Package 'trelliscope'

September 20, 2017

Type Package

Title Create and Navigate Large Multi-Panel Visual Displays

Version 0.9.8

Date 2017-09-20

Maintainer Ryan Hafen <rhafen@gmail.com>

Description An extension of Trellis Display that enables creation, organization, and interactive viewing of multi-panel displays created against potentially very large data sets. The dynamic viewer tiles panels of a display across the screen in a web browser and allows the user to interactively page through the panels and sort and filter them based on ``cognostic'' metrics computed for each panel. Panels can be created using many of R's plotting capabilities, including base R graphics, 'lattice', 'ggplot2', and many 'htmlwidgets'. Conditioning is handled through the 'datadr' package, which enables 'Trelliscope' displays with potentially millions of panels to be created against terabytes of data on systems like 'Hadoop'. While designed to scale, 'Trelliscope' displays can also be very useful for small data sets.

URL http://deltarho.org/docs-trelliscope

BugReports https://github.com/delta-rho/trelliscope/issues

License BSD_3_clause + file LICENSE

LazyData yes

NeedsCompilation no

Depends R (>= 3.0.0), datadr (>= 0.8.5),

Imports lattice (>= 0.20-23), ggplot2, data.table, base64enc, shiny (>= 0.12.0), htmlwidgets (>= 0.5.0), digest, jsonlite, hexbin, png, jpeg, DistributionUtils, htmltools, curl, rsconnect

Suggests Cairo, dplyr, testthat (>= 0.11.0), roxygen2 (>= 5.0.1), scagnostics, housingData

RoxygenNote 6.0.1

Author Ryan Hafen [aut, cre],
Jeremiah Rounds [ctb],
Barret Schloerke [ctb],
Landon Sego [ctb]

Repository CRAN

Date/Publication 2017-09-20 16:13:35 UTC

R topics documented:

`trelliscope-package` *Trelliscope: Create and Navigate Large Multi-Panel Visual Displays*

Description

An extension of Trellis Display that enables creation, organization, and interactive viewing of multi-panel displays created against potentially very large data sets. The dynamic viewer tiles panels of a display across the screen in a web browser and allows the user to interactively page through the panels and sort and filter them based on "cognostic" metrics computed for each panel. Panels can be created using many of R's plotting capabilities, including base R graphics, lattice, ggplot2, and many htmlwidgets. Conditioning is handled through the datadr package, which enables Trelliscope displays with potentially millions of panels to be created against terabytes of data on systems like Hadoop. While designed to scale, Trelliscope displays can also be very useful for small datasets.

Details

http://deltarho.org/docs-trelliscope/

Author(s)

Ryan Hafen

Examples

```
help(package = trelliscope)
```

applyCogFn *Apply Cognostics Function to a Key-Value Pair*

Description

Apply cognostics function to a key-value pair, obtaining additional default cognostics like the conditioning variable values in the case of conditioning variable division, the panel key, and between-subset variables.

Usage

```
applyCogFn(cogFn, kvSubset, conn = NULL, ...)
```

Arguments

cogFn	cognostics function
kvSubset	key-value pair
conn	the connection object or ddo/ddf object from which the key/value pair came (see details)
...	additional parameters for special cases (handled internally)

Details

The conn connection object is required in the case of a local disk connection so that the panel key default cognostic can be computed based on the file hash function, if used.

Note

This function is used inside of makeDisplay and is exposed for users who are curious about what the complete output of a cognostics function will look like.

See Also

cog, makeDisplay

Examples

```
# create a division with a between-subset variable
d <- divide(iris, by = "Species",
  bsvFn = function(x) list(msl = mean(x$Sepal.Length)))
# create a cognostics function that gets max sepal length
cogFn <- function(x)
  list(maxsl = max(x$Sepal.Length))
# apply the cognostics function to the first key-value pair
applyCogFn(cogFn, d[[1]])
```

`batting` *Lahman's Batting Data for 2014*

Description

2014 Batting statistics from Sean Lahman's baseball statistics database (released under CC BY-SA 3.0 license).

Usage

```
data(batting)
```

Format

A data frame with 1435 rows and 22 variables

Details

- playerID Player ID code
- yearID Year
- stint player's stint (order of appearances within a season)
- teamID Team; a factor
- lgID League; a factor with levels AA AL FL NL PL UA
- G Games: number of games in which a player played
- AB At Bats
- R Runs
- H Hits: times reached base because of a batted, fair ball without error by the defense
- X2B Doubles: hits on which the batter reached second base safely
- X3B Triples: hits on which the batter reached third base safely
- HR Homeruns
- RBI Runs Batted In
- SB Stolen Bases
- CS Caught Stealing
- BB Base on Balls
- SO Strikeouts
- IBB Intentional walks
- HBP Hit by pitch
- SH Sacrifice hits
- SF Sacrifice flies
- GIDP Grounded into double plays

Source

Lahman, S. (2015) Lahman's Baseball Database, 1871-2014, 2015 version, http://baseball1.com/statistics/.

Examples

```
## Not run:
library(lattice)

vdbConn(tempfile(), autoYes = TRUE)

# make "splod" directly from a data frame
splod(batting, name = "batting",
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))

# first transform the data into a "splodDat" object
batSplodDat <- makeSplodData(batting,
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))
# now make "splod"
splod(batSplodDat, name = "batting2", data = batSplodDat)

# custom panel function (color by league)
mySplodFn <- function(d) {
  xyplot(jitter(y) ~ jitter(x), groups = lgID, data = d,
    xlab = getSplitVar(d, "xVar"),
    ylab = getSplitVar(d, "yVar"),
    auto.key = TRUE
  )
}

splod(batSplodDat, name = "batting3",
  data = batSplodDat, panelFn = mySplodFn)

view()

## End(Not run)
```

---

cleanupDisplays *Remove Backed-Up Trelliscope Displays*

Description

Remove all display directories ending with "_bak"

Usage

```
cleanupDisplays(conn = getOption("vdbConn"))
```

Arguments

conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists

See Also

Other display_manipulation: getDisplay, listDisplays, removeDisplay, restoreDisplay, updateDisplay

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)

vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)

# look at a list of the displays in this vdb:
listDisplays()

# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
```

```
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

---

cog *Create a Cognostics Object*

Description

Create a cognostics object. To be used inside of the function passed to the cogFn argument of makeDisplay for each cognostics value to be computed for each subset.

Usage

```
cog(val = NULL, desc = "", group = "common", type = NULL,
  defLabel = FALSE, defActive = TRUE, filterable = TRUE,
  sortable = TRUE, log = NULL)
```

Arguments

val	a scalar value (numeric, characer, date, etc.)
desc	a description for this cognostic value
group	optional categorization of the cognostic for organizational purposes
type	the desired type of cognostic you would like to compute (see details)
defLabel	should this cognostic be used as a panel label in the viewer by default?
defActive	should this cognostic be active (available for sort / filter / sample) by default?
filterable	should this cognostic be filterable? Default is TRUE. It can be useful to set this to FALSE if the cognostic is categorical with many unique values and is only desired to be used as a panel label.
sortable	should this cognostic be sortable?
log	when being used in the viewer for visual univariate and bivariate filters, should the log be computed? Useful when the distribution of the cognostic is very long-tailed or has large outliers. Can either be a logical or a positive integer indicating the base.

Details

Different types of cognostics can be specified through the `type` argument that will affect how the user is able to interact with those cognostics in the viewer. This can usually be ignored because it will be inferred from the implicit data type of `val`. But there are special types of cognostics, such as geographic coordinates and relations (not implemented) that can be specified as well. Current possibilities for `type` are "key", "integer", "numeric", "factor", "date", "time", "geo", "rel", "hier", "href".

Value

object of class "cog"

See Also

makeDisplay, cogRange, cogMean, cogScagnostics, cogLoessRMSE

Examples

```
d <- stack(data.frame(EuStockMarkets))
d$time <- rep(as.numeric(time(EuStockMarkets)), 4)
d$year <- floor(d$time)

byIndexYear <- divide(d, by = c("ind", "year"))

cogFn <- function(x)
  list(lormse = cogLoessRMSE(values ~ time, data = x, span = 0.3, degree = 2),
       slope  = cogSlope(values ~ time, data = x),
       range  = cogRange(x$values),
       mean   = cogMean(x$values),
       max    = cog(max(x$values, na.rm = TRUE), desc = "max value"))

applyCogFn(cogFn, byIndexYear[[1]])

library(lattice)
panelFn <- function(x)
  xyplot(values ~ time, data = x,
    panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.loess(x, y, span = 0.3,
        degree = 2, evaluation = 200, col = "black")
    })

vdbConn(tempfile(), autoYes = TRUE)
makeDisplay(byIndexYear, name = "ts_index_year",
  cogFn = cogFn, panelFn = panelFn)

## Not run:
# sort and fiter the index/year panels by slope and loess RMSE
view(name = "ts_index_year")

## End(Not run)
```

cogDisplayHref *DisplayHref Cognostic*

Description

Create href that points to another trelliscope display with optional state

Usage

```
cogDisplayHref(state, label = "link", desc = "display link",
  group = "common", target = "_blank", defLabel = FALSE,
  defActive = FALSE, filterable = FALSE, sortable = TRUE, log = FALSE)
```

Arguments

state	the state of the display to link to, using stateSpec - at a minimum the name and group of the display must be specivied - additionally, default parameter settings (such as layout, sorting, filtering, etc.) can be set to be in effect when the display is launched (see stateSpec for details)
label	label of the href
desc, group, defLabel, defActive, filterable, sortable, log	arguments passed to cog
target	value to be used for the target attribute of the a html tag - default is "_blank" which will open the link in a new window

Value

a hash string

See Also

validateState, cogHref

Examples

```
## Not run:
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# divide housing data by county
byCounty <- divide(housingData::housing, by = c("county", "state"))

xlim <- as.Date(c("2008-01-31", "2016-01-31"))

# plot list price vs. time for each county
makeDisplay(byCounty, name = "county_time",
  panelFn = function(x)
```

```
    ggplot(x, aes(time, medListPriceSqft)) +
      geom_point() + xlim(xlim))

# divide housing data by state
byState <- divide(housingData::housing, by = "state")

# create a "displayHref" cognostic that links to the by county display
# filtered down to all counties in the current state
cogFn <- function(x) {
  state <- stateSpec(
    name = "county_time",
    sort = list(county = "asc"),
    layout = list(nrow = 2, ncol = 4),
    filter = list(state = list(select = getSplitVar(x, "state"))))

  list(countyPlots = cogDisplayHref(state = state, defLabel = TRUE))
}

# plot distribution of list price vs. time for each state
makeDisplay(byState, name = "state_time_CI",
  panelFn = function(x)
    ggplot(x, aes(time, medListPriceSqft)) +
      stat_summary(fun.data = "mean_cl_boot") + xlim(xlim),
  cogFn = cogFn)

# open up the state display
# try clicking on the link for "countyPlots"
# the by county display will be loaded filtered to the state
view("state_time_CI")

## End(Not run)
```

---

cogHref *Href Cognostic*

Description

Create href to be used as cognostics in a trelliscope display.

Usage

```
cogHref(x, label = "link", desc = "link", group = "common",
  target = "_blank", defLabel = FALSE, defActive = FALSE,
  filterable = FALSE, sortable = TRUE, log = FALSE)
```

Arguments

x	URL to link to

label	label of the href
desc, group, defLabel, defActive, filterable, sortable, log	arguments passed to cog
target	value to be used for the target attribute of the a html tag - default is "_blank" which will open the link in a new window

See Also

cog

Examples

```
d <- divide(iris, by = "Species")

# cognostics function that links to wikipedia
cogFn <- function(x) {
  link <- paste0("https://en.wikipedia.org/wiki/Iris_", getSplitVar(x, "Species"))
  list(wiki = cogHref(link, desc = "Look up species on wikipedia", defLabel = TRUE))
}

# test the cognostics function on a subset
applyCogFn(cogFn, d[[1]])

# make a display with this cognostics function
vdbConn(tempfile(), autoYes = TRUE)
makeDisplay(d, name = "iris_sl_sw",
  cogFn = cogFn, panelFn = function(x)
    lattice::xyplot(Sepal.Length ~ Sepal.Width, data = x))

## Not run:
# clicking the link under each panel will open wikipedia
view(name = "iris_sl_sw")

## End(Not run)
```

---

cogLoessRMSE *Compute RMSE of Loess Fit Cognostic*

Description

Compute RMSE of loess fit as a cognostic to be used in a trelliscope display.

Usage

```
cogLoessRMSE(..., desc = "RMSE of residuals from loess fit",
  group = "common", defLabel = FALSE, defActive = TRUE,
  filterable = TRUE, sortable = TRUE, log = FALSE)
```

Arguments

`desc, group, defLabel, defActive, filterable, sortable, log`	arguments passed to cog
`...`	arguments to be passed to `link{loess}`, such as the formula, data, smoothing parameters, etc.

See Also

cog

Examples

```
d <- stack(data.frame(EuStockMarkets))
d$time <- rep(as.numeric(time(EuStockMarkets)), 4)
d$year <- floor(d$time)

byIndexYear <- divide(d, by = c("ind", "year"))

cogFn <- function(x)
  list(lormse = cogLoessRMSE(values ~ time, data = x, span = 0.3, degree = 2),
       slope  = cogSlope(values ~ time, data = x),
       range  = cogRange(x$values),
       mean   = cogMean(x$values),
       max    = cog(max(x$values, na.rm = TRUE), desc = "max value"))

applyCogFn(cogFn, byIndexYear[[1]])

library(lattice)
panelFn <- function(x)
  xyplot(values ~ time, data = x,
    panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.loess(x, y, span = 0.3,
        degree = 2, evaluation = 200, col = "black")
    })

vdbConn(tempfile(), autoYes = TRUE)
makeDisplay(byIndexYear, name = "ts_index_year",
  cogFn = cogFn, panelFn = panelFn)

## Not run:
# sort and fiter the index/year panels by slope and loess RMSE
view(name = "ts_index_year")

## End(Not run)
```

cogMean *Compute Mean Cognostic*

Description

Compute mean to be used as cognostics in a trelliscope display.

Usage

```
cogMean(x, desc = "mean", group = "common", defLabel = FALSE,
  defActive = TRUE, filterable = TRUE, sortable = TRUE, log = FALSE)
```

Arguments

x	numeric vector from which to compute the mean
desc, group, defLabel, defActive, filterable, sortable, log	arguments passed to cog

See Also

cog

Examples

```
d <- stack(data.frame(EuStockMarkets))
d$time <- rep(as.numeric(time(EuStockMarkets)), 4)
d$year <- floor(d$time)

byIndexYear <- divide(d, by = c("ind", "year"))

cogFn <- function(x)
  list(lormse = cogLoessRMSE(values ~ time, data = x, span = 0.3, degree = 2),
       slope  = cogSlope(values ~ time, data = x),
       range  = cogRange(x$values),
       mean   = cogMean(x$values),
       max    = cog(max(x$values, na.rm = TRUE), desc = "max value"))

applyCogFn(cogFn, byIndexYear[[1]])

library(lattice)
panelFn <- function(x)
  xyplot(values ~ time, data = x,
    panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.loess(x, y, span = 0.3,
        degree = 2, evaluation = 200, col = "black")
    })

vdbConn(tempfile(), autoYes = TRUE)
```

```
makeDisplay(byIndexYear, name = "ts_index_year",
  cogFn = cogFn, panelFn = panelFn)

## Not run:
# sort and fiter the index/year panels by slope and loess RMSE
view(name = "ts_index_year")

## End(Not run)
```

---

cogNcol *Methods for Cognostics Connections*

Description

Methods for Cognostics Connections

Usage

```
cogNcol(x, ...)

cogNrow(x, ...)

cogNames(x, ...)

getCogData(x, rowIdx, colIdx, ...)
```

Arguments

x	cognostics connection object
...	other objects passed onto generic methods
rowIdx	index of rows to be retrieved from the cognostics connection
colIdx	index of columns to be retrieved from the cognostics connection

Note

These methods are used mainly by the trelliscope viewer and therefore must be exported. Their purpose is to provide a general interface for a cognostics store. Currently just data frames are used for cognostics, but in previous versions systems like MongoDB were used. These methods should never need to be used by an analyst.

See Also

makeDisplay

Examples

```
# see examples for makeDisplay()
```

cogPre *Methods Used in MapReduce for makeDisplay*

Description

Methods Used in MapReduce for makeDisplay

Methods Used in MapReduce for makeDisplay

Methods Used in MapReduce for makeDisplay

Methods Used in MapReduce for makeDisplay

Usage

```
cogPre(cogConn, ...)

cogEmit(cogConn, ...)

cogCollect(cogConn, ...)

cogFinal(cogConn, ...)
```

Arguments

cogConn	a cognostics connection object
...	additional parameters

Examples

```
# used internally when calling makeDisplay
```

cogRange *Compute Range Cognostic*

Description

Compute range to be used as cognostics in a trelliscope display.

Usage

```
cogRange(x, desc = "range (max - min)", group = "common",
  defLabel = FALSE, defActive = TRUE, filterable = TRUE,
  sortable = TRUE, log = FALSE)
```

Arguments

x	numeric vector from which to compute the range
desc, group, defLabel, defActive, filterable, sortable, log	arguments passed to cog

See Also

cog

Examples

```
d <- stack(data.frame(EuStockMarkets))
d$time <- rep(as.numeric(time(EuStockMarkets)), 4)
d$year <- floor(d$time)

byIndexYear <- divide(d, by = c("ind", "year"))

cogFn <- function(x)
  list(lormse = cogLoessRMSE(values ~ time, data = x, span = 0.3, degree = 2),
       slope  = cogSlope(values ~ time, data = x),
       range  = cogRange(x$values),
       mean   = cogMean(x$values),
       max    = cog(max(x$values, na.rm = TRUE), desc = "max value"))

applyCogFn(cogFn, byIndexYear[[1]])

library(lattice)
panelFn <- function(x)
  xyplot(values ~ time, data = x,
    panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.loess(x, y, span = 0.3,
        degree = 2, evaluation = 200, col = "black")
    })

vdbConn(tempfile(), autoYes = TRUE)
makeDisplay(byIndexYear, name = "ts_index_year",
  cogFn = cogFn, panelFn = panelFn)

## Not run:
# sort and fiter the index/year panels by slope and loess RMSE
view(name = "ts_index_year")

## End(Not run)
```

---

cogScagnostics *Compute Scagnostics*

Description

Compute list of scagnostics (see scagnostics) to be used as cognostics in a trelliscope display.

Usage

```
cogScagnostics(x, y, group = "scagnostics", defLabel = FALSE,
  defActive = TRUE, filterable = TRUE, sortable = TRUE, log = FALSE)
```

Arguments

x	vector of the x-axis data for a scatterplot
y	vector of the y-axis data for a scatterplot
group, defLabel, defActive, filterable, sortable, log	arguments passed to cog

See Also

cog

Examples

```
## Not run:
cogScagnostics(iris$Sepal.Length, iris$Sepal.Width)

## End(Not run)
```

---

cogSlope *Compute Slope of Linear Fit Cognostic*

Description

Compute the slope of a linear fit as a cognostic to be used in a trelliscope display.

Usage

```
cogSlope(..., desc = "Slope of fitted line", group = "common",
  defLabel = FALSE, defActive = TRUE, filterable = TRUE,
  sortable = TRUE, log = FALSE)
```

Arguments

desc, group, defLabel, defActive, filterable, sortable, log	arguments passed to cog
...	arguments to be passed to link{loess}, such as the formula, data, smoothing parameters, etc.

See Also

cog

Examples

```
d <- stack(data.frame(EuStockMarkets))
d$time <- rep(as.numeric(time(EuStockMarkets)), 4)
d$year <- floor(d$time)

byIndexYear <- divide(d, by = c("ind", "year"))

cogFn <- function(x)
  list(lormse = cogLoessRMSE(values ~ time, data = x, span = 0.3, degree = 2),
       slope  = cogSlope(values ~ time, data = x),
       range  = cogRange(x$values),
       mean   = cogMean(x$values),
       max    = cog(max(x$values, na.rm = TRUE), desc = "max value"))

applyCogFn(cogFn, byIndexYear[[1]])

library(lattice)
panelFn <- function(x)
  xyplot(values ~ time, data = x,
    panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.loess(x, y, span = 0.3,
        degree = 2, evaluation = 200, col = "black")
    })

vdbConn(tempfile(), autoYes = TRUE)
makeDisplay(byIndexYear, name = "ts_index_year",
  cogFn = cogFn, panelFn = panelFn)

## Not run:
# sort and fiter the index/year panels by slope and loess RMSE
view(name = "ts_index_year")

## End(Not run)
```

---

deployVDB *Deploy VDB to shinyapps.io or RStudio Connect*

Description

Deploy VDB to shinyapps.io or RStudio Connect

Usage

```
deployVDB(vdbConn = getOption("vdbConn"), appName = NULL, account = NULL,
  redeploy = TRUE, size = NULL, instances = NULL, quiet = FALSE)
```

Arguments

vdbConn	A vdbConn object containing the VDB connection settings
appName	name of application (app will be available at https://[account].shinyapps.io/[appName]/ or at https://beta.rstudioconnect.com/[account]/[appName] if using RStudio connect) - if not supplied, will use the name of VDB connection
account	passed to `rsconnect::configureApp`
redeploy	passed to `rsconnect::configureApp`
size	passed to `rsconnect::configureApp`
instances	passed to `rsconnect::configureApp`
quiet	passed to `rsconnect::configureApp`

Details

If you do not have a shinyapps.io account and have not set your account info, first visit here prior to calling this function: http://shiny.rstudio.com/articles/shinyapps.html.

syncLocalData

Examples

```
## Not run:

library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# make a simple display
d <- divide(iris, by = "Species")
makeDisplay(d, name = "sl_vs_sw",
  panelFn = function(x)
    qplot(Sepal.Width, Sepal.Length, data = x))

# add additional displays...

# assuming an account has already been configured with shinyapps.io
# or RStudio Connect
deployVDB(appName = "deployVDB-example")

## End(Not run)
```

---

dfCogConn *Initiate Data Frame Cognostics Connection*

Description

Initiate data frame cognostics connection

Usage

```
dfCogConn()
```

Value

"cogConn" object of class "dfCogConn"

Note

This should never need to be called explicitly. It is the default mechanism for storing cognostics in makeDisplay.

See Also

makeDisplay

Examples

```
# see examples for makeDisplay()
```

---

encodePNG *base64 Encoding of a .png File*

Description

base64 Encoding of a .png File

Usage

```
encodePNG(plotLoc)
```

Arguments

plotLoc	location of a png file on disk to encode as a base64 string

Examples

```
f <- tempfile(fileext = ".png")
png(f)
plot(1:10)
dev.off()
encodePNG(f)
```

getDisplay *Retrieve Display Object from VDB*

Description

Retrieve a display object from a VDB.

Usage

```
getDisplay(name, group = NULL, conn = getOption("vdbConn"))
```

Arguments

name	the name of the display
group	the group of the display
conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists

Details

If a display is uniquely determined by its name, then group is not required.

Value

a display object

See Also

Other display_manipulation: cleanupDisplays, listDisplays, removeDisplay, restoreDisplay, updateDisplay

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)
```

```
vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)

# look at a list of the displays in this vdb:
listDisplays()

# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

`getVdbPath` *Get Current VDB Path*

Description

Get Current VDB Path

Usage

```
getVdbPath()
```

Examples

```
vdbConn(tempfile(), autoYes = TRUE)
getVdbPath()
```

`listDisplays` *List Displays in a VDB*

Description

List displays in a VDB.

Usage

```
listDisplays(conn = getOption("vdbConn"))
```

Arguments

conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists

See Also

Other display_manipulation: `cleanupDisplays`, `getDisplay`, `removeDisplay`, `restoreDisplay`, `updateDisplay`

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)

vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)

# look at a list of the displays in this vdb:
listDisplays()
```

```
# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

`makeDisplay` *Create a Trelliscope Display*

Description

Create a trelliscope display and add it to a visualization database (VDB)

Usage

```
makeDisplay(data, name, group = "common", desc = "", mdDesc = NULL,
  height = 500, width = 500, panelFn = NULL, lims = list(x = "free", y =
  "free", prepanelFn = NULL), cogFn = NULL, state = NULL,
  preRender = FALSE, thumbIndex = 1, cogConn = dfCogConn(),
  output = NULL, conn = getOption("vdbConn"), verbose = TRUE,
  keySig = NULL, params = NULL, packages = NULL, control = NULL,
  detectGlobals = TRUE)
```

Arguments

data	data of class "ddo" or "ddf" (see ddo, ddf)
name	the name of the display (no special characters, spaces are converted to underscores)
group	the group the display belongs to, where displays are organized into groups (no special characters, spaces are converted to underscores). Defaults to "common"
desc	a description of the display (used in the viewer)
mdDesc	an optional longer-form description of the display and data, which can be text or can be a path to a markdown file or file with html snippets. The description will appear in the "Display Information" panel in the Trelliscope viewer.
height	reference dimensions (in pixels) for each panel (panels will be resized based on available space in the viewer)
width	reference dimensions (in pixels) for each panel (panels will be resized based on available space in the viewer)
panelFn	a function that produces a plot and takes one argument, which will be the current split of the data being passed to it. It is recommended that you first test panelFn on a single key-value pair using panelFn(data[[1]][[2]]). This function must return either an object of class "ggplot", "trellis", or return "NULL" (for base plot commands)
lims	either an object of class "trsLims" as obtained from setLims or a list with elements x, y, and prepanelFn, that specify how to apply prepanel and setLims
cogFn	a function that returns a named list, where each element of the list is a cognostic feature (with length 1). This list must be coerceable to a 1-row data frame. The function should take one argument, which will be the current split of the data being passed to it. Useful to test with cogFn(divExample(dat))
state	if specified, this tells the viewer the default parameter settings (such as layout, sorting, filtering, etc.) to use when the display is viewed (see validateState for details)
preRender	should the panels be pre-rendered and stored (TRUE), or rendered on-the-fly (FALSE, default)? Default is recommended unless rendering is very expensive. See Details.
thumbIndex	the index value to use for creating the thumbnail
cogConn	a connection to store the cognostics data. By default, this is dfCogConn().
output	how to store the panels and metadata for the display (unnecessary to specify in most cases – see details)
conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
verbose	print status messages?
keySig	a user-defined key signature (string - see details)
params	a named list of objects external to the input data that are needed in the distributed computing (most should be taken care of automatically such that this is rarely necessary to specify)

packages	a vector of R package names that contain functions used in panelFn or cogFn (most should be taken care of automatically such that this is rarely necessary to specify)
control	parameters specifying how the backend should handle things (most-likely parameters to rhwatch in RHIPE) - see rhipeControl and localDiskControl
detectGlobals	if TRUE params are automatically detected (packages are always auto-detected)

Details

Many of the parameters are optional or have defaults. For several examples, see the documentation at deltarho.org: http://deltarho.org/docs-trelliscope

Panels by default are not pre-rendered. Instead, this function creates a display object and computes and stores the cognostics. Panels are then rendered on the fly by the DeltaRho backend and pushed to the Trelliscope viewer as html with the panel images embedded in the html. If a user would like to pre-render the images for every subset (using preRender = TRUE), then by default the image files for the panels will be stored to a local disk connection (see localDiskConn) inside the VDB directory, organized in subdirectories by group and name of the display. Optionally, the user can specify the output parameter to be any valid "kvConnection" object, as long as it is one that persists on disk (e.g. hdfsConn).

keySig does not generally need to be specified. It is useful to specify when creating multiple displays that you would like to be treated as related displays, so that you can view them side by side. Two displays are determined to be related when their key signatures, typically computed as a md5 hash of the complete collection of keys, match. Sometimes two displays will have data where the keys match for a significant portion of subsets, but not all. Manually specifying the same keySig for each can ensure that they will be treated as related displays.

See Also

prepanel, setLims, divide

Examples

```
## Not run:
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# divide housing data by county
byCounty <- divide(housingData::housing, by = c("county", "state"))

xlim <- as.Date(c("2008-01-31", "2016-01-31"))

# plot list price vs. time for each county
makeDisplay(byCounty, name = "county_time",
  panelFn = function(x)
    ggplot(x, aes(time, medListPriceSqft)) +
      geom_point() + xlim(xlim))

# divide housing data by state
```

```
byState <- divide(housingData::housing, by = "state")

# create a "displayHref" cognostic that links to the by county display
# filtered down to all counties in the current state
cogFn <- function(x) {
  state <- stateSpec(
    name = "county_time",
    sort = list(county = "asc"),
    layout = list(nrow = 2, ncol = 4),
    filter = list(state = list(select = getSplitVar(x, "state"))))

  list(countyPlots = cogDisplayHref(state = state, defLabel = TRUE))
}

# plot distribution of list price vs. time for each state
makeDisplay(byState, name = "state_time_CI",
  panelFn = function(x)
    ggplot(x, aes(time, medListPriceSqft)) +
      stat_summary(fun.data = "mean_cl_boot") + xlim(xlim),
  cogFn = cogFn)

# open up the state display
# try clicking on the link for "countyPlots"
# the by county display will be loaded filtered to the state
view("state_time_CI")

## End(Not run)
```

---

makePNG *Make a png for a Subset*

Description

Make a png for a subset of an object of class "ddo" or "ddf". The user should never need to call this directly, but the viewer needs it, so it is exported with this documentation.

Usage

```
makePNG(dat, panelFn = NULL, file, width, height, origWidth = width,
  res = 72, basePointSize = 12, lims = NULL, pixelratio = 2)
```

Arguments

dat	a key-value pair
panelFn	panel function
file	file name for png
width, height, res	width, height, and resolution

origWidth	the original specified width of the plot
basePointSize	the base point size to use (for png pointsize argument)
lims	axis limits
pixelratio	pixel ratio of screen to which plot will be rendered (e.g. 2 for a retina display)

See Also

makeDisplay

Examples

```
# see examples for makeDisplay()
```

---

makeSplodData *Create Data Plottable by splod*

Description

Create pairwise scatterplot data plottable by splod

Usage

```
makeSplodData(data, id.vars = NULL)
```

Arguments

data	a data.frame
id.vars	variables to ignore when computing all pairs of variables

Value

an object of class 'localDiv' and 'splodDat' that can be passed to splod

References

Wilkinson, L., Anushka A., and Grossman, R. L. "Graph-Theoretic Scagnostics." INFOVIS. Vol. 5. 2005.

See Also

splod, splodPanelFn

Examples

```
## Not run:
library(lattice)

vdbConn(tempfile(), autoYes = TRUE)

# make "splod" directly from a data frame
splod(batting, name = "batting",
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))

# first transform the data into a "splodDat" object
batSplodDat <- makeSplodData(batting,
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))
# now make "splod"
splod(batSplodDat, name = "batting2", data = batSplodDat)

# custom panel function (color by league)
mySplodFn <- function(d) {
  xyplot(jitter(y) ~ jitter(x), groups = lgID, data = d,
    xlab = getSplitVar(d, "xVar"),
    ylab = getSplitVar(d, "yVar"),
    auto.key = TRUE
  )
}

splod(batSplodDat, name = "batting3",
  data = batSplodDat, panelFn = mySplodFn)

view()

## End(Not run)
```

`makeStateHash` *Make a URL hash out of state information*

Description

Make a URL hash out of cognostics state information

Usage

```
makeStateHash(x)
```

Arguments

`x`	a list of cognostics state parameters

Value

a URL hash

See Also

validateState

Examples

```
state <- stateSpec(
  name = "my_display",
  sort = list(state = "desc", county = "asc"),
  filter = list(
    county = list(regex = "Ben"),
    state = list(select = c("OR", "WA")),
    meanList = list(from = 50, to = 150)
  ),
  layout = list(nrow = 2, ncol = 4),
  labels = c("county", "state")
)

state <- validateState(state, checkDisplay = FALSE)

makeStateHash(state)
```

---

noMargins *Remove Margins from Trellis/Lattice Plot*

Description

Removes whitespace surrounding a trellis plot. Ideal for makeDisplay because when tiling multiple panels, a lot of space is wasted with the margins.

Usage

```
noMargins(..., topkey = FALSE, rightkey = FALSE)
```

Arguments

topkey	should extra space be added for a top key?
rightkey	should extra space be added for a right key?
...	a list of other parameters to be passed to par.settings

See Also

xyplot, makeDisplay

Examples

```
## Not run:
library(lattice)
xyplot(c(1:10) ~ c(1:10) | sample(letters[1:2], 10, replace=TRUE),
  par.settings=noMargins()
)

# with additional par.settings...
xyplot(c(1:10) ~ c(1:10) | sample(letters[1:2], 10, replace = TRUE),
  par.settings = noMargins(
    list(plot.symbol = list(col = "black"))
  )
)

## End(Not run)
```

---

phantomInstall *Get instructions on how to install phantomjs*

Description

Get instructions on how to install phantomjs

Usage

```
phantomInstall()
```

Examples

```
phantomInstall()
```

---

plot.trsPre *Plot results form prepanel*

Description

Plot results form prepanel

Usage

```
## S3 method for class 'trsPre'
plot(x, layout = c(2, 2), as.table = TRUE, strip = FALSE,
  strip.left = TRUE, between = list(y = 0.25), xlab = "Rank",
  ylab = "Panel Limits", ...)
```

Arguments

x	object of class "trsPre" created by prepanel
layout, as.table, strip, strip.left, between, xlab, ylab, ...	parameters for the lattice plot that is output (these are defaults - can ignore unless you want fine control)

Details

This function plots the sorted axis ranges for the x and y axis for the case of "same" (all axis limits share the same range) and "sliced" (all axis limits share the) and can be useful in helping determine how to ultimately set the limits.

Value

object of class "trellis" (plotted by default)

See Also

prepanel, makeDisplay

Examples

```
## Not run:
irisSplit <- datadr::divide(iris, "Species")
irisPreFn <- function(x) {
  list(
    xlim = range(x$Sepal.Length),
    ylim = range(x$Sepal.Width)
  )
}
irisPre <- prepanel(irisSplit, prepanelFn = irisPreFn)
plot(irisPre)

## End(Not run)

d <- datadr::divide(iris, "Species")

irisPreFn <- function(x) {
  list(
    xlim = range(x$Sepal.Length),
    ylim = range(x$Sepal.Width)
  )
}

irisPre <- prepanel(d, prepanelFn = irisPreFn)

plot(irisPre)

irisLims <- setLims(irisPre, x = "same", y = "sliced")
```

`prepanel` *Prepanel Function for Trelliscope Displays*

Description

Apply a prepanel function to objects of class "ddo" or "ddf" to determine ranges of x and y axis limits prior to creating a trelliscope display (`makeDisplay`). Useful in conjunction with `setLims`.

Usage

```
prepanel(data, prepanelFn = NULL, params = NULL, packages = NULL,
  control = NULL, verbose = TRUE)
```

Arguments

`data`	an object of class "localDiv" or "rhData"
`prepanelFn`	a prepanel function that returns a list specifying `xlim` and `ylim` for determining axis limits, and optionally `dx` and `dy` for determining aspect ratio (used to define slopes of line segments used for banking computations). prepanelFn can also be a panelFn (see `makeDisplay`) that returns either an object of class "trellis" or "ggplot", since xlim and ylim can be determined from these.
`params`	a named list of parameters external to the input data that are needed in the distributed computing (most should be taken care of automatically such that this is rarely necessary to specify)
`packages`	a vector of R package names that contain functions used in `prepanelFn` (most should be taken care of automatically such that this is rarely necessary to specify)
`control`	parameters specifying how the backend should handle things (most-likely parameters to `rhwatch` in RHIPE) - see `rhipeControl` and `localDiskControl`
`verbose`	print status messages?

Details

The plot method plots the sorted axis ranges for the x and y axis for the case of "same" (all axis limits share the same range) and "sliced" (all axis limits share the) and can be useful in helping determine how to ultimately set the limits.

You do not need to use `prepanel()` to ultimately create a display with `makeDisplay()`, but if you bypass, you will either need to specify your own limits in your plot command, or do nothing, in which case each individual plot will have limits based on the data in the split being plotted (the axes will be "free").

Axis limits are very important. What makes viewing groups of plots of subsets of data ("small multiples") so powerful is being able to make meaningful visual comparisons across plots. This is much easier to do if scales for each plot are commensurate.

This function is also useful for identifying subsets with very large outlying values, and in conjunction with `setLims`, allows you to account for that prior to the expensive process of creating all of the plots.

Value

object of class "trsPre". This is a list of the x and y axis ranges for each split, along with the aspect ratio banking value if dx and dy are supplied in `prepanelFn`. Can be used with `plot.trsPre` and `setLims`.

See Also

`x plot.trsPre`, `setLims`, `makeDisplay`

Examples

```
d <- datadr::divide(iris, "Species")

irisPreFn <- function(x) {
  list(
    xlim = range(x$Sepal.Length),
    ylim = range(x$Sepal.Width)
  )
}

irisPre <- prepanel(d, prepanelFn = irisPreFn)

plot(irisPre)

irisLims <- setLims(irisPre, x = "same", y = "sliced")
```

`print.cog` *Print a cognostics object*

Description

Print a cognostics object

Usage

```
## S3 method for class 'cog'
print(x, ...)
```

Arguments

`x`	a cognostics object
`...`	further arguments passed to or from other methods

print.dfCogConn *Print a dfCogConn object*

Description

Print a dfCogConn object

Usage

```
## S3 method for class 'dfCogConn'
print(x, ...)
```

Arguments

x	a "dfCogConn" object
...	further arguments passed to or from other methods

print.displayObj *Print a display object*

Description

Print a display object

Usage

```
## S3 method for class 'displayObj'
print(x, ...)
```

Arguments

x	an object of class "displayObj"
...	further arguments passed to or from other methods

print.qtrellis *Print a qtrellis Object*

Description

Print a qtrellis Object

Usage

```
## S3 method for class 'qtrellis'
print(x, ...)
```

Arguments

x	a "qtrellis" object
...	further arguments passed to or from other methods

print.vdbConn *Print a vdbConn Object*

Description

Print a vdbConn Object

Usage

```
## S3 method for class 'vdbConn'
print(x, ...)
```

Arguments

x	a "vdbConn" object
...	further arguments passed to or from other methods

Examples

```
conn <- vdbConn(tempfile(), name = "myvdb", autoYes = TRUE)
conn
```

qtrellis *Quick trelliscope display for data frame-like inputs*

Description

Quick trelliscope display for data frame-like inputs

Usage

```
qtrellis(x, panel = NULL, cog = NULL, by = NULL, layout = c(1, 1),
  conn = getOption("vdbConn"), ...)
```

Arguments

x	either a data frame
panel	a function taking one argument (which will be a subset of the input data frame) and returning a plot
cog	an optional cognostics funtion to be applied to each subset
by	if the input is a data frame, a character vector of column names to split the data by
layout	a vector indicating the number of rows and columns to arrange the panels in by default
conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists or if you would like to use a temporary one for this display
...	parameters passed to makeDisplay - most importantly name, group (see note below), width, and height

Note

If you don't have a vdb connection set up (see vdbConn), a temporary one will be created and used, and you can think of the plot as disposable. If you would like the plot to persist, set up a vdb connection. Likewise, if you don't supply name and group, the plot will be stored under defaults "qtrellis_plot" and "__qtrellis", and a subsequent call will cause the previous display with this name and group to be replaced. Therefore, if you want your display to persist, make sure a vdb connection has been set up prior to calling this function, and give it a unique name.

Examples

```
## Not run:
panel <- function(x)
  xyplot(Sepal.Width ~ Sepal.Length, data = x)

p <- datadr::divide(iris, by = "Species") %>%
  qtrellis(panel, layout = c(1, 3))
p
```

```
# data frame input (need to specify 'by')
iris %>% qtrellis(panel, by = "Species")

# dplyr grouped tbl input
library(dplyr)
p <- iris %>%
  group_by(Species) %>%
  qtrellis(panel, layout = c(1, 3))
p

## End(Not run)
```

`removeDisplay` *Remove a Display from a VDB*

Description

Remove a display from a VDB.

Usage

```
removeDisplay(name = NULL, group = NULL, conn = getOption("vdbConn"),
  autoYes = FALSE, verbose = TRUE)
```

Arguments

`name`	the name of the display
`group`	the group of the display
`conn`	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
`autoYes`	should questions to proceed with display removal be automatically answered with "yes"?
`verbose`	logical - print messages about what is being done

Details

If a display is uniquely determined by its name, then group is not required.

See Also

Other display_manipulation: `cleanupDisplays`, `getDisplay`, `listDisplays`, `restoreDisplay`, `updateDisplay`

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)

vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)

# look at a list of the displays in this vdb:
listDisplays()

# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

restoreDisplay *Restore a Backed-Up Display Object*

Description

Restore a Backed-Up Display Object

Usage

```
restoreDisplay(name, group = NULL, conn = getOption("vdbConn"),
  autoYes = FALSE)
```

Arguments

name	the name of the display
group	the group the display belongs to
conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
autoYes	should questions to proceed with display removal be automatically answered with "yes"?

See Also

Other display_manipulation: cleanupDisplays, getDisplay, listDisplays, removeDisplay, updateDisplay

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)

vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)
```

```
# look at a list of the displays in this vdb:
listDisplays()

# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

`setLims` *Specify Rules for x and y Limits for a Display*

Description

Based on results from `prepanel`, specify rules that will determine x and y axis limits to be passed as the `lims` argument when calling `makeDisplay`.

Usage

```
setLims(lims, x = "same", y = "same", xQuant = c(0, 1), yQuant = c(0,
  1), xRangeQuant = 1, yRangeQuant = 1, prop = 0.07)
```

Arguments

`lims`	object of class "trsPre"
`x`	x-axis limits rule (either "same", "sliced", or "free" - see details)

y	y-axis limits rule (either "same", "sliced", or "free" - see details)
xQuant	lower and upper quantiles at which to cut off x-axis limits, in the case of outliers. Used when x="same".
yQuant	same as xQuant but for y-axis
xRangeQuant	a single upper quantile at which to cut off the x-axis range, used when x="sliced", used in the case of a few splits having abnormally high range, which are wished to be excluded
yRangeQuant	same as xRangeQuant but for y-axis
prop	the proportion of the axis range to pad beyond the actual axis range

Details

This function reduces the list of axis limits computed for each split of a data set to an overall axis limit rule for the plot.

About "x" and "y" parameters: This is the same as in lattice. From lattice documentation: A character string that determines how axis limits are calculated for each panel. Possible values are "same" (default), "free" and "sliced". For relation="same", the same limits, usually large enough to encompass all the data, are used for all the panels. For relation="free", limits for each panel is determined by just the points in that panel. Behavior for relation="sliced" is similar, except that the length (max - min) of the scales are constrained to remain the same across panels.

Value

object of class "trsLims", which can be used in a call to makeDisplay

See Also

prepanel, makeDisplay

Examples

```
## Not run:
irisSplit <- datadr::divide(iris, "Species")
irisPreFn <- function(x) {
  list(
    xlim = range(x$Sepal.Length),
    ylim = range(x$Sepal.Width)
  )
}
irisPre <- prepanel(irisSplit, prepanelFn = irisPreFn)
irisLims <- setLims(irisPre, x = "same", y = "sliced")

## End(Not run)

d <- datadr::divide(iris, "Species")

irisPreFn <- function(x) {
  list(
    xlim = range(x$Sepal.Length),
```

```
    ylim = range(x$Sepal.Width)
  )
}

irisPre <- prepanel(d, prepanelFn = irisPreFn)

plot(irisPre)

irisLims <- setLims(irisPre, x = "same", y = "sliced")
```

---

splod *Create a Scatterplot Display*

Description

Create a scatterplot display (splod)

Usage

```
splod(data, id.vars = NULL, name = NULL, desc = NULL,
  cogFn = splodCogFn, panelFn = splodPanelFn, verbose = TRUE, ...)
```

Arguments

`data`	a data.frame or an object of class "splodDat"
`id.vars`	variables to ignore when computing all pairs of variables
`name, desc, cogFn, panelFn, verbose, ...`	parameters passed to `makeDisplay`

Value

an object of class 'localDiv' that can be passed to `splod`

References

Wilkinson, L., Anushka A., and Grossman, R. L. "Graph-Theoretic Scagnostics." INFOVIS. Vol. 5. 2005.

See Also

`makeDisplay`, `makeSplodData`, `splodPanelFn`

Examples

```
## Not run:
library(lattice)

vdbConn(tempfile(), autoYes = TRUE)

# make "splod" directly from a data frame
splod(batting, name = "batting",
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))

# first transform the data into a "splodDat" object
batSplodDat <- makeSplodData(batting,
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))
# now make "splod"
splod(batSplodDat, name = "batting2", data = batSplodDat)

# custom panel function (color by league)
mySplodFn <- function(d) {
  xyplot(jitter(y) ~ jitter(x), groups = lgID, data = d,
    xlab = getSplitVar(d, "xVar"),
    ylab = getSplitVar(d, "yVar"),
    auto.key = TRUE
  )
}

splod(batSplodDat, name = "batting3",
  data = batSplodDat, panelFn = mySplodFn)

view()

## End(Not run)
```

---

splodCogFn *Default Cognostics Function for splod*

Description

Default cognostics function for splod

Usage

```
splodCogFn(df)
```

Arguments

df	a subset of data created by makeSplodData

Value

a data.frame of scagnostics for the given subset

References

Wilkinson, L., Anushka A., and Grossman, R. L. "Graph-Theoretic Scagnostics." INFOVIS. Vol. 5. 2005.

See Also

splodPanelFn, splod, makeSplodData

Examples

```
## Not run:
library(lattice)

vdbConn(tempfile(), autoYes = TRUE)

# make "splod" directly from a data frame
splod(batting, name = "batting",
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))

# first transform the data into a "splodDat" object
batSplodDat <- makeSplodData(batting,
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))
# now make "splod"
splod(batSplodDat, name = "batting2", data = batSplodDat)

# custom panel function (color by league)
mySplodFn <- function(d) {
  xyplot(jitter(y) ~ jitter(x), groups = lgID, data = d,
    xlab = getSplitVar(d, "xVar"),
    ylab = getSplitVar(d, "yVar"),
    auto.key = TRUE
  )
}

splod(batSplodDat, name = "batting3",
  data = batSplodDat, panelFn = mySplodFn)

view()

## End(Not run)
```

---

splodPanelFn *Default Plot Function for splod*

Description

Default plot function for splod

Usage

```
splodPanelFn(df)
```

Arguments

df	a subset of data created by makeSplodData

Value

a trellis plot object of a scatterplot for the given subset

References

Wilkinson, L., Anushka A., and Grossman, R. L. "Graph-Theoretic Scagnostics." INFOVIS. Vol. 5. 2005.

See Also

splodCogFn, splod, makeSplodData

Examples

```
## Not run:
library(lattice)

vdbConn(tempfile(), autoYes = TRUE)

# make "splod" directly from a data frame
splod(batting, name = "batting",
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))

# first transform the data into a "splodDat" object
batSplodDat <- makeSplodData(batting,
  id.vars = c("playerID", "yearID", "stint", "teamID", "lgID"))
# now make "splod"
splod(batSplodDat, name = "batting2", data = batSplodDat)

# custom panel function (color by league)
mySplodFn <- function(d) {
  xyplot(jitter(y) ~ jitter(x), groups = lgID, data = d,
    xlab = getSplitVar(d, "xVar"),
    ylab = getSplitVar(d, "yVar"),
    auto.key = TRUE
  )
}

splod(batSplodDat, name = "batting3",
  data = batSplodDat, panelFn = mySplodFn)

view()

## End(Not run)
```

stateSpec *Set State Parameters*

Description

Set State Parameters

Usage

```
stateSpec(name = NULL, group = "common", labels = NULL, layout = NULL,
  sort = NULL, filter = NULL)
```

Arguments

name	the name of the display
group	the group of the display
labels	a vector of names of cognostics to be shown as labels underneath each panel. If not specified, the default is to show labels for any of the splitting variables that created the partition of the data being plotted.
layout	a list with optional elements nrow, ncol, and arrange. nrow and ncol specify the arrangement of the panels into rows and columns (nrow = 1 and ncol = 1 are defaults), and arrange can be either "row" or "col" and specified whether to sort the panels by row or by column ("row" is default)
sort	a named list where each name corresponds to a cognostic name and the value is either "asc" or "desc" for sorting in ascending or descending order. The order in which sorting is applied to each variable is according to the order of the variables specified.
filter	a named list where each name corresponds to a cognostic name and the value is a specification of either "regex" or "select" for categorical variables, or a range, "from" and "to", for quantitative variables. For a "regex", a simple regular expression string is specified, and the filter finds all matches for the regular expression against the variable. For "select" a vector of strings is specified, and all exact matches are returned. For the range filter, all values of the specified variable within the range "from" and "to" are returned. If either "from" or "to" are omitted, they are treated as -Inf and Inf respectively.

Details

Trelliscope allows you to specify either a default state in makeDisplay or specify the state of the display when you call view.

Examples

```
state <- stateSpec(
  name = "my_display",
  sort = list(state = "desc", county = "asc"),
```

```
  filter = list(
    county = list(regex = "Ben"),
    state = list(select = c("OR", "WA")),
    meanList = list(from = 50, to = 150)
  ),
  layout = list(nrow = 2, ncol = 4),
  labels = c("county", "state")
)

state <- validateState(state, checkDisplay = FALSE)

makeStateHash(state)
```

---

syncLocalData *Sync localDisk objects to VDB*

Description

Sync localDisk data that is used for VDB displays located throughout the system to a 'data' directory inside the VDB - useful for collecting data before syncing with a web server, and used inside of webSync and deployVDB.

Usage

```
syncLocalData(vdbConn = getOption("vdbConn"), rsync = NULL)
```

Arguments

vdbConn	VDB connection settings
rsync	location of rsync binary

See Also

webSync, webConn

Examples

```
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# divide iris data and store it as a local disk connection
d <- divide(iris, by = "Species",
  output = localDiskConn(tempfile(), autoYes = TRUE))

# make a simple display using this data
makeDisplay(d, name = "sl_vs_sw",
  panelFn = function(x)
    qplot(Sepal.Width, Sepal.Length, data = x))
```

```
# look at files in our VDB directory
list.files(getVdbPath())
# since the data for our display resides in a different path
# if we try to share this file with someone on another machine
# the data will not be found

# sync any local data objects used in any display to be
# centralized with the VDB directory, making it portable
## Not run:
syncLocalData() # requires rsync

## End(Not run)

# now there is a "data" directory that holds all local disk data
list.files(getVdbPath())
```

---

toHash *Methods for dealing with state and hashes*

Description

Methods for dealing with state and hashes

Usage

```
toHash(x)

fromHash(x)
```

Arguments

x	state or hash object

Note

Used in makeStateHash.

Examples

```
state <- stateSpec(
  name = "my_display",
  sort = list(state = "desc", county = "asc"),
  filter = list(
    county = list(regex = "Ben"),
    state = list(select = c("OR", "WA")),
    meanList = list(from = 50, to = 150)
  ),
  layout = list(nrow = 2, ncol = 4),
  labels = c("county", "state")
```

```
)

state <- validateState(state, checkDisplay = FALSE)

makeStateHash(state)
```

`updateDisplay` *Update a Display Object*

Description

Update a Display Object

Usage

```
updateDisplay(name, ..., group = NULL, conn = getOption("vdbConn"))
```

Arguments

`name`	the name of the display
`group`	the group the display belongs to
`conn`	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
`...`	display parameters to update which must be one of "desc", "width", "height", "keySig", "panelFn", "state" - see `makeDisplay` for details on these parameters.

See Also

Other display_manipulation: `cleanupDisplays`, `getDisplay`, `listDisplays`, `removeDisplay`, `restoreDisplay`

Examples

```
library(lattice)
library(ggplot2)

d <- divide(iris, by = "Species")

# two panel functions
p1 <- function(x)
  xyplot(Sepal.Length ~ Sepal.Width, data = x)
p2 <- function(x)
  qplot(Sepal.Width, Sepal.Length, data = x)

# try them both out on a subset
p1(d[[1]]$value)
p2(d[[1]]$value)
```

```
vdbConn(tempfile(), autoYes = TRUE)

makeDisplay(d, name = "lattice", panelFn = p1)
makeDisplay(d, name = "ggplot2", panelFn = p2)

# look at a list of the displays in this vdb:
listDisplays()

# get a the lattice display object
lobj <- getDisplay("lattice")
# look at one of the fields of the display object
lobj$desc

# we forgot to provide a description in makeDisplay
# let's add one without recreating it
updateDisplay("lattice", desc = "lattice plot of sepal width vs. length")
# let's see if it was updated
lobj <- getDisplay("lattice")
lobj$desc

# overwrite one of the displays
makeDisplay(d, name = "ggplot2", panelFn = function(x) qplot(1, 1))

# notice that there is a backup display:
list.files(file.path(getVdbPath(), "displays", "common"))

# oops - let's restore the previous one
restoreDisplay(name = "ggplot2", autoYes = TRUE)

# suppose we want to get rid of the lattice display
removeDisplay("lattice", autoYes = TRUE)

# check the list of displays
listDisplays()
```

`validateState` *Validate State Parameters*

Description

Validate state parameters for a Trelliscope display

Usage

```
validateState(x, displayObj = NULL, checkDisplay = TRUE)
```

Arguments

x	a list of state parameter settings (such as layout, sorting, filtering, etc.) to use when the display is viewed (see details)
displayObj	a display object to validate against (if not provided and checkDisplay is TRUE, it will be fetched based on x)
checkDisplay	should the state be checked against a display (to make sure fields match, etc.) or should it simply be checked for structure?

Value

a modified state parameter list that is valid, an object of class "cogState"

Note

See stateSpec for details on how to specify state.

See Also

view, cogDisplayHref

Examples

```
state <- stateSpec(
  name = "my_display",
  sort = list(state = "desc", county = "asc"),
  filter = list(
    county = list(regex = "Ben"),
    state = list(select = c("OR", "WA")),
    meanList = list(from = 50, to = 150)
  ),
  layout = list(nrow = 2, ncol = 4),
  labels = c("county", "state")
)

state <- validateState(state, checkDisplay = FALSE)

makeStateHash(state)
```

---

vdbConn *Connect to a VDB*

Description

Connect to a new or existing visualization database

Usage

```
vdbConn(path, name = NULL, autoYes = FALSE, updateFiles = TRUE,
  verbose = TRUE)
```

Arguments

path	the path on the local file system where the directory for the VDB is located
name	a character string giving the name of the VDB. If the VDB already exists and name = NULL, the previous name is used. If the VDB exists and a string is provided for name, the name is overwritten. The primary purpose of the name argument is to facilitate deploying the trelliscope display as a Shiny app. See the appName argument of deployVDB
autoYes	should questions to proceed with directory creation operations be automatically answered with "yes"?
updateFiles	upon connection, should the Trelliscope viewer app files be updated in the VDB directory?
verbose	should messages be printed about what is being done?

Details

Connecting to a VDB is required prior to calling makeDisplay or view.

Value

An object of class vdbConn that contains the path and name of the VDB. This object is also assigned to the vdbConn option, and can be retrieved via getOption("vdbConn")

Examples

```
conn <- vdbConn(tempfile(), name = "myvdb", autoYes = TRUE)
conn
```

---

vdbConvert *Convert a VDB to be usable with the new Trelliscope viewer (experimental)*

Description

Convert a VDB to be usable with the new Trelliscope viewer (experimental)

Usage

```
vdbConvert(overwrite = FALSE, basePath = NULL, convertPanels = TRUE,
  jsonp = TRUE, conn = getOption("vdbConn"), open = TRUE,
  autoYes = FALSE)
```

Arguments

overwrite	should existing converted files be overwritten? (not implemented)
basePath	the base directory to place the converted vdb in (doesn't need to exist)
convertPanels	should panels be converted to json for the new viewer? (good to set to FALSE if this has already been done but other aspects of the VDB have changed and need to be re-converted)
jsonp	should jsonp files be created instead of json?
conn	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
open	should the new viewer be opened after conversion?
autoYes	should questions to proceed with directory creation operations be automatically answered with "yes"?

Details

This is an experimental function that allows experimentation with the next generation Trelliscope viewer (https://github.com/hafen/trelliscopejs).

---

vdbCopyRSource *Copy files ending in .R from source directory into a VDB-wide global code directory for use in Trelliscope displays*

Description

These R files will be sourced in the global environment of the Trelliscope Viewer when the viewer is launched and the resulting objects will be available to all displays throughout the viewing session. Useful for getting custom functions into the global environment.

Usage

```
vdbCopyRSource(fromDir, conn = getOption("vdbConn"))
```

Arguments

fromDir	directory with R source
conn	vdb connection

Author(s)

Jeremiah Rounds

Examples

```
## Not run:
vdbCopyRSource(".") # copies R files from current directory

## End(Not run)
```

vdbGlobalsExist *Check to see if the VDB-wide global data file exists*

Description

Check to see if the VDB-wide global data file exists

Usage

```
vdbGlobalsExist(conn = getOption("vdbConn"))
```

Arguments

conn	vdb connection

Value

logical

Author(s)

Jeremiah Rounds

Examples

```
vdbGlobalsExist()
```

vdbGlobalsFile *Path to VDB global data storage file*

Description

Returns an appropriate file name to save VDB-wide globals to for use in Trelliscope displays

Usage

```
vdbGlobalsFile(conn = getOption("vdbConn"))
```

Arguments

conn	vdb connection

Details

Objects in this rdata file will be loaded by the Trelliscope Viewer when the viewer is launched and the data will be available to all displays throughout the viewing session.

Value

character file name

Author(s)

Jeremiah Rounds

Examples

```
## Not run:
save(foo, file = globalsPath())

## End(Not run)
```

`view` *View a Display or Run Shiny Display Viewer*

Description

View a display or run Shiny display viewer

Usage

```
view(name = NULL, group = NULL, state = NULL, openBrowser = TRUE,
  conn = getOption("vdbConn"), port = getOption("trelliscopePort"),
  copyFiles = TRUE)
```

Arguments

`name, group`	optional parameters to load the viewer with a pre-specified display - if not specified, the viewer will launch with a list to choose from
`state`	an optional list of state variables to set the default viewing state for layout, sorting, filtering, and labels (see details)
`openBrowser`	should the browser be automatically launched?
`conn`	VDB connection info, typically stored in options("vdbConn") at the beginning of a session, and not necessary to specify here if a valid "vdbConn" object exists
`port`	what port to use for the viewer - if not specified, will look for "trelliscopePort" set in R's global options, followed by a search for a system-level environment variable "TRELLISCOPE_PORT". If none of these are defined, a random port assigned provided by shiny will be used.
`copyFiles`	should updated viewer files be copied over to VDB directory?

Examples

```
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# make a simple display
d <- divide(iris, by = "Species")
makeDisplay(d, name = "sl_vs_sw",
  panelFn = function(x)
    qplot(Sepal.Width, Sepal.Length, data = x))

## Not run:
# open viewer with default providing list of displays to veiw
view()

# open viewer on port 8100
view(port = 8100)

# make port 8100 the default port for all future calls to view()
options(trelliscopePort = 8100)

# open viewer directly to "sl_vs_sw"
view(name = "sl_vs_sw")

# open viewer directly to "sl_vs_sw" and set intial state
view(state = stateSpec(
  name = "sl_vs_sw",
  layout = list(nrow = 1, ncol = 3),
  labels = c("panelKey", "Species")))

## End(Not run)
```

---

webConn *Initialize a Web Connection*

Description

Initialize a connection to a web server where Shiny apps are served.

Usage

```
webConn(user = NULL, ip = NULL, serverDir = "/srv/shiny-server",
  name = NULL)
```

Arguments

user	the username to log on to the web server
ip	the IP address of the web server - if NULL, it is assumed that your web server is on the same machine that you are working on

serverDir	the directory where Shiny apps go on the web server - defaults to the default location of /srv/shiny-server
name	the name of the directory in serverDir under which to store the application - if not supplied, it defaults to the name provided in the vdb connection

See Also

webSync

Examples

```
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# make a simple display
d <- divide(iris, by = "Species")
makeDisplay(d, name = "sl_vs_sw",
  panelFn = function(x)
    qplot(Sepal.Width, Sepal.Length, data = x))

## Not run:
# to sync to a server 'myshinyserver.org' with login 'user'
# need: passwordless ssh for user@myshinyserver.org)
# need: rsync installed on local machine
# (these should both be easy to do with local linux / OS X)

# set up a connection to a shiny server
webConn(user = "hafen", ip = "myshinyserver.org", name = "myapp")

# webSync() uses rsync to sync your local vdb
# to the one on your shiny server pointed to with webConn()
webSync()

# if shiny server is running on the remote on port 3838
# then the VDB will now be viewable at
browseURL("http://myshinyserver.org:3838/myapp")

## End(Not run)
```

---

webSync *Sync VDB files to a web server*

Description

Sync VDB files to a web server

Usage

```
webSync(vdbConn = getOption("vdbConn"), webConn = getOption("vdbWebConn"),
  fixPermissions = FALSE, verbose = FALSE, rsync = NULL)
```

Arguments

vdbConn	VDB connection settings
webConn	web connection settings
fixPermissions	should an attempt be made to fix permissions in the web directory?
verbose	show rsync output
rsync	location of rsync binary

Details

This requires rsync to be installed on your machine. If you are syncing via ssh, this only works if public key authentication is enabled between your local machine and the remote server.

See Also

webConn, syncLocalData

Examples

```
library(ggplot2)

vdbConn(tempfile(), autoYes = TRUE)

# make a simple display
d <- divide(iris, by = "Species")
makeDisplay(d, name = "sl_vs_sw",
  panelFn = function(x)
    qplot(Sepal.Width, Sepal.Length, data = x))

## Not run:
# to sync to a server 'myshinyserver.org' with login 'user'
# need: passwordless ssh for user@myshinyserver.org)
# need: rsync installed on local machine
# (these should both be easy to do with local linux / OS X)

# set up a connection to a shiny server
webConn(user = "hafen", ip = "myshinyserver.org", name = "myapp")

# webSync() uses rsync to sync your local vdb
# to the one on your shiny server pointed to with webConn()
webSync()

# if shiny server is running on the remote on port 3838
# then the VDB will now be viewable at
browseURL("http://myshinyserver.org:3838/myapp")
```

```
## End(Not run)
```

`widgetThumbnail` *Make a thumbnail for an htmlwidget panel*

Description

Make a thumbnail for an htmlwidget panel

Usage

```
widgetThumbnail(p, thumbPath, timeout = 1500)
```

Arguments

`p`	htmlwidget object
`thumbPath`	where to save thumbnail file
`timeout`	how many milliseconds to wait until plot is rendered

Note

This is used internally `makeDisplay` to create thumbnails of htmlwidget panel functions.

Index