fname2 <- system.file("src", "pw-esch.jpg", package="aplpack")   # fname2 <- "pw-esch.jpg"
p.set <- c(fname1, fname2)
iconplot(data,
         grp.xy             = 2 ~ 3+1,
         grp.color          = 1,
         grp.icon           = 3,
         icons              = p.set,
```

```
         colors             = c("blue", "green"),
         panel.space.factor = 0.05,
         panel.prop.to.size = c(.5, .5, 1),
         icon.aspect        = 1,
         icon.space.factor  = .10,
         icon.horizontal    = TRUE,
         icon.draft         = FALSE,
         icon.stack.type    = c("lb", "lt", "rb","rt")[1],
         icon.grey.levels   = list(2, 10),
         lab.side           = "t", lab.cex = .7,
         main = "walkman and pw icons, scaled subset of Titanic")

# files of different types as icons
  ## because of computation time
fname3 <- system.file("src", "pw-esch.ppm", package="aplpack") # fname3 <- "pw-esch.ppm"
fname4 <- system.file("src", "pw-esch.png", package="aplpack") # fname4 <- "pw-esch.png"
p.set <- c(fname2, fname3, fname4)
iconplot(trees,
         grp.xy             = Girth ~ Height,
         grp.icon           = Height,
         grp.color          = Volume,
         vars.to.factors    = c(Volume = 4, Girth = 3, Height = 3),
         panel.space.factor = 0.05,
         panel.prop.to.size = c(.7, .45),
         panel.frame        = FALSE,
         icons              = p.set,
         icon.cex           = 14,
         icon.grey.levels = 6, icon.space.factor   = 0.05 )

# using raster graphics objects as icons
data <- as.table(Titanic[1:2,,,,drop=FALSE])/10
image1 <- as.raster(  matrix( c(1,0,1,1,0,1,1,0,1), ncol = 3, nrow = 3))
image2 <- as.raster(  matrix( c(1,0,1,0,0,0,1,0,1), ncol = 3, nrow = 3))
iconplot(data,
         grp.xy            = 2 ~ 4+1,
         grp.color         = 1,
         grp.icon          = 4,
         colors            = c("blue", "green"),
         icons             = list(image1, image2),
         icon.aspect       = 1,
         icon.space.factor = .10,
         icon.horizontal   = TRUE,
         icon.draft        = FALSE,
         icon.stack.type   = c("lb", "lt", "rb","rt")[1],
         icon.grey.levels  = list(2, 10),
         lab.side = "t", lab.cex = .7, main = "some Titanic data")
# using internal generator "fir.tree"
  ## because of computation time
data <- trees
iconplot(data,
         grp.color          = 3,
         grp.xy             = 1 ~ 2,
         vars.to.factor     = c(5, 5, 8),
```

```
          icons              = "fir.tree",
          colors             = rainbow(8, start = .1, end = .5),
          icon.frame         = FALSE,
          lab.legend         = 2,
          lab.cex            = 0.7,
          main = "grouping by vars and by colors")

# using different internal generators
data <- trees
iconplot(data,
          grp.color          = 1,
          grp.xy             = 1 ~ 2,
          grp.icon           = 2,
          colors             = c("orange", "green", "orange", "red"),
          icons = c("nabla", "BI", "walkman", "car.simple", "bike", "circle"),
          vars.to.factor     = c(3,6),
          lab.legend         = 2,
          lab.cex            = 0.7,
          main = "grouping by vars, by icons and by colors")
# Traveller plot proposed by M. Mazziotta and A. Pareto
Mazzi.Pareto <-
 structure(list(Region = c("Piemonte", "Valle d'Aosta", "Lombardia",
 "Trentino-Alto Adige", "Veneto", "Friuli-Venezia Giulia", "Liguria",
 "Emilia-Romagna", "Toscana", "Umbria", "Marche", "Lazio", "Abruzzo",
 "Molise", "Campania", "Puglia", "Basilicata", "Calabria", "Sicilia",
 "Sardegna"), Mean = c(98.74, 104.07, 101.38, 106.1, 104.38, 105.55,
 102.76, 103.62, 101.84, 103.52, 102.05, 97.88, 102.9, 91.43,
 94.12, 96.78, 93.55, 92.59, 96.29, 100.45), Penalty = c(0.43,
 4.23, 0.64, 0.63, 0.77, 0.34, 0.29, 0.46, 0.27, 0.22, 0.15, 0.82,
 1.3, 1.02, 0.37, 0.21, 2.37, 0.51, 0.31, 0.76), MPI = c(98.3,
 99.84, 100.74, 105.47, 103.61, 105.21, 102.47, 103.16, 101.57,
 103.3, 101.9, 97.06, 101.6, 90.42, 93.75, 96.58, 91.18, 92.08,
 95.98, 99.69)), .Names = c("Region", "Mean", "Penalty", "MPI"
 ), row.names = c(NA, -20L), class = "data.frame")
dm <- cbind(Mazzi.Pareto,
            col = as.factor(rep(1:4, 5)),        # as.factor!!
            row = as.factor(rep(1:5, each = 4))) # as.factor!!
iconplot(dm, verbose = !TRUE, x.text = 60,  y.text = -10,  #t3s
          grp.xy            = row ~ col,
          grp.icon          = 0 + Mean +  Penalty + Region,
          vars.to.factor    = FALSE,
          icons             = "mazz.man",
          panel.reverse.y   = TRUE,
          icon.space.factor = 0,
          icon.frame        = FALSE,
          lab.parallel      = TRUE,
          lab.side          = c("",""),
          main = "Traveller plot")
# definition of a check list, tally or 'Krebholz'
check.list <- function(x, colors = rainbow(length(x))){
  num.split <- function(x, div = 5){
    x.name <- as.character(substitute(x))
    xn <- lapply( x, function(x)
```

```
        c(rep(div, x %/% div), if( 0 < ( h <- x %% div) ) h )
      )
      len <- max(sapply(xn, length))
      xn <- lapply( xn, function(x) c(x, rep(0, len - length(x) )))
      xn <- matrix( unlist(xn), ncol = len, byrow = TRUE )
      xn <- as.table(xn)
      dimnames(xn) <- list( seq( along = x ), 1:len)
      names(dimnames(xn)) <- c(x.name, "Blocks")
      xn
    }
    x.split <- num.split(x)
    rownames(x.split) <- paste(sep = ":", 1:length(x), x)
    iconplot(x.split,
             grp.xy             = 1 ~ 2,
             grp.col            = 1,
             colors             = colors,
             panel.space.factor = c(0.4, 0.3),
             panel.frame        = FALSE,
             icon.stack.len     = 5,
             icon.space.factor  = c(0.4, 0),
             icon.asp           = NA,
             icon.frame         = FALSE,
             lab.side           = "1",
             lab.cex            = 0.7,
             main = paste("score of", substitute(x)))
    x.split
  }
  set.seed(13); data <- sample(1:50, size = 15)
  check.list(data)
```

`plothulls` *plothulls for data peeling*

Description

`plothulls` plots convex hulls of a bivariate data set.

Usage

```
plothulls(x, y, fraction, n.hull = 1, main, add = FALSE, col.hull,
    lty.hull, lwd.hull, density = 0, ...)
```

Arguments

`x`	two column matrix of the coordinates of points of x-values of a data set
`y`	if x is one dimensional then y contains the y-values of the data set
`fraction`	... of points that lies inside the hull to be plotted
`n.hull`	number of directions sequential hulls to be plotted

main	title for the graphics
add	if TRUE no new plot is initialized
col.hull	color(s) of the hull(s)
lty.hull	line type(s) of the hull(s)
lwd.hull	line width(s) of the hull(s)
density	density argument of polygon() that draws the hulls
...	further arguments used in the call of plot() or points()

Details

The function plothulls computes hulls of a bivariate data set using the function chull. After finding a hull the hull maybe plotted. Then the data points of the hull will be removed and the hull of the remaining points is computed. The style of plotting a hull depends on the setting of col.hull, lty.hull, lwd.hull and density. density=NA has the effect that the regions of the hulls are filled by a color. Using fraction you can plot a single hull. n.hull defines the number of hull that should be drawn one after the other.

Value

The hull(s) are stored as a list of matrices with two columns, the innermost first and so on.

Note

Version of plothulls: 10/2013

Author(s)

Peter Wolf

References

Green, P.J. (1981): Peeling bivariate data. In: Interpreting Multivariate Data, V. Barnett (ed.), pp 3-19, Wiley. Porzio, Giovanni C., Ragozini, Giancarlo (2000): Peeling multvariate data sets: a new approach. Quanderni di Statistica, Vol. 2.

See Also

bagplot

Examples

```
# 10 hulls computed from the faithful data and plotted
plothulls(faithful, n.hull=10, lty.hull=1)
# plotting additionally a hull with 90 percent of points within the hull
plot(faithful)
plothulls(faithful, fraction=.90, add=TRUE, col.hull="red", lwd.hull=3)
# hull with 10 percent of points within the hull
plothulls(faithful, fraction=.10, col.hull="red", lwd.hull=3)
# first 3 hulls of the cars data set
```

```
n <- 3
plothulls(cars, n.hull=n, col.hull=1:n, lty.hull=1:n)
# 5 hulls represented by colored regions
n <- 5
cols <- heat.colors(9)[3:(3+n-1)]
plothulls(cars, n.hull=n, col.hull=cols, lty.hull=1:n, density=NA, col=0)
points(cars, pch=17, cex=1)
# 6 hulls: regions colored and boundaries shown
n <- 6
cols <- rainbow(n)
plothulls(cars, n.hull=n, col.hull=cols, lty.hull=1:n, density=NA, col=0)
plothulls(cars, n.hull=n, add=TRUE, col.hull=1, lwd.hull=2, lty=1, col=0)
```

`plotsummary` *graphical summaries of variables of a data set*

Description

`plotsummary` shows some important characteristics of the variables of a data set. For each variable a plot is computed consisting of a barplot, an ecdf, a density trace and a boxplot.

Usage

```
plotsummary(data, trim = 0, types = c("stripes", "ecdf", "density", "boxplot"),
            y.sizes = 4:1, design = "chessboard", main, mycols = "RB")
```

Arguments

`data`	Data set for computing a graphical summary.
`trim`	`trim` defines the fraction of observation for trimming on both ends of the data.
`types`	vector of types of representation of the data set. The elements of the vector will induce small plots which are stacked in vertical order. The first letter of the types is sufficient for defining a type.
`y.sizes`	defines the relative sizes of the small plots. The values are divided by their sum to get percentages.
`design`	if `design` is `chessboard` the graphics device is fragmented into rows and cols. Otherwise the images of a variable build vertical stripes.
`main`	defines a title for the graphics.
`mycols`	allows to define some colors for the showing the regions separated by the quartils.

Details

plotsummary can be use for a quick and dirty inspection of a data matrix or a list of variables. Without further specification some representation of each of the variables is built and stacked into a plot. The sizes of the types of representation can be set as well as the layout design of the graphics device. It is helpful to trim the data before processing because outliers will often hide the interesting characteristics.

Author(s)

Peter Wolf, pwolf@wiwi.uni-bielefeld.de

See Also

pairs, summary, str

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--\tor do  help(data=index)  for the standard data sets.
plotsummary(cars)
plotsummary(cars, types=c("ecdf", "density", "boxplot"),
            y.sizes = c(1,1,1), design ="stripes")
plotsummary(c(list(rivers=rivers, co2=co2), cars), y.sizes=c(10,3,3,1), mycols=3)
plotsummary(cars, design="chessboard")
# find all matrices in your R
ds.of.R <- function(type="vector"){
  dat <- ls(pos=grep("datasets",search()))
  dat.type <- unlist(lapply(dat,function(x) {
     num <- mode(x<-eval(parse(text=x)))
     num <- ifelse(is.array(x),"array",num)
     num <- ifelse(is.list(x),"list",num)
     num <- ifelse(is.matrix(x),"matrix",num)
     num <- ifelse(is.data.frame(x),"matrix",num)
     num <- ifelse(num=="numeric","vector",num)
     num }))
  return(dat[dat.type==type])
}
namelist <- ds.of.R("matrix")
# inspect the matrices one after the other
for(i in seq(along=namelist)){
  print(i); print(namelist[i])
  xy <- get(namelist[i])
  # plotsummary(xy,y.sizes=4:1,trim=.05,main=namelist[i])
  # Sys.sleep(1)
}
```

puticon *Add Icon(s) to a Plot*

Description

puticon() draws icons at the coordinates given by x and y.

Usage

```
puticon(x = 0, y = 0, icon = "", grey.levels = 0.5, icon.cex = 10,
        color = "red", ..., adj = c(0.5, 0.5), xpd = NA)
```

Arguments

x, y	numeric vectors of coordinates where to plot icon(s). If x is missing some information about internal icon generators are printed or plotted.
icon	icon to use. There are several ways to define an icon: If icon is a file name with one of the extensions c(".jpg",".JPG",".pnm",".PNM",".png",".PNG") puticon() tries to use the graphics file to plot the icon(s). To read jpeg and png files the functions jpeg and png of the packages jpeg and png are called. Note: If an image file is defined by an internet link it is temporarily downloaded using tempfile() and download.file(). If icon is a number a central symbol is plotted by invoking points. Remark: Usually the width of central symbols are a little bit smaller than par()$cin[1]*0.75. Therefore, it may be necessary to increase icon.cex to get an icon of a suitable size. If icon is a raster graphics object this object is used as icon. If icon is a string and if it is the name of an in internal icon generator (function) this generator is used to generate the icon(s). In the moment the following generators are implemented: BI,TL,cross.simple,cross,circle.simple,circle,car.simple,car,nabla, walkman,smiley.blueeye,smiley.normal,smiley,smiley.sad,mazz.man,bike, bike2,heart,bend.sign,fir.tree,comet,coor.system. If icon is a function it is used as an icon generating function.
grey.levels	An image from a file is transformed to a black-and-white picture. grey.levels defines the grey levels of the black-and-white picture that are replaced by color. If grey.levels is a single decimal number and is in (0,1) the pixels with a level greater than grey.levels are recolored by color. If grey.levels consists of two decimal numbers lower 1 pixels with grey levels lying in the interval are recolored by color. If grey.levels is an integer a vector of levels is created. If grey.levels is a vector and all(grey.levels < 1) puticon tries to different intensities of colors of color for recoloring pixels.
icon.cex	size(s) of icon(s) in mm. If icon.cex < 1 it is interpreted as ratio (width of icon) / (width of plotting area (par()$pin[1])) and is transformed to mm.
color	color(s) to be used for the pictogram(s). color can be a color code or name, for details see section Color Specification of the help of par.

...	Further parameters to be passed to the icon generating function.
adj	adj one or two values usually lying in [0, 1] and which specify the x (and y) adjustment of the icon(s).
xpd	controls clipping. See help of par for further explainations.

Details

For details concerning icon generating function see the help of iconplot(). If puticon() is called without argument x and icon is an empty string a list of internal generators will be returned. If x is missing and icon is the name of an internal generator a standardized version of the icon is plotted and the arguments of the generator (function) are printed.

Value

Usually Null is returned. However, if no coordinates are set and the name of an internal generator is assigned to icon puticon returns the definition of the generator function.

Note

Remark: the version of puticon of this package is an experimental version. Therefore, in the future some of the features may be changed and it is not sure that the function works as described on all types of graphics devices. In case of errors feel free to write a mail. Additional information and examples are found on the web page
http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/wolf_aplpack.

Author(s)

Peter Wolf

References

under construction

See Also

points, rasterImage, iconplot

Examples

```
# representation of data set "trees" by plotting characters
x <- trees[,1]; y <- trees[,2]; colors <- rainbow(100)[floor(trees[,3])]
plot(x, y, type = "n")
puticon(x, y, icon = 1, color = colors, icon.cex = 15, lwd = 6)
for(i in seq(along = x)){
  puticon(x[i], y[i], icon = i - 25 * ( i > 25),
          color = "red", icon.cex = 7, lwd = 4)
}
# representation of data set "trees" by fir.tree icons
x <- trees[,1]; y <- trees[,2]; colors <- rainbow(100)[floor(trees[,3])]
```

```
plot(x, y, type = "n")
puticon(x, y, icon = "fir.tree", icon.cex = 10, color = colors,
        height = y / 50, width = x / 10)
# standardized design of icon generator "fir.tree" and its definition
puticon( icon = "fir.tree" )
# list of implemented icon generators / generator functions
puticon()
# demo of internal icon generator functions
h <- puticon(); n <- length(h); y <- 1 + ((1:n)-1)
plot(1:n, xlim = c(0, n + 4), ylim = c(0, n / 2 + 4), type = "n")
for(i in 1:n)
  puticon(i, y[i] + (0:1), h[i], icon.cex = 3 + (1:2) , color = 3:4)
text(1:n - 0.3, y - 1, h, adj = c(0, 0.5))
# some smileys and Bielefeld logos of different colors and different sizes
plot(1:100, type = "n")
n <- 15; set.seed(26); x <- seq(10, 90, length = n); y <- runif(n, 10, 90)
sizes <- 5 + (1:n) / 4; my.color = rainbow(n); h <- 2 + (1:n)^0.5
puticon(x, y, icon = "BI", icon.cex = sizes, color = my.color)
puticon(x + h, y + h, icon = "smiley", color = my.color, icon.cex = sizes)

# icons with some letters
n <- 150; plot(1:n, 1:n, type = "n", xlab ="", ylab = "")
x <- runif(n, 1, n); y <- runif(n, 1, n); colors <- sample(rainbow(n))
for(i in 1:n)
  puticon(x[i], y[i], icon = "TL", icon.cex = 20,
          shiftY = runif(1, -10, 10), color = colors[i],
          L = paste(sample(letters, sample(1:5, size = 1)), collapse = ""))
# a modern painting
plot(1:20, xlim = c(-7,22), ylim = c(-7,22), type = "n", axes = FALSE,
   xlab ="", ylab = "")
rect(-7, -7, 22, 22, col = "gray")
n <- 100; set.seed(13); colors <- sample(rainbow(n)); CEX <- sort(runif(n, 2, 21))
for(i in 1:n){
  icon <- c("cross.simple", "cross", "circle.simple", "circle")[[sample(1:4, 1)]]
  puticon(runif(1, -5,20),  runif(1, -5, 20), icon,
          icon.cex = CEX[i], z = runif(1, 0.20, 0.45),
          whole = runif(1, 0.1, 0.6), color = colors[i])
}

# Traveller plot proposed by M. Mazziotta and A. Pareto.
#   M. Mazziotta, A. Pareto (2016):
#   Non-compensatory Aggregation of Social Indicaters: An Icon Representation.
#   url{http://link.springer.com/chapter/10.1007/978-3-319-05552-7_33}
Mazzi.Pareto <-
 structure(list(Region = c("Piemonte", "Valle d'Aosta", "Lombardia",
 "Trentino-Alto Adige", "Veneto", "Friuli-Venezia Giulia", "Liguria",
 "Emilia-Romagna", "Toscana", "Umbria", "Marche", "Lazio", "Abruzzo",
 "Molise", "Campania", "Puglia", "Basilicata", "Calabria", "Sicilia",
 "Sardegna"), Mean = c(98.74, 104.07, 101.38, 106.1, 104.38, 105.55,
 102.76, 103.62, 101.84, 103.52, 102.05, 97.88, 102.9, 91.43,
 94.12, 96.78, 93.55, 92.59, 96.29, 100.45), Penalty = c(0.43,
 4.23, 0.64, 0.63, 0.77, 0.34, 0.29, 0.46, 0.27, 0.22, 0.15, 0.82,
 1.3, 1.02, 0.37, 0.21, 2.37, 0.51, 0.31, 0.76), MPI = c(98.3,
```

```
    99.84, 100.74, 105.47, 103.61, 105.21, 102.47, 103.16, 101.57,
    103.3, 101.9, 97.06, 101.6, 90.42, 93.75, 96.58, 91.18, 92.08,
    95.98, 99.69)), .Names = c("Region", "Mean", "Penalty", "MPI"
    ), row.names = c(NA, -20L), class = "data.frame")
  plot(0, xlim = c(0.5, 4.5), ylim = c(0.83, 4.9),
       axes = FALSE,xlab = "", ylab = "" )
  x <- rep(1:4,5) - 1; y <- rep(5:1, each = 4)
  puticon( x, y, "mazz.man", icon.cex = 15, color = 1,
           Mean = Mazzi.Pareto$Mean, Penalty = Mazzi.Pareto$Penalty,
           Region = Mazzi.Pareto$Region, x.text = 70, y.text = -10 )
  # some cars
  plot(1:1000, type = "n", axes = FALSE, xlab = "", ylab = "")
  n <- 200; set.seed(13); x <- runif(n, -100, 1100); y <- runif(n, -100, 1100)
  colors <- sample(rainbow(n))
  for( i in 1:n ){
    puticon(x[i], y[i], icon = "car", icon.cex = runif(1, 10, 20),
             width = runif(1, 0, 1), height = runif(1, 0, 1), color = colors[i])
  }
  # fuzzy scatter plots as icons
  plot(-30:120, -30:120, type = "n", axes = FALSE, xlab = "", ylab = "")
  set.seed(13)
  puticon(50, 50, icon = "coor.system", icon.cex = .8, color = "blue",
          xxx = list(rnorm(20, 50, 15)), yyy = list(rnorm(100, 50, 15)*1000),
          axes = TRUE)
  puticon(x = c(20, 100, 95), y = c(100, 110, -45), icon = "coor.system",
          icon.cex = c(20, 30), color = c("green", "red", "magenta"),
          xxx = list(c(30, 50, 70), c(10, 20), c(80, 90, 10)),
          yyy = list(c(20, 60, 30), c(10, 20), c(10, 80, 90)), pcex = 10)
  # Marilyn Monroe or R icons via internet
  plot(1:20, type = "n",  axes = FALSE, xlab = "", ylab = "")
  f1 <- "http://www.radiopaula.cl/wp-content/uploads/2014/03/marilyn-monroe-3-andrew-fare.jp
  ## Not run: puticon(15, 17, icon = f1, icon.cex = 40, color = NA)
  ## Not run: puticon( c(6, 9, 12, 15), c(15, 13, 11, 9), icon = f1, icon.cex = 20,
     color = rainbow(4), grey.levels = 20)
## End(Not run)
  ## Not run: puticon( 4,  8, icon = f1, icon.cex = 40, color = "green", grey.levels = c(0.5
  ## Not run: puticon(10,  4, icon = f1, icon.cex = 40, color = "blue",  grey.levels = c(0.0
  plot(1:20, type = "n",  axes = FALSE, xlab = "", ylab = "")
  f1 <- "https://developer.r-project.org/Logo/Rlogo-4.png"
  ## Not run: puticon(15, 17, icon = f1, icon.cex = 40, color = NA)
  ## Not run: puticon( c(6, 9, 12, 15), c(15, 13, 11, 9), icon = f1, icon.cex = 20,
     color = rainbow(4), grey.levels = 20)
## End(Not run)
  ## Not run: puticon( 4,  8, icon = f1, icon.cex = 40, color = "green", grey.levels = c(0.5
  ## Not run: puticon(10,  4, icon = f1, icon.cex = 40, color = "blue",  grey.levels = c(0.0
  # simple raster graphics
  plot(1:20, pch = 1:20)
  puticon(1:20, sample(1:20), icon = 15, icon.cex = 20)
  image1 <- as.raster(  matrix( c(1,1,1,1,0,1,1,1,1), ncol = 3, nrow = 3))
  image2 <- as.raster(  matrix( c(0,1,0,1,0,1,0,1,0), ncol = 3, nrow = 3))
  image3 <- as.raster(  matrix( c(0,0,0,0,1,0,0,0,0), ncol = 3, nrow = 3))
  puticon( 7, 14,            icon = image1, icon.cex = .5, col = "orange")
  puticon( c(5, 10), c(5,5), icon = image2, icon.cex = c(.1, .2), color = 3:4)
```

```
puticon( 17, 10,           icon = image3, icon.cex = .30, col = "yellow")
# demo "my.house" of writing a generator function to generate icons
my.house <- function(col1 = 2, col2 = 3, col3 = 4){
  # initialize result object
  result <- NULL
  # compose object of type "polygon" consisting of
  # x-, y-values and colors
  x <- c(0, 1, 1, 0, 0, 1, 0.5,  0, 1) * 55 + 20
  y <- c(0, 0, 1, 1, 0, 1, 1.65, 1, 0) * 55 + 5
  res <- data.frame( x, y, color = col2)
  # add class "polygon" to the object and store it in "result"
  class(res) <- c(class(res), "polygon"); result <- c(result, list(res))
  # compose another object of type "polygon"
  res <- data.frame( x[c(1, 3, 4, 2)], y[c(1, 3, 4, 2)], color = col3)
  # add class "polygon" to the object and store it in "result"
  class(res) <- c(class(res), "polygon"); result <- c(result, list(res))
  n <- length(x)
  # compose object of type "segments" consisting of
  # x1-, y1-, x2-, y2-values, line widths and colors
  res <- data.frame( x[-n], y[-n], x[-1], y[-1], lwd.mm = 5, color = col1)
  # add class "segments" to the object and store it in "result"
  class(res) <- c(class(res), "segments"); result <- c(result, list(res))
  # output result object
  result
}
plot(1:100, type = "n")
n <- 50; x <- runif(n, 10, 90); y <- runif(n, 10, 90)
colors <- rainbow(n); sizes <- 5 + sample(1:n) / 2
puticon(x, y, icon = my.house, icon.cex = sizes,
        col1 = sample(colors), col2 = sample(colors), col3 = sample(colors) )
# demo "my.star" of writing a generator function to generate icons
my.star <- function(xx = 1:5, max.xx, star.txt = "..."){
  if(missing(max.xx)) max.xx <- max(xx)
  n <- length(xx); xx <- 50 * xx / max.xx
  colors <- rainbow(n); result <- NULL
  # compose object of type "segments" consisting of
  # x1-, y1-, x2-, y2-values, line widths and colors
  if( n > 1 ){
    x <- sin(2 * pi * (1:n) / n) * xx + 50
    y <- cos(2 * pi * (1:n) / n) * xx + 50
    res <- data.frame( 50, 50, x, y, lwd.mm = 2, color = colors)
  } else {
    res <- data.frame( 50, 50, x, y, width = 30, color = colors)
  }
  # add class "segments" to the object and store it in "result"
  class(res) <- c(class(res), "segments"); result <- c(result, list(res))
  # compose object of type "text" consisting of
  # x-, y-values, text, sizes of the text and colors
  res <- data.frame( 85, 20, txt = star.txt, t.cex.mm = 20, color = "blue")
  # add class "text" to the object and store it in "result"
  class(res) <- c(class(res), "text"); result <- c(result, list(res))
  # output result object
  result
```

```
}
plot(1:100, type = "n")
for(i in 1:10){
  puticon( runif(1, 0, 100), runif(1, 0, 100), icon = my.star, icon.cex = 20,
           xx = list(runif(14, 2, 10)), max.xx = 10, star.txt = letters[i])
}
```

`skyline.hist` *computes a skyline plot which is special histogram.*

Description

The function `skyline.hist` draws several histograms in one plot. The resulting image may look like a skyline.

Usage

```
skyline.hist(x, n.class, n.hist = 1, main, ylab="density",
             night = FALSE, col.bars = NA, col.border = 4, lwd.border = 2.5,
             n.shading = 6, lwd.shading = 2, col.shading = NA, lty.shading = 3,
             pcol.data = "green", cex.data = 0.3, pch.data = 16, col.data = 1,
             lwd.data = .2, permutation = FALSE,
             xlab, xlim, ylim, new.plot=TRUE, bty="n", ...)
```

Arguments

`x`	one dimensional data set.
`n.class`	number of classes that should be used to find the width of the bars of the histogram(s).
`n.hist`	number of histograms that should be plotted.
`main`	used for call of `title`.
`ylab`	text for y axis.
`night`	If `TRUE` the background will be colored blue. If `FALSE` there will be no colored background. Otherwise `night` is used as background color.
`col.bars`	defines the color of the bars. If `is.na(col.bars)` and `night==TRUE` the bars will be colored gray.
`col.border`	color of the borders of the bars.
`lwd.border`	line width of the borders of the bars.
`n.shading`	number of vertical lines for filling the bars of the histograms.
`lwd.shading`	line width of the vertical lines for shading the bars.
`col.shading`	color for the vertical lines for shading. If `NA` heat colors are used.
`lty.shading`	line type for the vertical lines for shading.
`pcol.data`	color of data points.

cex.data	character size of plotting character.
pch.data	plotting character of data points.
lwd.data	line width for segments between data points.
col.data	color for segments between data points.
permutation	if not FALSE a permutation of the data set is erformed.
xlab	text for y axis.
xlim	range of x.
ylim	range of y.
new.plot	logical. If TRUE a new plot is constructed.
bty	box type, used by plot.
...	further graphical parameters passed to plot.

Details

skyline.hist computes several histograms and plots them one upon the other. The histograms differ in the positions of the first cells, but all cells have the same width. The parameters n.class and n.hist have the greatest effect on the design of the result. col.border allows to color the border of the rectangular boxes of the histogram bars. col.bars defines the fill color of the bars. n.shading defines the number of vertical lines of type lty.shading and width lwd.shading that are drawn within the boxes. Another feature of skyline.hist is to represent the data points. The data points of a cell are plotted according their x-values and their ranks (within the points of the cell). The resulting points are connected by line segments and you will see a time series running from bottom to top in each cell. The points and lines can be specified by pcol.data, cex.data, pch.data, lwd.data, col.data. To get rid of the original order of the data you can permutated them (permutation=1). The "skyline" of the plot may be similar to the skyline of a town and the vertical lines may look like small windows of buildings. In Young et. al. you find "shaded histograms". These histograms have triggered the idea of skyline.hist and the representation of a one dimensional data set by laying histograms on top of otheroverlied histograms.

Value

The result of a call of hist is returned.

Author(s)

Peter Wolf, pwolf@wiwi.uni-bielefeld.de

References

F.W. Young, R.M. Valero-Mora, M. Friendly (2006): Visual Statistics. Wiley, p207–208.

See Also

hist, density

Examples

```
  # dev.off()
print(par())
  par(mfrow=c(1,1))
  for(n.c in c(2,4,8)){  # some values for n.class
    for(n.h in c(2,4,3)){# some values for number of n.hist
        n.s <- 9         # value for number of vertical lines
        skyline.hist(co2, n.shading = n.s, n.hist = n.h ,n.class = n.c,
                     night = n.h==3, col.border = n.h!=4)
    }
  }
  par(mfrow = c(1,1))
  skyline.hist(x=rivers, n.class=4, n.hist=2, n.shading=0, main="rivers",
              cex.data=.5, lwd.data = .2, col.data = "green", pcol.data = "red",
              col.border=NA, night=FALSE, ylab="density")
  skyline.hist(x=rivers, n.class=4, n.hist=5, n.shading=0, main="rivers",
              cex.data=.5, lwd.data = 1, col.data = "green", pcol.data = "red",
              col.border=NA, night="blue" , ylab="density", col.bars =NA)
  skyline.hist(x=rivers, n.class=10, n.hist=2, n.shading=0, main="rivers",
              cex.data=.5, lwd.data = 1, col.data = "green", pcol.data = "red",
              col.border=NA, night=FALSE , ylab="density", col.bars = "lightblue")
  skyline.hist(x=rivers, n.class=10, n.hist=1, n.shading=0, main="rivers",
              cex.data=1, lwd.data = 0, col.data = "green", pcol.data = "red",
              col.border=NA, night=FALSE , ylab="density", col.bars = "lightblue" )
  skyline.hist(x=rivers, n.class=6, n.hist=1, n.shading=0, main="rivers",
              cex.data=0.1, lwd.data = 2, col.data = "red", pcol.data = "green",
              night="orange" , ylab="density", col.bars = "white", col.border=1 )
  skyline.hist(x=rivers, n.class=6, n.hist=1, n.shading=0, main="rivers",
              cex.data=0.1, lwd.data = 2, col.data = "red", pcol.data = "green",
              col.border=NA, night=FALSE , ylab="density", col.bars = "lightblue")
  skyline.hist(x=rivers, n.class=6, n.hist=1, n.shading=5, col.shading = "blue",
              main="rivers",
              cex.data=0.1, lwd.data = 1, col.data = "black", pcol.data = "green",
              col.border=NA, night=FALSE , ylab="density", col.bars = "green")
  skyline.hist(x=rivers, n.class=6, n.hist=3, n.shading=5, col.shading = "blue",
              main="rivers", col.bars = "green",
              cex.data=0.1, lwd.data = 1, col.data = "black", pcol.data = "green",
              col.border="white", night="magenta" , ylab="density")
  skyline.hist(x=rivers, n.class=6, n.hist=4, n.shading=5, col.shading = "blue",
              main="rivers",
              cex.data=0.8, lwd.data = 1, col.data = "blue", pcol.data = "red",
              col.border=NA, night=FALSE , ylab="density", col.bars = "green")
```

`slider` *slider / button control widgets*

Description

`slider` and `gslider` construct a Tcl/Tk-widget with sliders and buttons to demonstrate the effects of variation of parameters on calculations and plots.

Usage

```
slider(sl.functions, sl.names, sl.mins, sl.maxs, sl.deltas, sl.defaults, but.functi
  but.names, no, set.no.value, obj.name, obj.value, reset.function, title, prompt=F
  sliders.frame.vertical=TRUE)

gslider(sl.functions, sl.names, sl.mins, sl.maxs, sl.deltas, sl.defaults, but.funct
  but.names, no, set.no.value, obj.name, obj.value, reset.function, title, prompt=F
  sliders.frame.vertical=TRUE, hscale=1, vscale=1,
  pos.of.panel = c("bottom","top","left","right")[1])
```

Arguments

`sl.functions`	set of functions or function connected to the slider(s)
`sl.names`	labels of the sliders
`sl.mins`	minimum values of the sliders' ranges
`sl.maxs`	maximum values of the sliders' ranges
`sl.deltas`	change of step per click
`sl.defaults`	default values for the sliders
`but.functions`	function or list of functions that are assigned to the button(s)
`but.names`	labels of the buttons
`no`	`slider(no=i)` requests slider `i`
`set.no.value`	`slider(set.no.value=c(i,val))` sets slider `i` to value `val`
`obj.name`	`slider(obj.name=name)` requests the value of variable `name` from environment `slider.env`
`obj.value`	`slider(obj.name=name,obj.value=value)` assigns `value` to variable `name` in environment `slider.env`
`reset.function`	function that induce a `reset.button` and contains the commands of it.
`title`	title of the control window
`prompt`	if TRUE slider functions are called by moving a slider, if FALSE slider functions are called after releasing the mouse button
`sliders.frame.vertical`	if TRUE the sliders are stacked one above the other; otherwise they are positioned side by side
`hscale`	horizontal scale factor for image size; compare tkrplot in package tkrplot
`vscale`	vertical scale factor for image size; compare tkrplot in package tkrplot
`pos.of.panel`	position of the panel field for sliders and buttons. Value of `pos.of.panel`: bottom, top, left or right.

Details

slider constructs a separated panel for controlling the parameters whereas gslider integrates a graphical device and buttons and sliders within one window.

The following actions can be done: a) definition of (multiple) sliders and buttons, b) request or specification of slider values, and c) request or specification of variables in the environment slider.env. The management takes place in the environment slider.env. If slider.env is not found it is generated.

Definition ... of sliders: First of all you have to define sliders, buttons and the attributes of them. Sliders are established by six arguments: sl.functions,sl.names,sl.minima,sl.maxima,sl.deltas, and sl.defaults. The first argument, sl.functions, is either a list of functions or a single function that contains the commands for the sliders. If there are three sliders and slider 2 is moved with the mouse the function stored in sl.functions[[2]] (or in case of one function for all sliders the function sl.functions) is called.

DEFINITION ... of buttons: Buttons are defined by a vector of labels but.names and a list of functions: but.functions. If button i is pressed the function stored in but.functions[[i]] is called.

REQUESTING ... a slider: slider(no=1) returns the actual value of slider 1, slider(no=2) returns the value of slider 2, etc. You are allowed to include expressions of the type slider(no=i) in functions describing the effect of sliders or buttons.

SETTING ... a slider: slider(set.no.value=c(2,333)) sets slider 2 to value 333. slider(set.no.value=c(i,value)) can be included in the functions defining the effects of moving sliders or pushing buttons.

VARIABLES ... of the environment slider.env: Sometimes information has to be trransferred back and forth between functions defining the effects of sliders and buttons. Imagine for example two sliders: one to control p and another one to control q, but they should satisfy: p+q=1. Consequently, you have to correct the value of the first slider after the second one was moved. To prevent the creation of global variables store them in the environment slider.env. Use slider(obj.name="p.save",obj.value=1-slider(no=2)) to assign value 1-slider(no=2) to the variable p.save . slider(obj.name=p.save) returns the value of variable p.save.

Dependencies The function gslider depends on package tkrplot.

Value

Using slider in definition mode slider returns the value of new created the top level widget. slider(no=i) returns the actual value of slider i. slider(obj.name=name) returns the value of variable name in environment slider.env. gslider return in definition mode the result of tkrplot which was called to construct the widget.

Author(s)

Hans Peter Wolf

Examples

```
# example 1, sliders only
if(interactive()){
```

```
## This example cannot be run by examples() but should work in an interactive R session
plot.sample.norm<-function(){
 refresh.code<-function(...){
   mu<-slider(no=1); sd<-slider(no=2); n<-slider(no=3)
   x<-rnorm(n,mu,sd)
   plot(x)
 }
 slider(refresh.code,sl.names=c("value of mu","value of sd","n number of observations"),
       sl.mins=c(-10,.01,5),sl.maxs=c(+10,50,100),sl.deltas=c(.01,.01,1),sl.defaults=c(0,1,2
}
plot.sample.norm()
}

# example 2, sliders and buttons
if(interactive()){
## This example cannot be run by examples() but should work in an interactive R session
plot.sample.norm.2<-function(){
 refresh.code<-function(...){
   mu<-slider(no=1); sd<-slider(no=2); n<-slider(no=3)
   type=  slider(obj.name="type")
   x<-rnorm(n,mu,sd)
   plot(seq(x),x,ylim=c(-20,20),type=type)
 }
 slider(obj.name="type",obj.value="l")
 slider(refresh.code,sl.names=c("value of mu","value of sd","n number of observations"),
       sl.mins=c(-10,.01,5),sl.maxs=c(10,10,100),sl.deltas=c(.01,.01,1),sl.defaults=c(0,1,20
       but.functions=list(
              function(...){slider(obj.name="type",obj.value="l");refresh.code()},
              function(...){slider(obj.name="type",obj.value="p");refresh.code()},
              function(...){slider(obj.name="type",obj.value="b");refresh.code()}
       ),
       but.names=c("lines","points","both"))
}
plot.sample.norm.2()
}

# example 2a, sliders and buttons and graphics in one window
if(interactive()){
## This example cannot be run by examples() but should work in an interactive R session
plot.sample.norm.2<-function(){
 refresh.code<-function(...){
   mu<-slider(no=1); sd<-slider(no=2); n<-slider(no=3)
   type=  slider(obj.name="type")
   x<-rnorm(n,mu,sd)
   plot(seq(x),x,ylim=c(-20,20),type=type)
 }
 slider(obj.name="type",obj.value="l")
 gslider(refresh.code,sl.names=c("value of mu","value of sd","n number of observations"),
       sl.mins=c(-10,.01,5),sl.maxs=c(10,10,100),sl.deltas=c(.01,.01,1),sl.defaults=c(0,1,20
       but.functions=list(
              function(...){slider(obj.name="type",obj.value="l");refresh.code()},
              function(...){slider(obj.name="type",obj.value="p");refresh.code()},
              function(...){slider(obj.name="type",obj.value="b");refresh.code()}
```

```
       ),
       but.names=c("lines","points","both"))
}
plot.sample.norm.2()
}

# example 3, dependent sliders
if(interactive()){
## This example cannot be run by examples() but should work in an interactive R session
print.of.p.and.q<-function(){
 refresh.code<-function(...){
   p.old<-slider(obj.name="p.old")
   p<-slider(no=1); if(abs(p-p.old)>0.001) {slider(set.no.value=c(2,1-p))}
   q<-slider(no=2); if(abs(q-(1-p))>0.001) {slider(set.no.value=c(1,1-q))}
   slider(obj.name="p.old",obj.value=p)
   cat("p=",p,"q=",1-p,"\n")
 }
 slider(refresh.code,sl.names=c("value of p","value of q"),
       sl.mins=c(0,0),sl.maxs=c(1,1),sl.deltas=c(.01,.01),sl.defaults=c(.2,.8))
 slider(obj.name="p.old",obj.value=slider(no=1))
}
print.of.p.and.q()
}

# example 4, rotating a surface
if(interactive()){
## This example cannot be run by examples() but should work in an interactive R session
R.veil.in.the.wind<-function(){
  # Mark Hempelmann / Peter Wolf
  par(bg="blue4", col="white", col.main="white",
      col.sub="white", font.sub=2, fg="white") # set colors and fonts
  refresh.code<-function(...){
    samp        <- function(N,D) N*(1/4+D)/(1/4+D*N)
    z<-outer(seq(1, 800, by=10), seq(.0025, 0.2, .0025)^2/1.96^2, samp) # create 3d matrix
    h<-100
    z[10:70,20:25]<-z[10:70,20:25]+h; z[65:70,26:45]<-z[65:70,26:45]+h
    z[64:45,43:48]<-z[64:45,43:48]+h; z[44:39,26:45]<-z[44:39,26:45]+h
    x<-26:59; y<-11:38; zz<-outer(x,y,"+"); zz<-zz*(65<zz)*(zz<73)
    cz<-10+col(zz)[zz>0];rz<-25+row(zz)[zz>0]; z[cbind(cz,rz)]<-z[cbind(cz,rz)]+h
    theta<-slider(no=1); phi<-slider(no=2)
    persp(x=seq(1,800,by=10),y=seq(.0025,0.2,.0025),z=z,theta=theta,phi=phi,
          scale=T, shade=.9, box=F, ltheta = 45,
          lphi = 45, col="aquamarine", border="NA",ticktype="detailed")
  }
  slider(refresh.code, c("theta", "phi"), c(0, 0),c(360, 360),c(.2, .2),c(85, 270)  )
}
R.veil.in.the.wind()
}
```

`slider.bootstrap.lm.plot` *interactive bootstapping for lm*

Description

`slider.bootstrap.lm.plot` computes a scatterplot and adds regression curves of samples of the data points. The number of samples and the degree of the model are controlled by sliders.

Usage

```
slider.bootstrap.lm.plot(x, y, ...)
```

Arguments

`x`	two column matrix or vector of x values if y is used
`y`	y values if x is not a matrix
`...`	additional graphics parameters

Details

`slider.bootstrap.lm.plot` draws a scatterplot of the data points and fits a linear model to the data set. Regression curves of samples of the data are then added to the plot. Within a Tcl/Tk control widget the degree of the model, the repetitions and the start of the random seed are set. After modification of a parameter the plot is updated.

Value

a message about the usage

Author(s)

Hans Peter Wolf

References

~~

See Also

`plot`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  daten<-iris[,2:3]
  slider.bootstrap.lm.plot(daten)

## End(Not run)
```

`slider.brush` *interactive brushing functions*

Description

These functions compute a pairs plot or a simple xy-plot and open a slider control widget for brushing.

`slider.brush.pairs` computes a pairs plot; the user defines an interval for one of the variables and in effect all data points in this interval will be recolored.

`slider.brush.plot.xy` computes an xy-plot; the user defines a interval for a third variable `z` and all points `(x,y)` will be recolored red if the `z` value is in the interval.

Usage

```
slider.brush.pairs(x, ...)
slider.brush.plot.xy(x, y, z, ...)
```

Arguments

`...`	new settings for global graphics parameters
`x`	matrix or data frame or vector
`y`	vector of y values if `x` is not a matrix
`z`	vector of z values if `x` is not a matrix

Details

`slider.brush.pairs` draws a pairs plot of the data set `x`. The first slider defines the lower limit of the interval and the second its width. By the third slider a variable is selected. All data points for which the selected variable is in the interval are recolored red.

`slider.brush.plot.xy` draws an xy-plot of the data set `x`. The first slider defines the lower limit of the interval of z values and the second one its width. All data points for which the variable z is in the interval are recolored red.

Value

a message about the usage

Author(s)

Hans Peter Wolf

References

W. S. Cleveland, R. A. Becker, and G. Weil. The Use of Brushing and Rotation for Data Analysis. InW. S. Cleveland and M. E. McGill, editors, Dynamic Graphics for Statistics. Wadsworth and Brooks/Cole, Pacific Grove, CA, 1988.

See Also

pairs, plot

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.brush.pairs(iris)

## End(Not run)
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.brush.plot.xy(iris[,1:3])

## End(Not run)
```

---

slider.hist *interactive histogram and density traces*

Description

The functions slider.hist and slider.density compute histograms and density traces whereas some parameter are controlled by sliders.

slider.hist computes a histogram; the number of classes is defined by a slider.

slider.density computes a density trace; width and type of the kernel are defined by sliders.

Usage

```
slider.hist(x, panel, ...)
slider.density(x, panel, ...)
```

Arguments

x	data set to be used for plotting
panel	function constructing additional graphical elements to the plot
...	additional (graphics) parameters which are passed to the invoked high level plotting function

Details

slider.hist draws a histogram of the data set x by calling hist and opens a Tcl/Tk widget with one slider. The slider defines the number of classes of the histogram. Changing the slider results in redrawing of the plot. For further details see the help page of hist. rug is used as the default panel function.

slider.density draws a density trace of the data set x by plot(density(...)) and opens a Tcl/Tk-widget with two sliders. The first slider defines the width of the density trace and the

second one the kernel function: `"1-gaussian","2-epanechnikov","3-rectangular","4-triangular","`
Changing one of the sliders results in a redrawing of the plot. For further details see the help page of `density`. `rug` is used as the default panel function.

Value

a message about the usage

Author(s)

Hans Peter Wolf

References

~~

See Also

`hist`, `slider`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.hist(log(islands))

## End(Not run)
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
slider.density(rivers,xlab="rivers",col="red")

## End(Not run)
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
slider.density(log(rivers),xlab="rivers",col="red",
  panel=function(x){
    xx<-seq(min(x),max(x),length=100)
    yy<-dnorm(xx,mean(x),sd(x))
    lines(xx,yy)
    rug(x)
    print(summary(yy))
  }
)

## End(Not run)
```

`slider.lowess.plot` *interactive lowess smoothing*

Description

`slider.lowess.plot` computes an xy-plot of the data and adds LOWESS lines. The smoother span and the number of iterations are selected by sliders.

Usage

```
slider.lowess.plot(x, y, ...)
```

Arguments

`x`	data set to be used for plotting or vector of x values
`y`	vector of y values in case `x` is not a matrix
`...`	additional (graphics) parameter settings

Details

`slider.lowess.plot` computes a scatterplot of the data. Then a LOWESS smoother line is added to the plot. For more details about the lowess parameters `f` and `iter` take a look at the help page of `lowess`. The parameters are set by moving sliders of the control widget. The first slider defines the smoother span `f` and the second one the number of iterations.

Value

a message about the usage

Author(s)

Hans Peter Wolf

References

for references see help file of `lowess`

See Also

`lowess`, `slider`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.lowess.plot(cars)

## End(Not run)
```

`slider.smooth.plot.ts` *interactive Tukey smoothing*

Description

`slider.smooth.plot.ts` computes smooth curves of a time series plot by Tukey's smoothers. The kind of smoothing is controlled by a Tcl/Tk widget.

Usage

```
slider.smooth.plot.ts(x, ...)
```

Arguments

`x`	time series
`...`	additional graphical parameters

Details

`slider.smooth.plot.ts` draws the time series `x`. The user selects a filter of the set `c("3RS3R","3RSS","3RSR",` step by step and the resulting curve is added to the plot. The selection is performed by pressing a button of the control widget of `slider.smooth.plot.ts`. The button `reset` restarts the smoothing process.

Value

a message about the usage

Author(s)

Hans Peter Wolf

References

Tukey, J. W. (1977). Exploratory Data Analysis, Reading Massachusetts: Addison-Wesley.

See Also

`plot`, `smooth`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.smooth.plot.ts(rnorm(100))

## End(Not run)
```

`slider.split.plot.ts` *interactive splitting of time series*

Description

`slider.split.plot.ts` plots linear fitted lines or summary statistics in sections of a time series. The sections are controlled by sliders.

Usage

```
slider.split.plot.ts(x, type="l", ...)
```

Arguments

`x`	time series or vector
`type`	plotting type: `type` will be forwarded to function `plot`
`...`	additional graphics parameters

Details

`slider.split.plot.ts` draws a time series plot and let you define sections of the series by fixing a limit on the time scale as well as a window width. The whole range of the series is partitioned in pieces of the same length in a way that the fixed limit will be one of the section limits. Then linear models are fitted and plotted in the sections. Alternatively – by pressing the button `fivenum summary` – summary statistics are drawn instead of the model lines.

The first slider fixes the width of the sections and the second one the limit between two of them.

By clicking on button `linear model` or `fivenum summary` the user switches between drawing model curves and five number summary.

Value

a message about the usage

Author(s)

Hans Peter Wolf

See Also

`plot`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.split.plot.ts(as.vector(sunspots)[1:100])

## End(Not run)
```

`slider.stem.leaf` *construction of stem and leaf display interactively*

Description

'slider.stem.leaf' computes a stem and leaf display within a graphics device. The parameters are controlled by a control widget.

Usage

```
slider.stem.leaf(x, main = main)
```

Arguments

`x`	data set for plotting
`main`	main title of the plot

Details

The function 'slider.stem.leaf' allows the user to construct a stem and leaf display within a graphics device. The main parameters will be set by a Tcl/Tk control widget. The line rule is selected by pressing one of the buttons 'Dixon', 'Sturges', 'Velleman'. A slider controls the separation of the stem. Additionally the character size device could be set.

Value

a short message is returned

Note

The function is a function of the package aplpack

Author(s)

Peter Wolf, Nov 2009

See Also

`stem`

Examples

```
 ## Not run:
   slider.stem.leaf(islands)

## End(Not run)
```

`slider.zoom.plot.ts`
interactive zooming of time series

Description

This function shows one or two sections of a time series. The window(s) is (are) controlled by sliders.

Usage

```
slider.zoom.plot.ts(x, n.windows, ...)
```

Arguments

`x`	time series
`n.windows`	`if(n.windows>1` two sections are defined
`...`	additional graphical parameters

Details

`slider.zoom.plot.ts` plots the original time series and it lets you select one or two sections of the series by fixing the width(s) and the starting point(s) of the region(s). Then the section(s) of the series is (are) plotted separately one below the other.

The first slider defines the width of the section(s). The second (third) one sets the start of the first (second) section.

Value

a message about the usage

Author(s)

Hans Peter Wolf

See Also

`plot`

Examples

```
## Not run:
## This example cannot be run by examples() but should be work in an interactive R session
  slider.zoom.plot.ts(co2,2)

## End(Not run)
```

---

spin3R *spin3R*

Description

Simple spin function to rotate and to inspect a 3-dimensional cloud of points

Usage

```
spin3R(x, alpha = 1, delay = 0.015, na.rm=FALSE)
```

Arguments

x	(nx3)-matrix of points
alpha	angle between successive projections
delay	delay in seconds between two plots
na.rm	if TRUE 'NA' values are removed otherwise exchanged by mean

Details

spin3R computes two-dimensional projections of (nx3)-matrix x and plots them on the graphics device. The cloud of points is rotated step by step. The rotation is defined by a tcl/tk control widget. spin3R requires tcl/tk package of R.

Note

version 05/2008

Author(s)

Peter Wolf

References

Cleveland, W. S. / McGill, M. E. (1988): Dynamic Graphics for Statistics. Wadsworth & Brooks/Cole, Belmont, California.

See Also

spin of S-Plus

Examples

```
xyz<-matrix(rnorm(300),100,3)
# now start:     spin3R(xyz)
```

`stem.leaf` *stem and leaf display and back to back stem and leaf display*

Description

Creates a classical ("Tukey-style") stem and leaf display / back-to-back stem and leaf display.

Usage

```
stem.leaf(data, unit, m, Min, Max, rule.line = c("Dixon", "Velleman", "Sturges"),
    style = c("Tukey", "bare"), trim.outliers = TRUE, depths = TRUE,
    reverse.negative.leaves = TRUE, na.rm = FALSE, printresult = TRUE)
stem.leaf.backback(x,y, unit, m, Min, Max, rule.line = c("Dixon", "Velleman",
    "Sturges"), style = c("Tukey", "bare"), trim.outliers = TRUE,
    depths = TRUE, reverse.negative.leaves = TRUE, na.rm = FALSE,
    printresult=TRUE, show.no.depths = FALSE, add.more.blanks = 0,
    back.to.back = TRUE)
```

Arguments

`data`	a numeric vector of data
`x`	first dataset for `stem.leaf.backback`
`y`	first dataset for `stem.leaf.backback`
`unit`	leaf unit, as a power of 10 (e.g., `100`, `.01`); if `unit` is missing `unit` is choosen by `stem.leaf`.
`m`	number of parts (1, 2, or 5) into which each stem will be separated; if `m` is missing the number of parts/stem (`m`) is choosen by `stem.leaf`.
`Min`	smallest non-outlying value; omit for automatic choice.
`Max`	largest non-outlying value; omit for automatic choice.
`rule.line`	the rule to use for choosing the desired number of lines in the display; `"Dixon"` = 10*log10(n); `"Velleman"` = 2*sqrt(n); `"Sturges"` = 1 + log2(n); the default is `"Dixon"`.
`style`	`"Tukey"` (the default) for "Tukey-style" divided stems; `"bare"` for divided stems that simply repeat the stem digits.
`trim.outliers`	if `TRUE` (the default), outliers are placed on `LO` and `HI` stems.
`depths`	if `TRUE` (the default), print a column of "depths" to the left of the stems; the depth of the stem containing the median is the stem-count enclosed in parentheses.

`reverse.negative.leaves`
: if `TRUE` (the default), reverse direction the leaves on negative stems (so, e.g., the leaf 9 comes before the leaf 8, etc.).

`na.rm`
: if TRUE "NA" values are removed otherwise the number of NAs are counted.

`printresult`
: if TRUE output of the stem and leaf display by `cat`.

`show.no.depths`
: if TRUE no depths are printed.

`add.more.blanks`
: number of blanks that are added besides the leaves.

`back.to.back`
: if FALSE two parallel stem and leaf displays are constructed.

Details

Unlike the `stem` function in the `base` package, `stem.leaf` produces classic stem-and-leaf displays, as described in Tukey's *Exploratory Data Analysis*. The function `stem.leaf.backback` creates back-to-back stem and leaf displays.

Value

The computed stem and leaf display is printed out. Invisibly `stem.leaf` returns the stem and leaf display as a list containing the elements `info` (legend), `display` (stem and leaf display as character vecter), `lower` (very small values), `upper` (very large values), `depths` (vector of depths), `stem` (stem information as a vector), and `leaves` (vector of leaves).

Author(s)

Peter Wolf, the code has been slightly modified by John Fox <jfox@mcmaster.ca> with the original author's permission, help page written by John Fox, the help page has been slightly modified by Peter Wolf.

References

Tukey, J. *Exploratory Data Analysis.* Addison-Wesley, 1977.

See Also

`stem`

Examples

```
stem.leaf(co2)
stem.leaf.backback(co2[1:120],co2[121:240])
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE)
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE,
                   add.more.blanks = 3, show.no.depths = TRUE)
stem.leaf.backback(rivers[-(1:30)],rivers[1:30], back.to.back = FALSE, unit=10, m=5,
                   Min=200, Max=900, add.more.blanks = 20, show.no.depths = TRUE)
```