Package ‘loa’

February 20, 2015

Type Package
Title Various Options and Add-ins for Lattice
Version 0.2.22
Date 2015-01-20
Author Karl Ropkins
Maintainer Karl Ropkins <kropkins@its.leeds.ac.uk>
Description This package, Lattice Options and Add-ins (or loa), contains various plots and functions that make use of the lattice/trellis plotting framework. The plots (which include loaPlot, GoogleMap and trianglePlot) use panelPal, a function that extends lattice and hexbin package methods to automate plot subscripting and panel-to-panel and panel-to-key synchronization/management. See ?loa for further details.
Depends R (>= 2.13.0), lattice
Imports MASS, grid, png, RgoogleMaps, RColorBrewer
License GPL (>= 2)
LazyLoad yes
LazyData yes
Repository CRAN
Repository/R-Forge/Project loa
Repository/R-Forge/Revision 51
Repository/R-Forge/DateTimeStamp 2015-01-20 12:15:54
Date/Publication 2015-01-21 00:13:59
NeedsCompilation no

R topics documented:

loa-package ................................................................. 2
1.1.loaPlot ............................................................... 4
1.2.GoogleMap.and.geoplotting.tools ................................. 8
1.3.trianglePlot .......................................................... 13
2.1.specialist.panels .................................................... 19
The loa package contains various plots, options and add-ins for use with the lattice package.

Details

<table>
<thead>
<tr>
<th>Package:</th>
<th>loa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Package</td>
</tr>
<tr>
<td>Version:</td>
<td>0.2.22</td>
</tr>
<tr>
<td>Date:</td>
<td>2014-01-20</td>
</tr>
<tr>
<td>License:</td>
<td>GPL (&gt;= 2)</td>
</tr>
<tr>
<td>LazyLoad:</td>
<td>yes</td>
</tr>
</tbody>
</table>

lattice provides an elegant and highly powerful implementation of the Trellis plotting structure described by Cleveland and colleagues. In particular the combination of panel... functions, which can be layered within plots to generate novel visualisations, and simple-to-use conditioning make it a hugely effective tool when working with data.

The loa package contains a number of plot functions developed to make use of this framework. These are summarized in section 1 of this manual, and include:

1.1. loaPlot for various XYZ plots.

1.2. GoogleMap and associated geoplotting functions.

1.3. trianglePlot and associated functions.

Other panel... functions and example data are summarized in sections 2 and 3, respectively:

2.1. Specialist panels, e.g. panel.kernelDensity,
2.2. Specialist panels for polar plotting, e.g. `panel.polarPlot`.

3.1. Example data, e.g. `lat.lon.meuse`.

While such 'stand alone' plot functions are of obvious value, the code framework is of possibly wider interest because it provides a template for the rapid third-party development of novel visualization functions and a highly flexible 'test bed' for the comparison of different data handling strategies.

Therefore, the functions in this package have been written in a relatively dissaggregated fashion so code can be easily rearranged or modified by others to quickly develop alternative plots within the `lattice` framework. Firstly, plot functions in section 1 have where possible been supplied as main plot functions and plot component functions that handle data, axes, panels, etc. Secondly, the workhorse functions, those common functions used through-out the package to simplify many routine operations have been grouped together and summarized in section 4:

4.1. `panelPal`
4.2. plot structure handlers: `formulaHandler`, etc.
4.3. Plot lim(s) and scale(s) handlers: `limsHandler`, `localScalesHandler`, etc.
4.4. Plot conditioning handlers: `condsPanelHandler`, etc.
4.5. Common plot argument handlers: `cexHandler`, `colHandler`, `zHandler`, etc.
4.6. Key handlers: `keyHandler`, etc.
4.7. Other panel functions: `getArgs`, etc.
4.8. List handlers: `listHandler`, etc.

And, finally, functions used for working with data post-plotting, are summarized in section 5:

5.1. Interactive functions for working with plot outputs: `getXY`, etc.

This package is very much intended to be an evolutionary exercise. I use it on a routine basis to develop plots for use elsewhere and compare data visualization methods. However, that working pattern can generate some very 'developer-centric' code. So, I would be very pleased to hear from others - what they did and did not like about the package; what they would have liked to have been different; and, perhaps most interesting for me what they are using it to do.

**Author(s)**

Karl Ropkins <k.ropkins@its.leeds.ac.uk>

**References**

Functions in loa make extensive use of code developed by others. In particular, I gratefully acknowledge the huge contributions of:


**See Also**

`loaPlot, GoogleMap, trianglePlot`
1.1.loaPlot  

### Description

loaPlot is a standard XYZ plotting function, where X and Y are the axes of a conventional XY plot and Z is an element (or elements if supplied in the form Z1 + Z2 + Z3...) visualized at associated XY coordinates. By default, loaPlot links Z to plot point size and color to generate a bubbleplot style output, or using modified plot calls other plot types.

### Usage

```r
loaPlot(x, data = NULL, panel = panel.loaPlot,
       ..., local.scales = FALSE, reset.xylims = TRUE,
       load.lists = NULL, by.group = NULL, by.zcase = NULL,
       preprocess = TRUE)
```

#standard panels

```r
panel.loaPlot(..., loa.settings = FALSE)
pattern.loaPlot2(..., loa.settings = FALSE)
```

#grids

```r
panel.loaGrid(grid.x = NULL, grid.y = NULL,
              xlim = NULL, ylim = NULL, ...,
              grid = NULL, panel.scales = NULL)
```

### Arguments

- **x**  
  A formula with the general structure `z ~ x * y | cond` applied like in the `lattice` function `levelplot`. `x` and `y` are the horizontal and vertical axes, `z` is any additional information to be used in point, symbol, surface or glyph generation, and `cond` is any additional conditioning to be applied. `x` and `y` are required elements; `z` and `cond` are typically optional.
  (Note: this element of the plot is handled by `formulaHandler`).

- **data**  
  If supplied, the assumed source of the elements of formula `x`, typically a data.frame.

- **panel**  
  `panel` is the function to be used when generating the content of the individual panels within the `lattice` plot. By default, this is the `loa` panel function `panel.loaPlot`.

- **...**  
  Additional arguments are passed on to related functions.
  For `loaPlot` these are `colHandler`, `cexHandler` and the function set by `panel`. This mechanism provides access to most common plot parameters, e.g. `col`, `pch`, and `cex` for plot symbol color, type and size, respectively. By default, both data point color and size are z-scaled for `loaPlot`. If `z` is supplied, and
cex and col are not set by the user in the plot command, these plot properties are managed by `cexHandler` and `colHandler`, respectively. `cexHandler` and `colHandler` arguments can be also be passed directly as part of the `loaPlot` command to fine-tune these, e.g. cex.range to change the cex range that z values are scaled to and col.region to change the color range that is applied to z when coloring points. See associated Help documents for further information.

`local.scales`  
For `loaPlot` only, logical. If TRUE, this removes the standard lattice axis from the plot. It is intended to be used with `panel` functions which generate their own axes or have no axes.

`reset.xylims`  
For `loaPlot` only, logical or character vector. If a logical, if the panel outputs are preprocessed (using `panelPal`), should the x and y limits be reset? If a character vector, one or more terms controlling post-processing plot range management: `refit.xylims`, equivalent to `reset.xylims = TRUE`; and `max.xylims`, to reset both x and y ranges to maximum. 
(Note: If xlim or ylim are supplied in the plot call, these will typically override all `reset.xylims` settings.)

`load.lists`  
For `loaPlot` only, character vector. In-development alternative to list based arguments. This option identifies plot call arguments that `loaPlot` should manage using `listLoad`. See associated help documentation for further details.

`by.group`, `by.zcase`  
For `loaPlot` only. Arguments for routine by group and by zcase handling of plot inputs. 
Important: These are current under review.

`preprocess`  
For `loaPlot` only, logical, passed to `panelPal`. If TRUE, and used with a correctly configured panel function, this processes the plot input before generating the plot. This means color scales in the different plot panels and the key are automatically aligned and the associated trellis object output contains the panel function outputs rather than the inputs. See `panelPal` Help documents for further information.

`loa.settings`  
For `panel...` functions only, logical, passed to `panelPal` to manage plot reworking. See associated Help documents for further information.

`grid.x`, `grid.y`, `xlim`, `ylim`, `grid`, `panel.scales`  
For `panel.loaGrid` only, grid settings, typically recovered by `loaPlot`.

### Details

`loaPlot` provides lattice-style conditioning/handling for a range of commonly used XYZ plotting options. It is perhaps easiest pictured as a 'mid point' alternative somewhere between the standard `lattice` plot functions `xyplot` and `levelplot`.

The default form of the plot uses an extension of the subscripting methods described by Deepayan Sarkar in Chapter 5 of Lattice (see sections on scatterplots and extensions). The default output is a bubble plot (see example 1 below).

### Value

`loaPlot` returns a trellis object, much like a conventional `lattice` plot function.
**Note**

panel.loaPlot2 is an alternative version of panel.loaPlot that is currently under revision. Please use with care.

loaPlot arguments by.group and by.zcase are currently in revision. Please use with care.

**Author(s)**

Karl Ropkins

**References**

This function makes extensive use of code developed by others.


**See Also**

In loa: **panelPal**

In other packages, see

lattice: **xyplot** and **levelplot**.

**Examples**

```r
## Example 1
## Basic usage

doPlot(Ozone~Solar.R*Temp|Wind>8,
   data=airquality,
   col.regions="Blues")

# Notes:
# Formula structure z ~ x * y | cond like levelplot.
# Data (and groups) assignment like in standard lattice plots.
# By default z is linked to col and cex.
# Unless overridden by user inputs or group or zcase setting.
# Plot passed via ...Handler functions to provide shortcut plot
# reworking, e.g. here colHandler handles color scales
# using col.region to generate a color range.
# (Here, arguments like "Blues" and c("green", "red") are
# allowed and handled using functions in the RColorBrewer
# package.)

# Formula stuctures:
# ~ x * y like xyplot y ~ x
# ~ x * y | cond like xyplot y ~ x | cond
# z ~ x * y like xyplot y ~ x, col=f(z), cex=f(z)
```
1.1.loaPlot

# z ~ x * y | cond  like xyplot y ~ x | cond, col=f(z), cex=f(z)
# z ~ x * y, groups = g  like xyplot y ~ x, groups=g, cex=f(z)
# z1 + z2 ~ x * y  (zcases)
# etc

## Example 2
## Basic modifications

```r
loaPlot(Ozone~Solar.R*Temp, groups=airquality$Wind>8,
        data=airquality)
```

# When groups are applied, by default group id is linked to col.
# The follow example illustrates three options:

```r
loaPlot(Ozone~Solar.R*Temp, groups=airquality$Wind>8,
        data=airquality,
        group.args=c("pch"), pch=c(1,4),
        col="blue")
```

# notes:
# Here, group.args is used to change the default group arguments.
# (So, pch rather than col is used to identify groups.)
# pch is then assigned by group rather than by (x,y) case or z case.
# (See panelPal Help further further details of assignments in loa.)
# col supplied by the user supercedes the default z linkage.
# (So, here cex remains z scales but col is fixed as blue.)

## Example 3
## Key handling

```r
loaPlot(Ozone~Solar.R*Temp, data=airquality,
        col.regions=c("green", "red"))
```

# Key settings are by the key argument (as in lattice)
# or key... arguments via keyHandler and listLoad, so e.g.:

```r
loaPlot(Ozone~Solar.R*Temp, data=airquality,
        col.regions=c("green", "red"),
        key.fun = draw.loaColorKey)
```

# Notes:
# By default the loaPlot uses draw.loaPlotZKey to generate
# its color key unless an alternative is supplied via key.fun.
# (Here, the draw.colorKey wrapper draw.loaColorKey is used to
# generate a color bar similar to that in levelplot.)

## Example 4
## panels
# The combined use of loaPlot, panelPal and appropriately configured
# panel functions provides automatical handling of a range of plot
# elements, e.g.:

loaPlot(Ozone~Solar.R*Temp|Wind>8, data=airquality,
        col.regions="Reds", panel=panel.binPlot)

# Notes:
# Here, the choice of default key is set by the panel... function;
# the panel by default bins data by location and for each bin cell
# calculates the mean Ozone concentration just like a standard
# lattice panel would, but it also tracks these values (calculated
# within the panels) and scales panel-to-panel and panel-to-key
# so users do not have to do that retrospectively; and, finally,
# it retains in-panel calculations so users can recover them.
# (See associated helps for further details: ?panelPal about methods;
# and ?panel.binPlot about the panel function.)

---

## Description

Plotting georeferenced data on maps using lattice and RgoogleMaps

## Usage

```r
GoogleMap(x, data = NULL, panel = panel.loaPlot,
          map = NULL, map.panel = panel.GoogleMapsRaster,
          recolor.map = FALSE, ..., lon.lat = FALSE)

GoogleMap.old(x, data = NULL, map = NULL,
              map.panel = panel.GoogleMapsRaster,
              panel = panel.xyplot,
              recolor.map = FALSE, ...)

googleMap(...)

quickMap(lat, lon, show.data = FALSE, ...)

# map handlers
makeMapArg(ylim, xlim, aspect = NULL,
           recolor.map = FALSE, ...)
```
getMapArg(object = trellis.last.object())

# map panel handlers
panel.GoogleMapsRaster(map)
panel.GoogleMaps(map)

# axis handlers
xscale.components.GoogleMaps(lim, ..., map = map)
yscale.components.GoogleMaps(lim, ..., map = map)
axis.components.GoogleMaps(map, xlim = NULL, ylim = NULL, ...)

Arguments

x For GoogleMap and GoogleMap.old only. A formula setting the plot structure, by default z ~ latitude * longitude | cond. The axis elements latitude and longitude are required, while z and conditioning cond are optional.

data For GoogleMap and GoogleMap.old only. If supplied, the assumed source of the elements of formula x, typically a data.frame.

panel, map, panel For GoogleMap and GoogleMap.old only. The panels to use when generating the plot data and mao layers, respectively. panel is by default the standard loa scatter plot panel panel.loaPlot. map.panel can be the default panel.GoogleMapsRaster or the alternative panel.GoogleMaps.

map For GoogleMap and related functions only. If supplied, a modified RgoogleMaps output, to be used as the plot background. If NULL (default), this is generated using the RgoogleMaps function GetMap, the supplied latitude, longitude ranges, and any additional RgoogleMaps arguments supplied within the call. The map is supplied via makeMapArg which modifies the RgoogleMaps output before returning it to simplify local handling and (lattice) plotting.

recolor.map For GoogleMap and RgoogleMapsWrapper only. If supplied, a vector of elements that R can treat as colors, used as a color scale to recolor map. This uses standard RColorBrewer functions, so can handle arguments like recolor.map = c("white", "grey") for greyscale, etc. Disabled by the default FALSE or NULL.

lon.lat For GoogleMap only, logical. Should the plot formula be applied as z ~ lon * lat | cond? This operation is handled using the formula.type argument in formulaHandler

lat, lon For quickMap only. Numeric vectors of latitude and longitude values.

ylim, xlim, lim The latitude and longitude plot ranges. ylim and xlim are only required by makeMapArg, which uses these to set the requested map size. For the axis handlers (yscale... and xscale...) the local alternative lim is used for both ylim and xlim in generic code. In GoogleMap and quickMap, if supplied, xlim and ylim are passed to lattice function xypplot via LatLon2XY.centered to handle local scaling.

aspect The aspect ratio of the plot. If not supplied (recommended), this is determined based on map size, but can be forced by user.
show.data  For quickMap only, a Logical. Should the lat, lon values supplied be plotted on the map (show.data = TRUE) or just be used to define the range/size of the map being generated? Default show.data = FALSE.

object  For getMapArg only, a lattice plot to recover an RgoogleMaps map from. (If not supplied, this is assumed to last lattice plot.)

...  Additional arguments are passed on to related functions.

For quickMap these are makeMapArg and the lattice function xyplot.

For GoogleMap these are makeMapArg, cexHandler, cexHandler and xyplot.

makeMapArg uses the RgoogleMaps function GetMap. So, most GetMap arguments can be directly accessed from either GoogleMap or quickMap via this route, e.g. maptype = "satellite". The returned object is then modified to simplify its handling by the associated panel and axis functions.

By default both data point colour and size are z-scaled for GoogleMap. If z is supplied, and cex and col are not set by the user in the plot command, these plot properties are managed by cexHandler and colHandler, respectively. cexHandler and colHandler arguments can be passed direct as part of a GoogleMap command to fine-tune these, e.g. cex.range to change the cex range that z values are scaled to and col.region to change the color range that is applied to z. See associated Help documents for further information.

Similarly, argument passing to xyplot in both GoogleMap and quickMap provides access to most common plot parameters, e.g. col, pch, and cex for plot symbol color, type and size, respectively.

getMapArg recovers the map from a lattice plot object generated with GoogleMap. Unless the plot object is supplied in the getMapArg call, this is assumed to be the last lattice (trellis) output.

Details

NOTE: GoogleMap and related panel and axis handling functions are currently in development functions and may be subject to changes.

GoogleMap provides lattice-style conditioning/handling for RgoogleMaps outputs. This uses loaPlot and the latest version of panelPal to manage default panel and key settings.

GoogleMap.old is the previous version of the GoogleMap which uses the previous version of panelPal

googleMap is a GoogleMap wrapper, included because this alternative form of the plot name was used in earlier versions of the package.

quickMap is crude map plotter intended to demonstrate the use of the other 'handler' functions when building dedicated mapping functions.

makeMapArg accepts latitude and longitude ranges and RgoogleMaps function GetMap arguments, and produces an output suitable for use with the panel1.GoogleMapsRaster and panel1.GoogleMaps panel functions or in subsequent GoogleMap calls if, e.g., the users wishes to reuse an existing map.

panel1.GoogleMapsRaster and panel1.GoogleMaps are lattice panel functions that generate map layers for a lattice plot using makeMapArg outputs.

yscale.components.GoogleMaps and xscale.components.GoogleMaps are y- and x-axis handlers for use with the above panels.
axis.components.GoogleMaps is a wrapper that combines yscale.components.GoogleMaps and xscale.components.GoogleMaps and allows both axis to be set from the lattice function argument axis rather than each individually, via yscale.components and xscale.components.

Value

GoogleMap and quickMap return trellis objects, much like conventional lattice plot functions.

makeMapArg returns a modified form of the RgoogleMaps function GetMap output suitable for use as the map argument with the above functions. Note: the automatic assignment of the RgoogleMaps function argument size

getMapArg recovers the map from an existing GoogleMap output.

panel.GoogleMapsRaster and panel.GoogleMaps generate panel outputs suitable for use in standard lattice panel functions.


Note

Google Maps outputs are 2D projections of curve sections of the Earth’s surface. Therefore, the assignment of points within panels and the annotation of latitudes and longitudes along axis needs to be locally handled to account for this.

GoogleMap and quickMaps use RgoogleMaps functions LatLon2XY, LatLon2XY.centered and XY2LatLon to locally scale both axis and data.

Important: Users wanting to add data to these plots, e.g. using update or layers in latticeExtra, should first rescale the data. Likewise, users wanting to add maps to other plots will need to rescale plotted data to use these maps. See Example 1 below.

Important: The Google API returns a map panel larger than the data (latitude, longitude) range requested. However, it does this using a limited number of panel sizes. This means you may get back a map that is larger than necessary. As xlim and ylim are passed to the API when they are called resetting these can produce similar effects (so you may not get exactly the map range you ask for! If you want to manually optimise the map ranges, the best option is currently to start with:

googleMap(..., size=c(640,640))

...and then reduce either or both of these values until you generate an appropriate map size.

Author(s)

Karl Ropkins

References

This function makes extensive use of code developed by others.


See Also

In other packages, see
RgoogleMaps: GetMap, LatLon2XY, LatLon2XY.centered; and, XY2LatLon.
lattice: xypplot, panel.xypplot; and panel.levelplot.

Examples

```r
## Example 1
## quickMap code
## as example of third-party use of functions

quickMap <- function(lat, lon, show.data = FALSE, ...){
  #get map
  map <- makeMapArg(lat, lon, ...)

  #scale axis for map projection
  map.axis.comps <- axis.components.GoogleMaps(map)
  map.axis <- function(components, ...)
    axis.default(components = map.axis.comps, ...)

  #scale data for map projection
  temp <- LatLon2XY.centered(map, lat, lon)
  lat <- temp$newY
  lon <- temp$ newX

  #plot data on map
  xypplot(lat-lon,
          xlim = map$xlim, ylim = map$ylim,
          aspect = map$aspect,
          axis = map.axis,
          panel = function(...){
            panel.GoogleMapsRaster(map)
            if(show.data)
              panel.xypplot(...)
          }, ...)
}

## Example 2
## Off-line GoogleMap examples

# Use a subsample of lat.lon.meuse
temp <- lat.lon.meuse[sample(1:155, 15),]

GoogleMap(zinc~latitude*longitude, col.regions=c("grey", "darkred"),
          ...)
1.3.trianglePlot

Triangle plots

Description

Triangle plot functions for Lattice.

Usage

```r
trianglePlot(x, data = NULL, ..., ref.cols = TRUE)
```

#standard panels

```r
panel.trianglePlot(x = NULL, y = NULL, a0 = NULL, b0 = NULL, c0 = NULL, ..., loa.settings = FALSE, plot = TRUE, process = TRUE)
```
panel.trianglePlotFrame(..., grid = NULL, axes = NULL)
panel.trianglePlotGrid(alim = NULL, blim = NULL, clim = NULL, ...
  ..., grid = TRUE, panel.scales = NULL)
panel.trianglePlotAxes(alim = NULL, blim = NULL, clim = NULL, ...
  ..., axes = TRUE, ticks = TRUE, annotation = TRUE,
  panel.scales = NULL)

# data handlers
triABC2XY(a, b = NULL, c = NULL, ..., force.abc = TRUE,
         if.na = "remove.row", if.neg = "remove.row",
         verbose = FALSE)
triXY2ABC(x, y = NULL, ..., force.xy = TRUE,
         verbose = FALSE)
triLimsReset(ans)

Arguments

**x**

For trianglePlot only, a formula with the general structure z ~ a0 + b0 + c0 | cond.
The elements a0, b0 and c0, the inputs for the three axis on the triangle plot, are
required, while z and conditioning (cond) are optional. (For other functions, x
may be used as the pair to y. See y below.)

data

For trianglePlot only, if supplied, the assumed source of the elements of for-
mula x, typically a data.frame.

...  

Additional arguments.

**ref.cols**

Either a logical to turn off/on grid color-coding or a vector of colors to be applied
to a0, b0 and c0 axes and grids. These are applied to the grid lines and axes tick
and annotation components. Some users, particularly those less familiar with
triangle plots, can find such color referencing helpful when analyzing such plots.
By default, the colorings are quite subtle, so users can see the effect if they look
for it but it does not take over the plot when it is not focused on. Finer control
can be achieved using axes, ticks, grid, etc. (See below).

**y, a, a0, b, b0, c, c0**

(and x in relevant functions). a/a0 , b/b0 and c/c0 are the three scales of the
triangle plot, and x and y are the equivalent 2-D projections. The arguments are
typically options in panel functions (panel... functions), conversion functions
(triABC2XY and triXY2ABC) and the scaling function triLimsReset.

**loa.settings, plot, process**

loaPlot arguments used to manage panelPal activity.

**grid, axes, ticks, annotation**

User-resets for the axes, grid, tick and annotation elements of the plots. These
can be NULL or FALSE to turn off, TRUE to show, a vector (in which case they
are assumed to be color assignments) or a list of standard plot parameters, e.g.
col, lty, lwd, etc. for color, line type and line thickness, etc. Plot parameter as-
signments are applied to all axes unless specific axes are identified. For example,
trianglePlot calls including grid.col = 2 or grids = list(col = 2) make
all axes red, while calls including grid.a0.col = 2 or grid = list(a0 = list(col = 2)
only recolor the first (a0) axis.
1.3.trianglePlot

alim, blim, clim
Delimiters for a, b and c scales, equivalent to xlim and ylim in conventional plots, but less flexible. See Details below for more information.

panel.scales
A local argument, typically a list, that controls the appearance of the a0/b0/c0 axes. This is roughly equivalent to the scales argument used by conventional lattice plots to handle x and y axis, but intended for non-standard scales, such as the triangle axes used here. It can be set directly or used in combination with the local scale(s) handler function `localScalesHandler` to override/hijack standard scales operations. (See note below).

force.abc, force.xy
Logicals. If a list or data.frame is supplied to triABC2XY or triXY2ABC as a source or a/b/c or x/y respectively should appropriately named elements be used regardless of order? See Note below.

if.na
Character. Handling method to be used if NAs are present. The default 'remove.row' replaces all entries in the same row with NAs. (Note: this is different from `na.omit` which would remove the whole row. Here, the row is retained as NAs to maintain indices for conditioning.) Other options currently include: 'make.zero' to replace the NA with 0; and 'keep.as.is' to leave unchanged.

if.neg
Character. Like if.na but for negative values: 'remove.row' to replace all entries in the same row with NAs; 'make.zero' to replace all negative values with 0; 'rescale.col' rescales any column (i.e., a, b or c) that contains a negative from zero by subtracting the minimum.

verbose
Logical, default FALSE. Should a full output be returned? The alternative FALSE generates a minimal report.

ans
For triLimsReset only, a trellis output, e.g. a lattice plot, to be scaled and plotted based on the assumption that it is a trianglePlot.

Details

trianglePlot generates a triangle plot using the lattice framework.

panel.trianglePlot... functions handle the appearance of triangle plot outputs. panel.trianglePlot, which is assigned as the default panel manages both the data layer of the plot and the plot frame (axes, grid, annotation, etc). panel.trianglePlotAxes and panel.trianglePlotGrid generate axes and grid components of the plot, and panel.trianglePlotFrame is a wrapper for these. The data layer, which by default is panel.loaPlot, can be accessed separately using the data.panel argument.

triABC2XY converts supplied (a, b, c) coordinates to an (x, y) scale suitable for use with panel.trianglePlotFrame, etc.

triXY2ABC converts supplied (x,y) coordinates from triangle plots to the associated proportional (a, b, c) scale.

There are various options for range limiting with triABC2XY and triXY2ABC, and therefore trianglePlot as well. Firstly, limits can be set individually with alim, blim and clim, much like with xlim and ylim for conventional plots. However, they can also be set at once using lims, as in e.g. lims = c(0, 1) to set all axes to full ranges, or on the basis of minimum and maximum cut-offs using abc.mins and abc.maxs, respectively.
trianglePlot uses `localScalesHandler` to override normal lattice handling of scales. This allows parameters for axes other than 'x' and 'y' to be passed via the scales argument for axis generation within the plot panel itself. The function does this by recovering the information for each of the local axes (here a0, b0 and c0) from scales, and passing this on to the plot as the argument panel.scales which can then be evaluated by an appropriate panel... function like panel.trianglePlotAxes. At the same time it also resets scales to stop the standard axes being generated. The intention here is two-fold. Firstly, to provide plot users with an axes control mechanism like the standard scales control of x and y that they already know. And, secondly, to provide developers with a simple framework for the quick addition of non-standard axes or scales. See `localScalesHandler` and `panel.localScale` for further details.

trianglePlot uses `getPlotArgs` to manage lattice defaults and plot developer and user resets for the different plot components (axes, ticks, grid, annotation). As with `localScalesHandler`, the intention here is to provide more routine access to higher level plot control.

**Value**

trianglePlot returns trellis objects, much like conventional lattice plot functions. `panel.trianglePlot...` functions are intended for use within a trianglePlot function call.

`triABC2XY` returns a list containing the named components x and y, which are the 2-D (x,y) transformations of supplied (a,b,c) trianglePlot elements.

`triXY2ABC` returns a list containing the named components a, b and c, which are the (a,b,c) trianglePlot coordinates associated with supplied 2-D (x, y) that trianglePlot would generate.

`resetTriLims` returns a supplied trellis object, rescaled based on the assumption that it is a trianglePlot.

**Note**

**General:**

With triangle plots, the (a0, b0, c0) scales are proportional. So regardless of the absolute sizes of a coordinate set (a,b,c), values are plotted and handled as proportions, i.e. a/(a+b+c), b/(a+b+c) and c/(a+b+c), respectively. This means that absolute values of a, b and c are lost when points are plotted on these axes. So, the function `triXY2ABC` returns the relative proportions of a, b and c, not the absolute amounts, when translating a 2-D (x,y) coordinates into the associated (a, b, c) coordinates.

**Development:**

This is an in-development plot, and 'best handling' strategies have not been decided for several elements. So, future versions of these functions may differ significantly from the current version.

In particular:

Current axes assignments, e.g. (a, b, c) versus (a0, b0, c0), etc., have not be finalised. So, these may change in future versions of the function.

Currently, trianglePlot scale adjustment options have been limited. Options under evaluation include: (1) by `alim`, `blim`, `clim` setting, equivalent to `xlim` and `ylim`, (2) by `lims` to set all axes ranges the same, and (3) by `maxs` to setting all axes range maximums and `mins` to set all axes range minimums, etc.

These options are currently only available via the data converters.
One of the issues here is that the axes ranges are all inter-linked. The range of one axes is a function of the other two axes ranges. So, setting these can generate contradictions. For example, \( \text{lims}=c(0,0.1) \) should in theory set all ranges to \((0,0.1)\). But, the triangle \( a = b = c = c(0, 0.1) \) cannot exist. Therefore, the plot would attempt to recover the extended range that includes all the requested ranges \((a = c(0, 0.1), b = c(0, 0.1)\) and \(c = c(0, 0.1))\), which in this case is the full range: \( a = b = c = c(0, 1)\). Suggestions on this topic are very welcome.

**trianglePlot:**

As part of the loa version 0.2.19 update, trianglePlot was rewritten to run with the most recent version of `panelPal` function. This means all plot functions in loa now use the most recent version of `panelPal`.

This update should provide improved plot handling similar to recent versions of `loaPlot` and `GoogleMap` functions which both already (from versions 0.2.0 onwards) use the latest version of `panelPal`.

**panel.trianglePlotAxes:**

Code currently in revision. Please handle with care.

**triABC2XY:**

Code currently in revision. Please handle with care.

**Author(s)**

Karl Ropkins

**References**

These functions make extensive use of code developed by others.

Currently, several triangle plotting methods are under evaluation for use within this package. These are:

- The tri-plot method of Graham and Mudgley:

- The triangle.param method of Chessel (as coded in R package ‘ade4’)

- And the trilinear plot of Allen as reported by Zhu:

In this version of the package tri-plot methods are used for the `triABC2XY` and `triXY2ABC` transforms and a modification `triangle.param` methods is used to calculate suitable values for `alim`, `blim` and `clim`. 
As elsewhere, the use of lattice is also gratefully acknowledged:

See Also

In loa: For in-panel axis/scale generation, see loaPlot, panelPal, localScalesHandler and
panel.localScale.
In other packages: xyplot in lattice.

Examples

## Example 1
## Basic triangle plot usage

trianglePlot(cadmium~copper+lead+zinc|lime,
data=lat.lon.meuse)

# Notes:
# Formula structure z ~ a0 + b0 + c0 |cond, where a0, b0 and
c0 are the three axes of the triangle plot
# Data (and groups) assignment like in standard lattice plots.
# By default z is linked to col and cex.
# Unless overridden by user inputs or group or zcase setting.
# Plot handling is similar to loaPlot
# (So, see ?loaPlot and ?panelPal for further details.)

# Formula variations:
# basic triangle plot without z values assignment
# trianglePlot(~copper+lead+zinc, data=lat.lon.meuse)
# ... with z values set
# trianglePlot(cadmium~copper+lead+zinc, data=lat.lon.meuse)
# ... with grouping
# trianglePlot(cadmium~copper+lead+zinc, groups = lat.lon.meuse$lime,
data=lat.lon.meuse)

## Example 2
## Basic frame (axes, grid, tick, annotation) management

# trianglePlot(~1+1+1, type="n")
# make using type="n"
# empty frame

# trianglePlot(~1+1+1, type="n",
# grid = FALSE)
# turn off grid

# trianglePlot(~1+1+1, type="n",
# grid.col = 2)
# or change plot settings

# trianglePlot(~1+1+1, type="n",
# grid.a0.lty = 1)

# trianglePlot(~1+1+1, type="n",
# grid = list(a0 = list(lty = 1)))
trianglePlot(~1+1+1, type="n",
  grid.alpha = 0.2,
  ticks.alpha = 0.2)          ## grid and tick alpha reset

# notes:
# Here, grid and ticks arguments are used to remove or modify these
# elements of the plot frame individually.
# Setting can be management in list form like in normal lattice or
# in a loa shorthand where e.g. the argument grid.a0.lty = 1 is equivalent
# to grid = list(a0 = list(lty = 1))
# (So, quicker if you are only changing a small number of elements.)

2.1.specialist.panels  Special panel functions 01

Description

Specialist panel functions for use with lattice and loa plots.

Usage

panel.kerneldensity(x, y, z = NULL, ..., n = 20,
  kernel.fun = NULL, panel.range = TRUE,
  process = TRUE, plot = TRUE, loa.settings = FALSE)

panel.binplot(x = NULL, y = NULL, z = NULL,
  breaks=20, x.breaks = breaks, y.breaks = breaks,
  x1=NULL, x2=NULL, y1=NULL, y2=NULL,
  statistic = mean, pad.grid = FALSE, ...,
  plot = TRUE, process = TRUE, loa.settings = FALSE)

Arguments

x, y, z      lattice function arguments passed down to individual panels.
...
Additional arguments, typically passed on. See below.
n
For panel.kerneldensity, the number of x and y cases to estimate when estimating density.
kernel.fun   A function that can estimate kernel densities.
panel.range  For panel.kerneldensity, a logical (default FALSE) indicating if the kernel density estimation data range should be forced to the full panel range. See Below
process, plot, loa.settings
For panel... functions that intended to be handled using panelPal. process and plot, logicals, indicating if the process and plot sections of the panel function should be run. loa.settings, a logical indicating if the safe mode setting should be returned. See below and panelPal Help documents for further details.
breaks, x.breaks, y.breaks

For panel.binPlot, how many break points to introduce when binning data. 
breaks can be used to use the same number of breaks on both axes, while 
x.breaks and y.breaks can be used to set these separately.

x1, x2, y1, y2

For panel.binPlot, vectors giving the bin cell dimensions used when binning 
x and y elements. Typically ignored and calculated within the plot call.

statistic

For panel.binPlot, the function to use when calculating z values for each set 
of binned. By default, this is mean. So, if a z element is supplied in the plot call, 
the data is binned according to x and y values, and the mean of z values within 
each bin reported/plotted. If z is not supplied, statistic is reset to length to 
generate a frequency plot and a warning generated.

pad.grid

For panel.binPlot, Logical, should empty bins be reported? If TRUE, they are 
reported as NAs; if FALSE, they are not reported.

Details

Some specialist panel... functions can be used be used with conventional lattice functions like 
xyplot, e.g.:

xyplot(..., panel = panel.kernelDensity)
xyplot(..., n = 50, panel = panel.kernelDensity)
xyplot(..., panel = function(...)) panel.kernelDensity(..., n = 50))
#etc

However, they are intended for use with panelPal. The combination provides a mechanism for the 
routine preprocessing of panel data, the association of specialist keys, and the routine alignment of 
panel and legend settings in cases where values are reworked within the panel function call.

Typically, these are intended for use with panelPal, e.g.:

loaPlot(..., panel = panel.kernelDensity)
#etc

panel.kernelDensity generates kernel density estimations based on the supplied x and y data 
ranges. It ignores any supplied z information if supplied in the form:

loaPlot(z~x*y, ..., panel = panel.kernelDensity)

panel.binPlot bins supplied z data according to x and y values and associated break points (set 
by break arguments), and then calculates the required statistic for each of these. By default, this 
is mean, but alternative functions can be set using the statistic argument. if no z values are 
supplied, as in

loaPlot(~x*y, ..., panel = panel.binPlot)

... panel.binPlot resets statistic to length and gives a count of the number of elements in 
each bin.

These panel... functions

Value

As with other panel... functions in this package, output are suitable for use as the panel argument 
in loa (and sometimes lattice) plot calls.
Note

Both panel... functions can color both regions and lines. For panel.kernelDensity these are colored regions and contour lines separating them. For panel.binPlot these are the individual data bins and the borders surrounding them. In both cases the color scheme applied to the colored regions are controlled by col.regions and the colors applied to the lines are controlled by col.

panel.kernelDensity passes additional arguments on to the kernel.fun to estimate kernel density and the lattice function panel.contourplot to generate the associated plot. If no kernel.fun is supplied in the panel call, the MASS function kde2d is used to estimate kernel density.

panel.binPlot passes limited arguments on to lrect.

Author(s)

Karl Ropkins

References

These function makes extensive use of code developed by others.


See Also

In loa: panelPal

In lattice: xyplot, panel.contourplot, lrect

Examples

```r
## Example 1
## Specialist kernel density panel example

a <- rnorm(1000)
b <- rnorm(1000)
c <- rnorm(1000)

xyplot(a~b, panel = panel.kernelDensity, at = 0:5*5)
loaPlot(~a*b, panel = panel.kernelDensity)

# Note 1:
# at sets col.regions for the color surface, but, as this is calculated
# in-panel, this is not known at time of call. So, you need to set when
# using specialist panels with standard lattice plots.
# (Same is true for any panel where plot attributes that are set in-panel
# and need to be known in all panels and keys for consistent output.)
# loa panels include separate process and plot steps that panelPal can use
```
2.2.specialist.panels

# to track these.

# Note 2:
# By default, the panel ignores z data.
# compare:
# loaPlot(c~a*b, panel = panel.kernelDensity)  #where z term (c) ignored
# loaPlot(c~a*b, panel = panel.kernelDensity, n=100) #finer surface resolution

## Example 2
## Specialist bin plot panel example

# By default, the panel bins supplied z case as mean
# modify by supplying alternative as statistic
loaPlot(c~a*b, panel = panel.binPlot)
loaPlot(c~a*b, panel = panel.binPlot, statistic=max)

# Note:
# If z is not supplied, statistic defaults to length to give a count
#
# loaPlot(~a*b, panel = panel.binPlot)  #where z term not supplied

#etc.

---

2.2.specialist.panels  Special panel functions 02

Description

In development specialist panel functions for polar plotting

Usage

```
panel.polarPlot(x = NULL, y = NULL, r = NULL, theta = NULL, ..., 
    data.panel = panel.loaPlot, loa.settings = FALSE, 
    plot = TRUE, process = TRUE)
```

#grid, axes and axes labelling

```
panel.polarFrame(..., grid = TRUE, axes = TRUE, labels = TRUE, 
    panel.scales = NULL, grid.panel = panel.polarGrid, 
    axes.panel = panel.polarAxes, labels.panel = panel.polarLabels)
```

```
panel.polarAxes(axes.theta = NULL, axes.r = NULL, thetalim = NULL,
```
2.2.specialist.panels

```r
rlim = NULL, ..., axes = NULL, panel.scales = NULL)

panel.polarGrid(grid.theta = NULL, grid.r = NULL,
    thetalim = NULL, rlim = NULL, ..., grid = NULL,
    panel.scales = NULL)

panel.polarLabels(labels.theta = NULL, labels.r = NULL,
    thetalim = NULL, rlim = NULL, ..., labels = NULL,
    panel.scales = NULL)
```

Arguments

- **x, y** The x and y coordinates of plot points.
- **r, theta** The equivalent polar coordinates of the plot points. If these are not supplied, x and y are assumed to be polar coordinates and these are calculated by the function.
- **...** Additional arguments, typically passed on. For `panel.polarPlot` these are passed to the `data.panel`. See below.
- **data.panel** The panel to use to handle data once polar coordinates have been checked for or generated. For `panel.polarPlot`, by default this is `panel.loaPlot`.
- **loa.settings, plot, process** loa panel management arguments, handled by `panelPal`. See associated help documentation for further details.
- **grid, axes, labels** plot management options for the grid, axis and axis label elements of the plot. These can be logicals (TRUE to include the element or FALSE to remove it) or lists of plot parameters.
- **panel.scales** loa plot management argument used when generating grids, axes and labels within the plot panel.
- **grid.panel, axes.panel, labels.panel** Used by the `panel...Frame` functions to identify the `panel...` functions to use when generating the grid, axes and axis labelling elements of the plot.
- **axes.theta, axes.r, thetalim, rlim** For `panel.polarAxes` axes settings. `axes.theta` and `axes.r` are the theta and r coordinates of the axis reference points, tick marks, etc. `thetalim` and `rlim` are the plot/axes ranges (like `xlim` and `ylim` in standard lattice plots).
- **grid.theta, grid.r** Like `axes.theta` and `axes.r` but for grid.
- **labels.theta, labels.r** Like `axes.theta` and `axes.r` but for labels.

Details

The `panel.polar...` series of the functions are intended for use with `loaPlot`. 
panel.polarPlot generates a 'bubble plot' style output on polar coordinates. It generates axes and annotation within each plot panel using the other panel functions.

panel.polarGrids, panel.polarAxes and panel.polarLabels generate plot grid, axes and axes labelling elements of the plot. panel.polarFrame provides a wrapper for these plot elements.

Users can fine-tune axes, grids and labels by supplying additional arguments in plot calls, or replace these elements with purpose written functions to more completely modify plot appearance.

Value

The panel.polar... functions are intended to be used as the panel argument in loa plot calls. So, e.g.:

a <- 1:360
loaPlot(a=a*a, panel=panel.polarPlot)

They can also be used with relatively simple lattice plots. However, some features of loa plots managed by panelPal, e.g. default plot appearance management, automatic grouping and panel and key alignment will not be available.

Note

panel.polarPlot is in-development. Function arguments may change.

Author(s)

Karl Ropkins

References

These function makes extensive use of code developed by others.


See Also

In loa: loaPlot; and panelPal.

In other packages: xyplot in lattice.

Description

In development specialist panel functions for generating zcase glyph structures.
2.3.specialist.panels

Usage

```r
panel.zcasePiePlot(..., zcase.rescale=TRUE,
           loa.settings = FALSE)

panel.zcasePieSegmentPlot(..., zcase.rescale=TRUE,
           loa.settings = FALSE)
```

Arguments

- `...` Additional arguments, typically setting the color and properties of the plotted glyphs. See below.
- `zcase.rescale` Should the glyph element be rescaled? See below.
- `loa.settings` loa options, to be handled by `panelPal`.

Details

All these `panel...` functions generate glyphs using `z` inputs and plot these at the associated `(x, y)` location. So, for example a called which used one of the panels and the plot formula:

```
z1 + z2+ z3 + z4 ~ x * y
```

... would plot a series of glyphs, each containing four elements that would be scaled according to `z1`, `z2`, `z3` and `z4`, and each at the associated `(x, y)` location. This means there will be one discrete glyph for each row of data supplied to the plot call.

- `panel.zcasePiePlot` generates a series of `x/y` referenced pie graphs. By default, pie dimensions are assigned as: Pie size (radius) proportional to sum of `z` cases and scaled using `cexHandler` (`z1 + z2 + z3 + z4` for the above formula); Number of Pie segments equal to number of `z` cases (so, 4 for the above formula); Pie segment width (phi) proportional to the individual `zcase` (so, `z1 / (z1 + z2 + z3 + z4) * 360` for first pie segment for the above formula).

- `panel.zcasePieSegmentPlot` is a variation on the conventional pie plot where segment radius rather than segment width is varying by `zcase`.

Value

These `panel...` functions are intended to be used as the `panel` argument in `loa` plot calls. So, e.g.:

```
a <- 1:10
b <- 10:1
loaPlot(a+b-a*a, panel=panel.zcasePiePlot)
loaPlot(a+b-a*a, panel=panel.zcasePieSegmentPlot)
```

Note

Functions in development. Arguments may change.
Author(s)
Karl Ropkins

References

These function makes extensive use of code developed by others.

See Also

In loa: loaPlot, panelPal
In other packages: xyplot in lattice.

Examples

```R
## Example 1
## plotting georeferenced pie plots

# Use a subsample of lat.lon.meuse
temp <- lat.lon.meuse[sample(1:155, 15),]

# plot Cu/Pb/Zn pie plots at sampling locations
# loaPlot(copper+1ead+zinc~longitude*latitude,
# panel=panel.zcasePiePlot, data=temp)
# then rescale smaller pie segments on the fly

loaPlot(copper+1ead+zinc~longitude*latitude,
        panel=panel.zcasePiePlot, data=temp)
```

Description

In development specialist panel functions

Usage

```R
panel.compareZcases(x=x, y=y, z=NULL, ...,
                      loa.settings = FALSE)
```
2.4.specialist.panels

Arguments

- **x, y, z**: Standard plot data series, typically vectors.
- **...**: Additional arguments, typically passed on.
- **loa.settings**: loa options, to be handled by panelPal.

Details

The `panel.compareZcases` generates a simple plot which compares z and y elements.

Value

These `panel...` functions are intended to be used as the `panel` argument in `loa` plot calls. So, e.g.:

```r
x <- 1:10
y <- 1:10
z <- y + rnorm(10)
loaPlot(z~x*y, panel=panel.compareZcases, col.regions="Reds")
```

Note

These are ad hoc `panel...` functions. Not sure of their life expectancy...

Author(s)

Karl Ropkins

References

These function makes extensive use of code developed by others.


See Also

In `loa`: `loaPlot`, `panelPal`.

In other packages: `xyplot` in `lattice`. 
3.1.example.data

**example data for use with loa**

---

**Description**

Example data intended for use with examples in loa.

**Usage**

```r
lat.lon.meuse

roadmap.meuse
```

**Format**

- `lat.lon.meuse` is a modified form of the `meuse` data set taken from the `sp` package. Here, coordinate (x,y) elements have been transformed to latitudes and longitudes and the object class has been converted from `SpatialPointsDataFrame` to `data.frame`.
- `roadmap.meuse` is a previously downloaded map intended for use with off-line plot examples using `lat.lon.meuse`.

**Details**

- ```r
library(sp); library(gstat); library(rgdal)
data(meuse)
coordinates(meuse) <- x + y
proj4string(meuse) = CRS("+init=epsg:28992")
meuse1 = spTransform(meuse, CRS("+init=epsg:4326"))
meuse2 = as.data.frame(meuse1)
mzn=meuse2[,c(14,13,4)]
names(mzn)=c("Latitude","Longitude","zinc")
```
- `roadmap.meuse` was generated using:
  ```r
  GoogleMap(zinc~latitude*longitude, data=lat.lon.meuse, size=c(450,500), maptype="roadmap")
  roadmap.meuse <- getMapArg()
  ```

**References**

For `meuse`:

M G J Rikken and R P G Van Rijn, 1993. Soil pollution with heavy metals - an inquiry into spatial variation, cost of mapping and the risk evaluation of copper, cadmium, lead and zinc in
the floodplains of the Meuse west of Stein, the Netherlands. Doctoraalveldwerkverslag, Dept. of Physical Geography, Utrecht University.


Stichting voor Bodemkartering (Stiboka), 1970. Bodemkaart van Nederland : Blad 59 Peer, Blad 60 West en 60 Oost Sittard: schaal 1 : 50 000. Wageningen, Stiboka.

For sp:


Examples

```r
## data structure of lat.lon.meuse
head(lat.lon.meuse)

## Use a subsample of lat.lon.meuse
temp <- lat.lon.meuse[sample(1:155, 15),]

## various loaPlot examples
## using lat.lon.meuse
loaPlot(~longitude*latitude, data=temp)
loaPlot(cadmium~longitude*latitude, data=temp)
loaPlot(cadmium~longitude*latitude, col.regions=c("lightblue", "darkblue"),
        data=temp)
loaPlot(cadmium~longitude*latitude, col.regions=c("lightblue", "darkblue"),
        panel.zcases = TRUE,
        key.z.main="Concentrations", data=temp)

## (off line) GoogleMap example
## using lat.lon.meuse and roadmap.meuse
GoogleMap(zinc~latitude*longitude, data=temp,
          map=roadmap.meuse, col.regions=c("grey", "darkred"))

# Note 1:
# With loaPlot and GoogleMap, note latitude, longitude axes
# assignments:
# loaPlot plots z ~ x * y | cond.
# GoogleMap plots z ~ lat * lon | cond (z ~ y * x | cond)
# Note 2:
# Here, the map argument is supplied so example works off-line.
# If not supplied and R is on-line, GoogleMap will (try to) get map
# from the Google API. Look at:
# GoogleMap(zinc=latitude*longitude, data=lat.lon.meuse,
# col.regions=c("grey", "darkred"))
# (The map will appear slightly different, because non-default
# size and maptype settings were used to make roadmap.meuse. See above.)

### 4.1.panel.pal

**panelPal**

#### Description

lattice plot management using the loa function panelPal

#### Usage

```r
panelPal(ans, panel = NULL, preprocess = FALSE,
          reset.xylims = FALSE, legend = NULL,
          by.group = NULL, by.zcase = NULL, ...)

panelPal.old(x, y, subscripts, at, col.regions, ..., 
             panel = panel.xyplot, ignore = NULL,
             group.fun = NULL)

loaHandler(panel = NULL, ...)
```

#### Arguments

- **ans**
  For panelPal only, a standard trellis object, such as that generated by *lattice*
  function `xyplot`.

- **panel**
  A panel function, e.g. `panel.xyplot`. If supplied in `panelPal` call, typically
  the one used to generate `ans`. If supplied in `panelPal.old`, the panel that is
  intended to used when generating a plot.

- **preprocess**, **reset.xylims**, **legend**, **by.group**, **by.zcase**
  For panelPal only, loa plot management arguments. `preprocess`: Logical, should the supplied panel function be preprocessed? `reset.xylims`: Logical, should the plot dimensions be reset if changed, e.g. by preprocessing? `legend`: the legend as with standard lattice plots, `by.group`: a vector of plot argument names to be linked to any group conditioning, `by.zcase`: a vector of plot argument names to be linked to any z case conditioning. See Details below.
Additional arguments, typically passed on.

For panelPal.old only, panel arguments passed down to individual panels.

Any additional arguments that panelPal.old should ignore and pass on to panel unchecked/unmodified.

Fine control of the standard lattice plot argument group. It can be a vector or list containing the same number of elements as there as groups. These can be functions (or the names of functions as characters) setting individual functions for group or sets of parameters to be evaluated using the panel function. For example, the current NULL default generates a list of col and pch settings that produce a conventional grouped scatter plot output when the group argument is applied to the panel default panel.xyplot. See Details below.

panelPal provides a range of generic plot management features.

Firstly, it allows plot as well as panel defaults to be managed by the panel... function. This allows the panel developer to control plot-level components of the plot output, e.g. which key to use with the plot and what default settings to apply to it. See example 1 below.

Secondly, it uses a generalised extension of the subscripting methods described by Deepayan Sarkar in Chapter 5 of Lattice (see sections on scatterplots and extensions) to automatically handle plot argument subscripting, demonstrated in example 2 below.

Thirdly, it applies an extension of the method used by the hexbin lattice panel to pass hex cell counts (calculated in panels) to the plot key and standardise the assignment of associated parameters within all panels to provide more general panel-to-panel and panel-to-scale. The method is briefly discussed in Chapter 14 of Sarkar.

This method has also been extended by isolating processing and plot components of the panel... function operation allowing results of any calculations made in-panel to be retained rather than lost when plot is generated.

Fourthly, group... and zcase... arguments can used to manage plot group and zcase based plot outputs.

Some panelPal are implemented if specially structured (or loa-friendly) panel... functions are supplied. These are illustrated in the final example below.

loaHandler is a workhorse that assesses information in 'loa' friendly panel... functions. As well as checking this, loaHandler also checks the supplied panel for any default plot settings. This allows users to manage the appearance of a plot from the panel or automatically associated color keys.

Both panelPal and panelPal.old are intended to be used with trellis plot outputs.

panelPal should be employed retrospectively. So, for example:

p1 <- xyplot(...)  
panelPanel(p1, ...)
The previous version, currently retained as `panelPal.old`, was employed developed as a `panel...` function wrapper and intended to be employed within the `plot` call. So, for example:

```r
xyplot(..., panel = function(...) panelPal(..., panel=panel))
```

Because it is run within the `plot` call, and therefore within each panel called, it does not provide features that require panel-to-plot, panel-to-key or panel-to-panel communication.

`loaHandler` returns either a logical (FALSE if not loa 'friendly'; TRUE if loa 'friendly') or a list of default arguments to be used when plotting.

**Note**

The `by`, `group` and `by` `zcase` arguments of `panelPal` and the `group` `fun` argument of `panelPal.old` are all currently under review. Please do not use these.

**Author(s)**

Karl Ropkins

**References**

These function makes extensive use of code developed by others.

lattice:


hexbin:


`panelPal.old` and `panelPal` both apply an extension of the subscripting methods described by Deepayan Sarkar in Chapter 5 of Lattice (see sections on scatterplots and extensions) to automatically handle plot argument subscripting.

`panelPal` applies an extension of the method used by hex bin lattice panel to communicate hex cell counts (calculated in panels) panel-to-panel and panel-to-scale. The method is briefly discussed in Chapter 14 of Sarkar.

**See Also**

`lattice`, `xyplot`,

**Examples**

```r
## the combination of panelPal and specially
## structured panel... functions provides
## several additional plot features:

## example 1
## plot management from the panel... functions.

# loaHandler can be used to see if a panel is loa-friendly
```
loaHandler(panel.xyplot) #FALSE
loaHandler(panel.loaPlot) #panel defaults

# note that these include a list called
# default.settings. These are settings that are
# automatically added to the plot call.

# Here this assigns a specialist key to that
# panel. However, the same mechanism can also
# be used to turn off plot elements like the
# standard lattice axes, when using in panel
# alternatives

# first some silly data
a <- rnorm(1000)
b <- rnorm(1000)

# now compare:
# default plot
# note bubble plot style key
loaPlot(a-a*b)

# bin plot
# with classic color key
loaPlot(a-a*b, panel = panel.binPlot)

## example 2
## automatic subscripting with loa

# Other arguments are not automatically
# aligned with the main plots.

# For example, consider the data:
a <- 1:10
ref <- rep(1:2, each=5)

# and associated lattice xyplot output:
xyplot(a-a|ref, col=ref, pch=19)

# Here, the 'col' argument does not
# automatically track plot conditioning.

# With lattice plots you need to assign
# arguments you want to track in this
# manner using subscripts, as discussed
4.2.plot.structure.handlers

Handler functions for plot structure arguments.

Description

Function(s) for handling (front end) plot arguments like x and strip that manage the plot structure.

Usage

```r
formulahandler(x, data = NULL, groups = NULL, ..., expand.plot.args = TRUE, formula.type = "z~x+y|cond", panel.zcases = FALSE, coord.conversion = NULL, lattice.like = NULL, check.xy.dimensions = TRUE, check.coord.dimensions = TRUE, get.zcase.dimensions = TRUE, output = "extra.args")

stripHandler(..., striplab=NULL)

getzcaseDimensions(...)
```

Arguments

- **x**: A formula to used to generate a lattice plot. See Below.
- **data**: If supplied, the assumed source of the elements of formula x, typically a data.frame.
- **groups**: If supplied, the grouping argument to be used with x and data.
- **...**: Additional arguments are passed on to related functions.
- **expand.plot.args**: For formulahandler only, logical. Should any short elements of the plot structure be expanded?
4.2.plot.structure.handlers

``` R
formula.type For formulaHandler only, character vector or function. The plot structure to be used when generating the plot, e.g. z ~ x * y | cond for loaPlot

panel.zcases For formulaHandler only, logical. Should zcase arguments, e.g. z1 and z2 in z1 + z2 ~ x * y | cond, be treated as panel conditioning terms rather than grouping terms?

coord.conversion For formulaHandler only, function. If supplied, the conversion to use to convert coordinate information supplied using other coordinate systems to (x, y).

lattice.like For formulaHandler only, list. For preprocessing, a list of plot terms that can be passed directly to lattice/loa plots.

check.xy.dimensions, check.coord.dimensions For formulaHandler only, logicals. Should the formula structure be tested before attempting to generate the plot? See Note below.

get.zcase.dimensions For formulaHandler only, logical. Should the dimensions of any multiple zcases be calculated? See Note below.

output For formulaHandler only, character vector. The format to return function output in.

striplab For stripHandler only, character vector. If supplied, the label to add to the panel strip when conditioning is applied. By default, it applies the standard lattice convention, i.e., show for numerics.
```

Details

**formulaHandler** manages the formula component or x element of of loa plot functions.

For example, for loaPlot it assumes the general formula structure z ~ x * y | cond, and applies it in a similar fashion to the lattice function levelplot.

Within the formula part of the plot call x and y are the horizontal and vertical axes, z is any additional information to be used in point, symbol, surface or glyph generation, and cond any additional conditioning to be applied. (The coordinates, x and y, are required elements; z and cond are typically optional.)

**stripHandler** manages the strip component of loa plot functions.

**getZcaseDimensions** tracks the dimensions of multiple z

Value

**formulaHandler** returns a list, containing the plot elements defined in the supplied formula.

**stripHandler** returns a list containing all supplied arguments, subject to the following modifications: strip, Updated or generated if not supplied; striplab, added to strip via the strip argument var.name, if this is undeclared in call.

**getZcaseDimensions** returns a list containing all the supplied arguments, plus two additions arguments (if supplied in the call): zcase.zlim and z.rowsum.lim. zcase.zlim is a list of lim values, one for each zcase. z.rowsum.lim is the range of 'by-row' sums of zcases. These are calculated using any zcase information supplied in the call.
4.3.lims.and.scales.handlers

Note

These functions are in development and may be subject to changes.

The current version of `formulahandler` includes code from the `stats` function `get_all_vars`. It operates in a similar fashion to the previous version but checks `zcase` dimensions.

The previous version of `formulahandler` was a wrapper for the `lattice` function `latticeParseFormula`. This version of `formulahandler` was updated to simplify multiple `z` argument handling.

The latest version of `formulahandler` includes more flexible `formula.type` handling. For example, it can now handle formulas that have more than two coordinates. As a result, the `check.xy.dimensions` argument was replaced with a `check.coord.dimensions` argument. The previous argument will however remain in the function formals and function as before until all related code has been updated.

The latest version of `formulahandler` uses `getZcaseDimensions` to calculate the dimensions of `z` if it is multi-part, e.g. `z1 * z2 * etc ~ x * y` rather than `z ~ x * y`.

Author(s)

Karl Ropkins

References

This function makes extensive use of code developed by others.


See Also

In loa: `loaPlot`; `panelPal`

In other packages: `levelplot` in `lattice`.

4.3.lims.and.scales.handlers

Plot limits and scales handlers

Description

In development functions for limits and scales handling with `lattice` plots.

Usage

```r
limsHandler(x=NULL, y=NULL, z=NULL, ..., lim.borders = 0.1)
localScalesHandler(scales = NULL, ..., allowed.scales = c("x", "y"),
                  disallowed.scales = NULL, remove.box = FALSE)
```
panel.localscale(x.loc, y.loc, lim, ..., panel.scale = NULL,
    label.before = TRUE, x.offset = NULL, y.offset = NULL,
    axis = TRUE, ticks = TRUE, annotation = TRUE)

Arguments

x, y, z x, y and/or z data series.

lim.borders numeric vector, giving the relative border to extend ...lim ranges by when generating axes or scales. The lim.borders are relative proportions. So, the default setting of 0.1 adds an extra +/- 10 are supplied the first three are treated as codex, y and z lim.borders, respectively. If less than three values are supplied, the three values are generated by wrapping. So, the default setting of 0.1 is equivalent to c(0.1, 0.1, 0.1).

scales, panel.scale A list of elements like the scales argument used with lattice functions. Current default elements draw (= TRUE), arrows (= FALSE), tick.number (= 5), abbreviate (= FALSE), minlength (= 4), and tck (= 1).

... Additional arguments.

allowed.scales A character vector containing the names of the axes to be generated for as part of a local axis.

disallowed.scales A character vector containing the names of any axes that are not required. Note: If found, these are removed from scales before evaluation.

remove.box A logical, default FALSE. Should the box lattice typically places around standard plots be removed? This can be useful if you are using a panel... function to generate axes within the plot.

x.loc, y.loc, lim two member vectors setting the x and y locations where the scale is to be drawn (x.loc ans y.loc), and the limits of the range to be annotated on the scale (lim). Note: These are currently handled 'as is', i.e. for both locations and limit, the first element is the start point and the second is the end point, and any other elements are ignored.

label.before, x.offset, y.offset Scale annotation overrides. label.before is a logical, which controls the position of annotation, tick marks and/or arrows, etc relative to the scale line (i.e., above/left before or below/right after). By default panel.localscale generates tick marks and labels at right angles to the scale line/axis. x.offset and y.offset force the offsets when adding tick marks and annotation.

axis, ticks, annotation If supplied, fine controls for the appearance of the axis line, axis tick marks and axis annotation on the generated scale. These can be vectors, in which they are assumed to be color assignments, or lists of common plot parameters, such as col, lty, lwd, etc.
Details

`limshandler` generates `xlim`, `ylim` and/or `zlim` ranges for axes or color scales for use in a range of plots.

`localScalesHandler` provides a relatively crude mechanism for the removal of conventional lattice plot axes and the generation of alternative axes using a `panel...` function like `panel.localScale`.

Value

`limshandler` returns a list containing `...lim` ranges for any of the elements codex, `y` and/or `z` supplied to it.

`localScalesHandler` returns a list containing the elements: `scales`, `panel.scales` and possibly `par.settings`. `scales` turns off the standard axes annotation. `panel.scales` is a list of named elements, one per named axis, describing the axis layout. If `remove.box = TRUE`, the additional argument `par.settings` is also supplied.

All arguments should be passed on to the associated plot.

`panel.scales` or axis-specific elements in `panel.scales` can then be evaluated by an associated `panel...` function run from within the lattice plot call. This would typically take the form:

```
panel.my.axis(panel.scale = panel.scale$axis, ...)
```

`panel.localScale` is a local axis/scale plotter. It can be used in combination with `localScalesHandler`, and should called once for each axis that is required, or it can be used 'stand alone' panel to add a local scale to a lattice plot.

Note

`panel.localScale` is currently in revision. Scale arrows are currently not available.

Author(s)

Karl Ropkins

References

These function makes extensive use of code developed by others.


See Also

In other packages: `xyplot` in `lattice`.

Examples

```r
## See trianglePlot Example 2 for example application
```
4.4.cond.handlers  Plot Conditioning

Description

Plot conditioning handling

Usage

```r
condsPanelHandler(..., conds = NULL, panel = NULL,
   by.cond = NULL, process = TRUE, plot = TRUE)

groupsPanelHandler(..., groups = NULL, panel = NULL,
   by.group = NULL, process = TRUE, plot = TRUE)

zcasesPanelHandler(..., zcases = NULL, panel = NULL,
   by.zcase = NULL, process = TRUE, plot = TRUE)

groupsAndZcasesPanelHandler(panel=NULL, ..., 
   plot = TRUE, process = TRUE)

groupsHandler(z = NULL, groups = NULL, ..., group.ids = NULL,
   handler = "zzz")

zcasesHandler(z = NULL, zcases = NULL, ..., zcases.ids = NULL,
   handler = "zzz")

groupsAndZcasesHandler(..., loa.settings = NULL)

stepwiseZcasesGlyphHandler(zcases = NULL, ..., zcase.ids = NULL,
   panel.elements = NULL, loaGlyph = NULL)
```

Arguments

```r
...

Additional arguments. See Notes below.

conds, panel, by.cond
```

For all supplied additional arguments, conds is a vector of conditioning indices. This is typically a logical, numeric, factor or character vector which can be used to assign other elements undeclared call arguments to specific subsets. panel identifies the panel... function, and should also be supplied so loa can manage processing and plot activities correctly. by.cond identifies routine plot operations associated with the requested conditioning. This can be a list of plot arguments or panel... functions that should be associated with the requested conditioning. See process and plot below and associated Notes.
4.4. cond.handlers

plot, process, loa.settings
Passed to and handled by panelPal. For panels that can be preprocessed, plot and process turn off or on processing and the plotting steps of the panel code. See panelPal Help documentation from further details.

groups, by.group
As conds and by.cond but for grouping.

zcases, by.zcase
As conds and by.cond but for zcase condition.

z, handler
The z data series and any associated plot arguments that need special handling.

group.ids, zcases.ids, zcase.ids
If given, vectors of the unique cases in groups and zcases, respectively.

panel.elements
If given, the names of all plot arguments that have been vectorized by panelPal.

loaGlyph
(For stepwiseZcasesGlyphHandler only), the loa glyph to drawn. See loa.glyphs for further details.

Details

NOTE: These functions are currently in development and may be subject to changes.

condsPanelHandler is a general purpose function that can be used to routinely manage plot conditioning within a panel... function call. It takes all undeclared arguments are supplied to it, and subsets them by unique case in the supplied conds argument. Then it modifies each of these based on the associated elements of by.cond and processes and/or plots the results depending on process and plot settings.

groupsPanelHandler is similar but is intended for use with the plot call argument groups.

zcasesPanelHandler is similar but is intended for use with arguments conditioned within the z term of the plot formula. So, for example, for unique handling of z1 and z2 cases in the plot loaPlot(z1+z2~x*y).

groupsAndZcasesPanelHandler is a wrapper for groups and zcase that allows users to simultaneously and uniquely handle both types of conditioning.

stepwiseZcasesGlyphHandler is a...Handler function for generating glyph plots based on multiple z inputs.

Value

All...PanelHandlers functions are intended for use with panelPal. Using different combinations of these allows plot developers a high degree of flexibility.

Note

This function is in development and may be subject to changes.

Author(s)

Karl Ropkins
4.5.plot.argument.handlers

References

This function makes extensive use of code developed by others.


See Also

panelPal

For information on related functions in other packages, see

lattice: xyplot; panel.xyplot; and panel.levelplot.

Description

Functions for use the routine handling of some common plot arguments.

Usage

cexHandler(z = NULL, cex = NULL, 
    cex.range = NULL, expand.outputs = TRUE, 
    ref = NULL, ..., zlim = NULL)

colHandler(z = NULL, col = NULL, 
    region = NULL, colorkey = FALSE, legend = NULL, 
    pretty = FALSE, at = NULL, cuts = 20, 
    col.regions = NULL, alpha.regions = NULL, 
    expand.outputs = TRUE, ref = NULL, 
    ..., zlim = NULL, output="col")

colRegionsHandler(...)

pchHandler(z = NULL, pch = NULL, pch.order = NULL, 
    expand.outputs = TRUE, ref = NULL, ..., 
    zlim = NULL)

zHandler(z = NULL, expand.outputs = TRUE, 
    ref = NULL, ...)

Common plot argument handlers
Arguments

`z`  
If supplied, a vector of values intended to used as a scale when assigning a property.  
For `cexHandler`, the cex of, e.g., points on a scatter plot. Here, size scales are managed using a reference range `cex.range`, but superseded by `cex` settings, if also supplied.  
For `colHandler`, the color of, e.g., points on a scatter plot. Here, color scales are managed using a `colorkey` method similar to that used by the `lattice` function `levelplot`, see below (arguments `region`, `colorkey`, `pretty`, `at`, `cuts`, `col.regions` and `alpha.regions`). If `z` is NULL or not supplied, all colors are set by `col` if supplied or as the default lattice symbol color if both `z` and `col` are not supplied.  
For `pchHandler`, the pch of, e.g., points on a scatter plot. Here, plot symbols are managed using a reference vector `pch.order`, but superseded by `pch` settings, if also supplied.  
For `zHandler`, any vector that should to expanded by wrapping to a given length, e.g. the length of the x (or y) data series to plotting.

cex, col, pch  
For associated handlers, the parameter value(s) to be managed (i.e., cex for `cexHandler`, etc. Note: In all cases if these are not NULL these supersede any supplied `z` or ... `Handler` modification.

cex.range  
If supplied, the range for `z` to be rescaled to when using this to generate a cex scale. NOTE: `cex.range = FALSE` disables this cex scaling and uses `z` values directly; `cex.range = TRUE` applied default scaling, equivalent to `cex.range = c(0.75, 3)`.

region, colorkey, legend, pretty, at, cuts, col.regions, alpha.regions  
The colorscale settings to be used when generating a colorkey. The most useful of these are probably `col.regions` which can be used to reset the color scale, `alpha.regions` which sets the col.region alpha transparency (0 for invisible to 1 for solid) and `colorkey` which can be a logical (forcing the colorkey on or off) or a list of components that can be used to fine-tune the appearance of the colorkey. Note: The generation of colorscales is handled by `RColorBrewer`.

pch.order  
A vector of symbol ids (typically the numbers 1 to 24) to used when plotting points if, e.g. using a scatter plot. By default, all points are plotted using the first of these pch ids unless any conditioning (e.g. grouping or zcase handling) is declared and linked to `pch`, in which symbols are assigned in series from `pch.order`.

expand.outputs, ref  
`expand.outputs` is a Logical (default TRUE): should outputs be expanded to the same length as `ref`? This can be useful if, e.g., coloring points on a scatter plot that may be conditioned and therefore may require subscript handling, in which case `ref` could be the x or y data series, or any other vector of the same length. Note: if `ref` is not supplied `expand.outputs` is ignored.

zlim  
The range over which the scale is to be applied if not `range(z)`.

output  
For `colHandler`. The function output. Either the `col` vector alone (`output='col'`) or the full list of color parameters.

...  
Additional arguments, currently all ignored.
4.5.plot.argument.handlers

Details

The ...Handler functions are argument handlers intended to routinely handle some common activities associated with plotting data.

cexHandler manages symbol sizes. It generates a (hopefully) sensible cex scale for handling plot symbol size based on a supplied input (z).

colHandler manages colors. It works like the colorkey in levelplot in lattice, to generate a colorscale based on a supplied input (z).

colRegionsHandler is a wrapper for colHandler that can be used to with the col.regions argument.

zHandler expands (by wrapping) or foreshortens vectors.

Value

cexHandler returns a vector, which can be used as the cex argument in many common plotting functions (e.g. plot, xyplot).

colHandler depending on output setting returns either the col vector or a list containing elements (z, col, legend, at, col.regions and alpha.regions), which can be used to create a col series scaled by z and an associated colorkey like that generated by levelplot for other lattice functions (e.g. xyplot).

colRegionsHandler returns a vector of color values suitable for use with the col.regions argument.

pchHandler returns a vector of pch values of an appropriate length, depending on expand.outputs and ref settings.

Note

cexHandler recently revised. Default cex range now smaller, in line with feedback.

Author(s)

Karl Ropkins

References

These function makes extensive use of code developed by others.


See Also

In other packages: See xyplot in lattice.
Examples

```r
#some trivial data
a <- 1:10

## Example 1
## Simple plot with cex handling
myplot1 <- function(x, y, z = NULL, cex = NULL,
                   cex.range = NULL, ...){

  #set cex
cex <- cexHandler(z, cex, cex.range)

  #plot
  xyplot(y~x, cex = cex,...)
}
myplot1(a, a, a)

# compare
# myplot1(a, a)  #like plot(x, y)
# myplot1(a, a, a*100)  #as myplot1(a, a, a)
# #because cex scaled by range
# myplot1(a, b, c,
#     cex.range = c(1,5))  #cex range reset
# myplot1(a, b, c,
#     cex.range = c(10,50),
#     cex = 1)  #cex supersedes all else if supplied

## Example2
## plot function using lists/listUpdates
myplot2 <- function(x, y, z = NULL, ...){

  #my default plot
default.args <- list(x = y~x, z = z,
                      pch = 20, cex = 4)

  #update with whatever user supplied
  plot.args <- listUpdate(default.args, list(...))

  #col Management
  plot.args$col <- do.call(colHandler, plot.args)
do.call(xyplot, plot.args)
}

#with colorkey based on z case
myplot2(a, a, a)
```
4.6.key.handlers

Key handling

Description

Workhorse functions for routine use of keys in plots.

Usage

keyHandler(key = NULL, ..., output = "key")

#keys
draw.loaPlotZKey(key = NULL, draw = FALSE, vp = NULL, ...)
draw.loaColorKey(key = NULL, draw = FALSE, vp = NULL, ...)
draw.loaColorRegionsKey(key = NULL, draw = FALSE, vp = NULL, ...)
draw.zcasePlotKey(key = NULL, draw = FALSE, vp = NULL, ...)

Arguments

key The key to be used.
...
Any additional arguments to be used to modify the key before plotting.
output The format to return the function output in. This is 'key' for all routine (in plot)
use.
draw, vp lattice and grid arguments using when plotting GROB objects. Generally,
these can be ignored.
Details

keyHandler is a general function that routine generates defaults arguments for add a key to a plot. draw...key functions are all specialist plots keys. They are typically modifications of or variations on similar functions in lattice, e.g. draw.key and draw.colorkey.

draw.loaPlotZKey is the default 'bubble plot' key used with loaPlot.

draw.loaColorKey and draw.loaColorRegionsKey are variations on the draw.colorkey function in lattice.

draw.zcasePlotKey is a simple legend based on zcase annotation.

Value

keyHandler return a list of plot arguments to be used to generate a key.

When run within plot calls, the draw...key functions associated color keys. If they are used with loa plots and suitable panel... functions, color scales are automatically aligned.

Note

In Development: Function structures may change in future package updates.

Author(s)

Karl Ropkins

References

These functions make extensive use of code developed by others.


See Also

In other packages: See xyplot in lattice.
Usage

```
parHandler(scheme = NULL, ...)
```

```c
scheme
```

getArgs(source = TRUE, local.resets = TRUE,
user.resets = TRUE, is.scales.lines = FALSE,
elements = NULL, ..., defaults = list(),
defaults.only = FALSE)
```

getPlotArgs(defaults.as = "axis.line", source = TRUE,
local.resets = TRUE, user.resets = TRUE,
elements = NULL, ..., is.scales.lines = NULL,
defaults.only = TRUE)
```

isGood4LOA(arg)
```

Arguments

scheme

The color scheme to apply. This can be a list of parameters to apply or a char-
acter vector for a pre-defined scheme. Current pre-defined schemes include
'greyscale' (for black and white figures).

source, local.resets, user.resets

When recovering plot arguments with `getArgs` or `getPlotArgs`, places to search
for relevant parameters. If supplied these would typically be vectors or lists. If
vectors, they are assumed to be `col` setting. If lists, they are assumed to be lists
of named parameters for inclusion. There are two cases that need to be handled
specially: (1) some sources, `local.resets` and/or `user.resets` may contain
both axis-specific and general information, e.g. For a scales list, parameters
to be applied just to the x axis in `scales$x` and parameters to be applied to all
scales in `scales`. In such cases these need to be checked in order (see `elements`
below.) (2) Some sources, e.g. axis scales, contain both text and line param-
eters, with e.g. line settings declared as `col.line`, etc., rather than `col`, etc.,
(which are intended for use with text.) When supplied these need to be handled
correctly (see `is.scales.lines` below). `local.resets` and `user.resets` are
intended as overrides for the code developer and user, respectively. These can
be logicals as well as vectors or lists. If logicals they turn on/off the associ-
ated plot components (using `isGood4LOA`). The check/update order is `source`,
then `source$element`, then `local.reset`, then `local.reset$element`, then
`user.reset`, then `user.reset$element`. This means that the developer always
has last say regarding the default appearance of a plot component and the user
always has the very last say from the command line if the local.reset is included as a formal argument in that plot.

is.scales.lines
When recovering arguments with getArgs or getPlotArgs, should source be treated as a lattice scales list? If so, and source is checked for line parameters, line-specific terms such as col.line, etc., will be recovered as col, etc., while general terms (meant for text in scales lists) will be ignored. (Note: getPlotArgs guesses this based on defaults.as if not supplied.)

elements
When recovering arguments with getArgs or getPlotArgs, this identifies the elements in source, local.resets and user.resets that may contain case-specific information. As with lattice handling of scales axis-specific information in source$element(s) is assumed to take priority over general information in source. (Note: if elements are not declared only general/top level information in source, local.resets and user.resets is considered at present.)

... Other arguments, often ignored.

defaults, defaults.only, defaults.as
When recovering arguments with getArgs, defaults is an optional 'fall-back' in case nothing is recovered from source, local.resets and user.resets. defaults.only is a logical: if TRUE only parameters named in defaults are searched for, otherwise all parameters are recovered. With getPlotArgs, defaults.as selects an appropriate default. This should be a trellis parameter name, e.g. 'axis.line', 'axis.text', etc. The function uses this to identify appropriate plot parameters to search for/select, e.g. pch, col, cex, etc for 'plot.symbol', and to identify default values for each of these (if defaults.only = TRUE).

arg
For isGood4LOA a plot argument that can used to turn a plot panel or panel component on or off.

Details
getArgs returns a list of parameters/values based on lattice, developer and user settings. If multiple elements are identified as containing case-specific information, the list will contain one list of plot parameters for each named element.

getPlotArgs is a variation of getArgs intended for use with panel... and l... type lattice functions. It returns a list of plot parameters for different plot components, e.g. symbols, lines, or text.

isGood4LOA is a simple workhorse that checks if a supplied arg should be used by loa. (See value and note below.)

parHandler manages the default appearance of plots.

Value
getArgs and getPlotArgs return lists of located parameters/values. For example, the call
getPlotArgs(default.as = "axis.line")
returns a list containing the lattice defaults for an axis line (alpha, col, lty and lwd) These can then be used in combination with appropriate x and y values in llines, or panel.lines calls. The
arguments `local.resets` and `user.resets` can be added into the call to provide developer and user overrides. (See note below.)

`isGood4L0A` returns a logical (TRUE or FALSE), depending on the type of a supplied argument. This returns FALSE for NULL, for all FALSE logicals, and any arg that has previously been tagged as 'not wanted'.

`parHandler` returns a list a list suitable for use as `par.settings` with most `lattice` plots.

**Note**

`getPlotArgs` is intended as a 'workhorse' for plot developers, to recover `lattice` settings, impose their own preferences on these, and in turn to provide users with similar options to quickly override developer settings.

`isGood4L0A` only exists because I, perhaps wrongly, equate `arg = NULL` with `arg = FALSE` when that argument is a component of a plot defined in the plot formals. For example, in `trianglePlot` I want `grids = NULL` to turn off the plot grids much like `grids = FALSE`, but got fed up always writing the same everywhere. Does not mean it is right, particularly useful or even clever...

The `getPlotArgs/isGood4L0A` combination is a first attempt at providing plot developers with a simple tool to integrate plot argument management by `lattice`, the plot developer and the plot user. It is intended to be applied in the form shown in the Examples below.

Axis, tick, grid and annotation handling in `trianglePlot` is intended to illustrate this type of application.

**Author(s)**

Karl Ropkins

**References**

These function makes extensive use of code developed by others.


**See Also**

In other packages: See `xyplot` in `lattice`.

**Examples**

```r
# getPlotArgs/isGood4L0A notes

# in formals
# my.plot <- function(..., user.reset = TRUE, ...)

# in main code body
# local.resets <- [what developer wants]
# plot.arg <- getPlotArgs("[type]", source, local.reset, user.reset)

# in panel call
```
list handlers

List manipulation

Description

Workhorse functions for routine list handling in loa and elsewhere.

Usage

```r
listHandler(a, use = NULL, ignore = NULL,
  drop.dots=TRUE)
listUpdate(a, b, use = NULL, ignore = NULL,
  use.a = use, use.b = use,
  ignore.a = ignore, ignore.b = ignore,
  drop.dots = TRUE)
listExpand(a, ref = NULL, use = NULL,
  ignore = NULL, drop.dots = TRUE)
listLoad(..., load = NULL)
```

Arguments

- **a**: A required list. The list to be modified.
- **b**: For listUpdate only, a required second list, the contents of which are used to update `a` with.
- **use, use.a, use.b**: Vectors, all defaults NULL. If supplied, a vector of the names of list entries to be used. Other entries are then discarded. `use` is applied to all supplied lists, while `use.a`, `use.b`, etc. can be used to subset `a` and `b` lists individually.
- **ignore, ignore.a, ignore.b**: Vectors, default NULL. As with `use`, etc., but for entries to be ignored/not passed on for modification.
4.8.list.handlers

ref
For listExpand only, a vector, default NULL. A reference data series, the length of which is used as the expansion length to be applied when wrapping of list entries.

drop.dots
Logical, default TRUE. If TRUE, this removes "..." entries from list names before updating.

... 
For listLoad only, any additional arguments.

load
For listLoad only, a vector, default NULL. The names of any lists to be automatically generated from the additional arguments supplied as part of the command call.

Details
listHandler is a general function used by other list... functions for routine list preprocessing.
listUpdate is a list handler intended for use when managing user updates for default options (see examples).
listExpand is a list handler that expands vectors to a given reference length, intended for use for data wrapping.
listLoad is a list generator. See Note below.

Value
By default, all list... functions return results as lists.
listHandler, listUpdate and listExpand functions all return a modified (or updated) version of supplied list a.
listLoad (in-development) returns modified (or updated) version of additional arguments as a list. See Note below.

Note
listLoad is an in-development workhorse function that generates lists based on the supplied load argument.
It assumes each element of load is the name of an expected list and searches the associated additional arguments for arguments to populate it with using the rule 'load[arg] is an element of list [load]'. So, for example, for a call including the arguments load = 'key' and key.fun = draw.colorkey, it would strip out both arguments and return key = list(fun=draw.colorkey). Used in functions in series it allowed list-in-list that can be commonplace when modifying, for example, key elements in conventional lattice plots to be simplified.

Author(s)
Karl Ropkins

References
These functions make extensive use of code developed by others.
See Also

lattice, xyplot.

Examples

```r
## Example 1
## general

# two lists
list1 <- list(a = 1:10, b = FALSE)
list2 <- list(b = TRUE, c = "new")

# updating a with b
# keeps unchanged list1 entry, a
# updates changed list1 entry, b
# adds new (list2) entry, c
listUpdate(list1, list2)

## Example 2
## use in plot functions
## to simplify formals

# some data
a <- 1:10
b <- rnorm(10, 5, 2)

# a bad plot function
badplot <- function(x, ...){
    # setting defaults in xyplot call itself
    xyplot(x = x, pch = 20, col = "red",
           panel = function(...){
               panel.grid(-1, -1)
               panel.xyplot(...)
               panel.abline(0,1)
           }, ...)
}

# badplot(a-b)  #OK

# compare with
# badplot(a-b, xlim=c(0,10))  #OK
# badplot(a-b, col= "blue")  #not OK

# (because col hardcoded into badplot function
# AND duplicated in call and '...'
# so user cannot update col
```
4.9.loa.shapes

Description

Simple shapes.

```r
# a standard correction

stdplot <- function(x, pch = 20, col = "red", ...){
    # setting defaults in xyplot call itself
    xyplot(x = x, pch = 20, col = "red",
           panel = function(x=x, pch=pch, col=col, ...){
               panel.grid(-1, -1)
               panel.xyplot(x=x, pch=pch, col=col, ...)
               panel.abline(0,1)
           }, ...)
}

# stdplot(a-b)  # OK
# stdplot(a-b, col = "blue",
#     xlim=c(1:20))  # also OK

# An alternative correction using lists and
# listUpdate that removes the need for formal
# definition of all modified plot arguments

myplot <- function(x, ...){
    # defaults I set for myplot form of xyplot
    mylist <- list(x = x, pch = 20, col = "red",
                   panel = function(...){
                       panel.grid(-1, -1)
                       panel.xyplot(...)
                       panel.abline(0,1)
                   })
    # plot
    do.call(xyplot, listUpdate(mylist, list(...)))
}

# myplot(a-b)  # OK
# myplot(a-b, col = "blue",
#     xlim=c(1:20))  # also OK
```
Usage

loaPolygon(x, y, ..., polygon = NULL,
    loa.scale = NULL)
loaCircle(..., polygon = NULL, radius = 1)
loaPieSegment(..., polygon = NULL, start = 0,
    angle=360, radius = 1, center=TRUE)

Arguments

  x, y    The x and y points at which to plot the requested shape.
  ...    Any additional arguments, usually passed on.
  polygon A list with elements x and y giving the polygon/shape to be plotted.
  loa.scale A list of parameters that can be used to fine-tune the polygon plotting.
  radius The radius to used when drawing either circles or pie segments.
  start, angle When drawing pie segments, angle the angle of of the segment and start point.
  center Should the segment begin and end at the center?

Details

loaPolygon is a general function for drawing polygons. It is intended as an alternative to lpolygon, and other standard loa... shapes are typically wrappers for this function.
loaCircle draws a circle with an origin at (x, y).
loaPieSegment draws a pie segment (or slice of cake) shape. It is typically used as building block for pie plots and other similar glyph structures.

Value

All these functions generate simple shapes and are intended to be run within panel... functions as building blocks for more complex glyph type structures.

Author(s)

Karl Ropkins

References

These functions make extensive use of code developed by others.

See Also

In other packages: See lrect, and similar, in lattice
Description

Recovering information from existing lattice plots.

Usage

getXY(n = -1, ..., unit = "native", scale.correction = NULL)

getLatLon(..., map = NULL, object = trellis.last.object(),
        scale.correction = function(x) {
          temp <- XY2LonLon(map, x$x, x$y)
          as.list(as.data.frame(temp))
        })

screenLatticePlot(object = trellis.last.object(), ...)

Arguments

- **n**: If positive, the maximum number of points to locate. If negative (default), unlimited.
- **unit**: The unit to use when reporting located points, by default "native".
- **scale.correction**: The correction to apply if the plot has locally scaled axes. See Note below.
- **map, object**: For getLatLon only. The plot layer as generated makeMapArg and the plot object generated by GoogleMap. The map is strictly required as a reference when converting plot points to associated latitude, longitude values, and can be supplied directly as map, or recovered from the plot, which can be supplied as object. If neither are supplied (as in default use), the function attempts to recover map from the last lattice plot via trellis.last.object.
- **...**: Additional arguments, passed on to related functions.

These may be subject to revision, but are currently: trellis.focus for panel selection (if working with multi-panel plots) and lpoints to set point properties (if marking selected points). For getLatLon, additional arguments are also passed to XY2LonLon for x, y to latitude, longitude rescaling.

Details

getXY is an interactive function which returns the locations of points on a plot selected using the mouse (left click to select points; right click and stop to end point collection; escape to abort without returning any values).
It is a wrapper for the `grid` function `grid.locator` that behaves more like `locator`, the equivalent function intended for use with `plot` outputs.

By default `getXY` selections are not automatically marked. Adding common plot parameters to the function call overrides this behaviour, e.g. to add red symbols and lines.

```r
ans <- getXY(col = "red", pch = 4, type = "b")
```

`getXY` also provides a mechanism to handle data plotted on locally scaled axes. See Note below.

`getLatLon` is wrapper for `getXY` for use with GoogleMap outputs and other similarly georeferenced plots. See Note below.

`screenLatticePlot` is a crude plot screening function. It is currently in development.

### Value

`getXY` returns the x and y coordinates of the selected points on a plot as a list containing two components, `x` and `y`.

`getLatLon` returns the latitude and longitude values of the selected points on a map as a list containing two components, `lat` and `lon`.

### Note

`getXY` recovers the (x, y) coordinates of points selected on a previously generated plot.

Some plots, use local scaling. For example, when plotting latitude, longitude data on a map a scale correction may be used to account for the curvature of the Earth. Similarly, if different data series are plotted on primary and secondary axes in a single plot, some or all data may be normalised. In such cases scaling may be local, i.e. what you actually plot may not be exactly what the annotation says it is.

Using `getXY` on such plots would recover the actual (x, y) coordinates of the points selected.

However, corrections can be applied using `scale.correction`, if it is supplied, to convert these to the same scale as the axes annotation. The correction should be a function that can be applied directly to a standard `getXY` output (a list of x and y values) and rescale x and y to give their ‘corrected’ values.

`getLatLon` provides an example of the mechanism, and is for use with georeferenced plots that have been locally scaled using `rgoogleMaps` functions like `LatLon2XY`. `getLatLon` uses `XY2LatLon` to rescale x and y values and then as... functions to convert the outputs of this step to a list format like that generated by `locator`, `grid.locator` or `getXY`.

### Author(s)

Karl Ropkins

### References

This function makes extensive use of code developed by others.


See Also

In other packages: See `grid.locator`; `trellis.focus` and `lpoints` in `lattice`. See `LatLon2XY` in `RgoogleMap`. 
Index

*Topic datasets
3.1.example.data, 28

*Topic methods
1.1.loaPlot, 4
1.2.GoogleMap.and.geoplotting.tools, 8
1.3.trianglePlot, 13
2.1.specialist.panels, 19
2.2.specialist.panels, 22
2.3.specialist.panels, 24
2.4.specialist.panels, 26
4.1.panel.pal, 30
4.2.plot.structurehandlers, 34
4.3.lims.and.scales.handlers, 36
4.4.cond.handlers, 39
4.5.plot.argument.handlers, 41
4.6.key.handlers, 45
4.7.other.panel.functions, 46
4.8.list.handlers, 50
4.9.loa.shapes, 53
5.1.plot.interactives, 55

axis.components.GoogleMaps
(1.2.GoogleMap.and.geoplotting.tools), 8

cexHandler, 3–5, 10, 25
cexHandler
(4.5.plot.argument.handlers), 41
colHandler, 3–5, 10
colHandler
(4.5.plot.argument.handlers), 41
colRegionsHandler
(4.5.plot.argument.handlers), 41
condsPanelHandler, 3
colsPanelHandler (4.4.cond.handlers), 39
draw.loaColorKey (4.6.key.handlers), 45
draw.loaColorRegionsKey
(4.6.key.handlers), 45
draw.loaPlotZKey (4.6.key.handlers), 45
draw.zcasePlotKey (4.6.key.handlers), 45

example.data (3.1.example.data), 28

formulaHandler, 3, 4, 9
formulaHandler
(4.2.plot.structure.handlers), 34

getArgs, 3
getArgs (4.7.other.panel.functions), 46
panel.kernelDensity
  (2.1.specialist.panels), 19
panel.levelplot, 12, 41
panel.loaGrid (1.1.loaPlot), 4
panel.loaPlot, 9
panel.loaPlot (1.1.loaPlot), 4
panel.loaPlot2 (1.1.loaPlot), 4
panel.localScale, 16, 18
panel.localScale
  (4.3.lims.and.scales.handlers), 36
panel.polarAxes
  (2.2.specialist.panels), 22
panel.polarFrame
  (2.2.specialist.panels), 22
panel.polarGrid
  (2.2.specialist.panels), 22
panel.polarLabels
  (2.2.specialist.panels), 22
panel.polarPlot, 3
panel.polarPlot
  (2.2.specialist.panels), 22
panel.trianglePlot (1.3.trianglePlot), 13
panel.trianglePlotAxes
  (1.3.trianglePlot), 13
panel.trianglePlotFrame
  (1.3.trianglePlot), 13
panel.trianglePlotGrid
  (1.3.trianglePlot), 13
panel.xyplot, 12, 41
panel.zcasePiePlot
  (2.3.specialist.panels), 24
panel.zcasePieSegmentPlot
  (2.3.specialist.panels), 24
panelPal, 3, 5, 6, 10, 14, 17–21, 23, 24, 26, 27, 36, 40, 41
panelPal (4.1.panel.pal), 30
parHandler (4.7.other.panel.functions), 46
pchHandler
  (4.5.plot.argument.handlers), 41
plot, 43, 56

quickMap
  (1.2.GoogleMap.and.geoplotting.tools), 8

RColorBrewer, 9, 42
roadmap.meuse (3.1.example.data), 28
screenLatticePlot
  (5.1.plot.interactives), 55
stepwiseZcasesGlyphHandler
  (4.4.cond.handlers), 39
stripHandler
  (4.2.plot.structure.handlers), 34
trellis.focus, 55, 57
triABC2XY (1.3.trianglePlot), 13
trianglePlot, 2, 3, 49
trianglePlot (1.3.trianglePlot), 13
triLimReset (1.3.trianglePlot), 13
triXY2ABC (1.3.trianglePlot), 13
xscale.components.GoogleMaps
  (1.2.GoogleMap.and.geoplotting.tools), 8
XY2LatLon, 11, 12, 55
xyplot, 5, 6, 9, 10, 12, 18, 21, 24, 26, 27, 30, 32, 38, 41, 43, 46, 49, 52
yscale.components.GoogleMaps
  (1.2.GoogleMap.and.geoplotting.tools), 8
zcasesHandler (4.4.cond.handlers), 39
zcasesPanelHandler (4.4.cond.handlers), 39
zHandler, 3
zHandler (4.5.plot.argument.handlers), 41