Package ‘playwith’

February 20, 2015

Type Package
Title A GUI for interactive plots using GTK+
Version 0.9-54
Date 2012-08-28
Author Felix Andrews <felix@nfrac.org>
Maintainer Felix Andrews <felix@nfrac.org>
Depends R (>= 2.7.0), lattice (>= 0.16-3), cairoDevice, gWidgetsRGtk2
(>= 0.0-45), grid
Imports RGtk2 (>= 2.20.1), gWidgets (>= 0.0-30), gridBase, grDevices,
graphics, stats, utils
Suggests latticist, zoo, MASS, ggplot2, sp
Description A GTK+ graphical user interface for editing and
interacting with R plots.
License GPL (>= 2)
URL http://playwith.googlecode.com/
Repository CRAN
Date/Publication 2012-08-28 11:57:12
NeedsCompilation yes

R topics documented:

autoplay .......................................................... 2
callArg .......................................................... 3
convertFromDevicePixels ...................................... 5
identifyGrob ..................................................... 7
panel.usertext ................................................... 8
parameterControlTool ......................................... 9
playDo ........................................................... 10
playGetIDs ....................................................... 11
playPointInput .................................................. 13
playSelectData .................................................. 14
autoplay

Set playwith to run automatically.

Description

Set playwith to run automatically with Lattice graphics and/or base graphics.

Usage

autoplay(on = NA, lattice.on = on, base.on = on, grid.on = on, ask = FALSE)

Arguments

on TRUE to set playwith to run automatically to display Lattice and base graphics. FALSE to revert to the default plot device (i.e. getOption("device")).

lattice.on run automatically to display Lattice graphics.

base.on run automatically to display base graphics (plot etc).

grid.on run automatically to display grid graphics (except Lattice).

ask if TRUE, select the appropriate plot call from a list (the call stack) when automatically starting a base graphics plot.

Details

When lattice.on is TRUE, the print.trellis function, which is typically called implicitly to create lattice plots, will trigger playwith, passing the original high-level call. So for lattice plots only, this is like changing your default plot device. It only replaces the screen device: plotting to a file device will work as normal. This feature requires lattice package version 0.17-1 or later.

When base.on is TRUE, any new base graphics plot will trigger playwith (via a hook in plot.new), and the high-level plot call is taken to be the first call (to a named function) on the call stack. The usual base graphics paradigm of building up a plot incrementally will not work well, because only the initial plot call is recorded, so any further additions will be lost when the plot is redrawn. For similar reasons, multiple-figure plots may not be redrawn correctly. The high-level plot will be called twice initially, due to constraints of the mechanism.

The grid.on argument is analogous to base.on for grid graphics, using a hook in grid.newpage.
Note that this automatic behaviour is not a full replacement for calling `playwith` directly, since it does not allow you to specify any of the optional arguments.

Another possibility is `options(device="playwith")`. This will act as a default device but without most of the interactive features. The plot can still be annotated with text and arrows in this case.

**Author(s)**

Felix Andrews <felix@nfrac.org>

**See Also**

`playwith`

**Examples**

```r
if (interactive()) {

## lattice graphics in the playwith interface:
autoplay(lattice=T)
xyplot(Sepal.Length ~ Sepal.Width | Species, data=iris)

dev.off()

## lattice graphics in your usual screen device:
autoplay(lattice=F)
xyplot(Sepal.Length ~ Sepal.Width | Species, data=iris)

## base graphics in the playwith interface:
autoplay(TRUE)
frog <- rnorm(64)
hist(frog)

dev.off()

## base graphics in your usual screen device:
autoplay(FALSE)
hist(frog)

}
```

---

**callArg**

*Work with plot call arguments*

**Description**

Part of the `playwith` Application Programming Interface.
Usage

callArg(playState, arg, eval = TRUE, data = NULL)
callArg(playState, arg) <- value

mainCall(playState)
mainCall(playState) <- value

updateMainCall(playState)

Arguments

playState  a playState object representing the plot, window and device.
arg        the argument name or number in the main plot call. This can also be a language object (e.g. quote(scales$log)). Argument numbers start from 1 (so 0 refers to the main function name). This is evaluated in the calling environment, so can refer to local variables (e.g. data[[mynname]]).
eval       whether to evaluate the argument before returning it. Otherwise, just return the argument as it appears in the call.
data       a list or environment in which to evaluate the argument. Typically this will be the "data" argument to lattice or qplot functions.
value      the value to assign.

Details

These functions get and set argument values in the playState plot call.

As convenience functions for setting arguments in playState$call, the callArg function helps by:

• referring to the main call that accepts plot arguments, which is not necessarily the top-level call.
• automatically evaluating variables that have been stored in a local environment (playState$env).
• converting lists to language objects as needed.
• enforcing exact matching of argument names exact=TRUE (see [[]).

Value

returns the value of the specified argument, possibly evaluated in a local environment (playState$env).

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

playwith.API
Examples

if (interactive()) {

library(lattice)
playwith(xyplot(1:10 ~ 1:10))
playState <- playDevCur()

callArg(playState, "pch") ## NULL
callArg(playState, "pch") <- "$
callArg(playState, "pch") ## "$'
playReplot(playState)

## referring to local variables
tmp <- "x"
callArg(playState, quote(scales[[tmp]]$cex)) <- 2
playReplot(playState)
}

convertFromDevicePixels

Utilities for working with grobs and viewports in device coordinates.

Description

Utilities for working with grobs and viewports in device coordinates.

Usage

convertFromDevicePixels(x.px, y.px, unitTo = "native", valueOnly = FALSE)
convertToDevicePixels(x, y)

inViewport(x.px, y.px, viewport)
grobBBDevicePixels(grob, viewport, pad = 2)

grobBoundingBoxes(draw = TRUE,
    gp.box = gpar(col = "yellow", lwd = 5, alpha = 0.2),
    gp.text = gpar(cex = 0.75, alpha = 0.5))

Arguments

x.px, y.px locations in device coordinates (i.e. pixels, with origin at top-left corner of device). May be vectors.
unitTo the unit to convert to.
valueOnly to return values as numeric (native) rather than units.
x, y locations in the current viewport (numeric native or units). May be vectors.
viewport a viewport name or vpPath.
grob
    a grob.
pad
    number of pixels to expand bounding boxes.
draw
    whether to draw bounding boxes and grob names.
gp.box
    graphical parameters for bounding boxes.
gp.text
    graphical parameters for grob name text.

Details
    Not yet...

Value
    convertFromDevicePixels returns a list with \( x, y \) (units or numeric) locations in the current viewport.
    convertToDevicePixels returns a list with \( x, y \) (numeric) locations in pixels from the top-left corner of the device.
    inViewport returns a logical: whether the given pixel location is inside the given viewport.
    grobBBDevicePixels returns a list with \( x, y \) (numeric) locations in pixels giving the bounding box of the given grob. The grob must exist in the given viewport.
    grobBoundingBoxes returns a data.frame giving information about all grobs in the current scene.

Author(s)
    Felix Andrews <felix@nfrac.org>

See Also
    grid.convert, grobX, grid.ls, grid.show.layout

Examples
    print(xyplot(1:10 ~ 1:10 | c("a", "b")))
    vpname <- trellis vpname("panel", 1, 1)
    downViewport(vpname)
    convertToDevicePixels(x = 5:10, y = 5:10)
    convertToDevicePixels(unit(0, "npc"), unit(0, "npc"))
    convertFromDevicePixels(x = 100, y = 100)
    inViewport(x = 100, y = 100, vpname)
    inViewport(x = c(0, 100), y = c(0, 100), vpname)
    myGrob <- grid.circle(r = 0.3, name = "myCircle")
    grobBBDevicePixels(myGrob, vpname)
    str(grobBoundingBoxes(draw = TRUE))
identifyGrob

return names of clicked grid-objects.

Description

This identifies which grobs (grid objects) bounding boxes were clicked on (or otherwise identified by pixel coordinates), and returns their names. The names can be used by other grid functions, like grid.edit or grid.remove.

Usage

identifyGrob(xy.pixels = grid.locator(), classes = NULL)

Arguments

xy.pixels if given, a list with components x and y, being pixel coordinates in the current plot device.

classes if given, one or more class names of grobs. This can be used to filter the results to only specified types of objects. Examples: "text", "lines", "segments", "rect".

Details

None yet.

Value

a character vector of grob names, or NULL.

Note

This will give an error if the current plot has no grobs, as returned by grid.ls.

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

grobBoundingBoxes

Examples

if (interactive()) {
  xyplot(1:10 ~ 1:10)
  identifyGrob()
  str(grid.get(identifyGrob()[1]))
  grid.remove(identifyGrob(class = "points"))[1])
Description

These are versions of panel.text and panel.points with different (customisable) style settings.

Usage

```r
panel.usertext(x, y = NULL, labels = seq_along(x), col,
alpha, cex, srt = 0, lineheight, font,
fontfamily, fontface, adj = c(0.5, 0.5),
pos = NULL, offset = 0.5, ...)
panel.brushpoints(x, y = NULL, col, pch, alpha,
fill, cex, ...)
panel.brushlines(x, y = NULL, type, col, alpha,
lty, lwd, ...)
```

Arguments

- `x, y` text or point locations.
- `labels, col, alpha, cex, srt, lineheight` see the usual lattice functions in `llines`.
- `font, fontfamily, fontface, adj, pos, offset` as above.
- `pch, fill, type, lty, lwd, ...` as above.

Details

The settings for `panel.usertext` are taken from `trellis.par.get("user.text")`, but default to `trellis.par.get("add.text")` if undefined. "user.text" is preferred to "add.text" for annotations because the latter also applies to strip text and key text.

The settings for `panel.brushpoints` are taken from `trellis.par.get("brush.symbol")`, but default to hard-coded settings if undefined.

The settings for `panel.brushlines` are taken from `trellis.par.get("brush.line")`, but default to hard-coded settings if undefined.

These functions are used in `playwith`.

Author(s)

Felix Andrews <felix@nfrac.org>
parameterControlTool

See Also

l1lines, trellis.par.get

Examples

```r
xyplot(1:10 ~ 1:10, panel = function(...) {
    panel.brushpoints(...)
    panel.usertext(..., pos = 1)
})
```

Description

Part of the playwith Application Programming Interface.

Usage

```r
## Note: this is only to be called inside a tool constructor function.
parameterControlTool(playState, name, value, label = name,
    handler = NULL, horizontal = TRUE)
```

Arguments

- **playState**: a playState object, as passed in to the constructor function.
- **name**: the parameter name as it appears in the plot call.
- **value**: the possible or starting values for the parameter. Can be a single or vector value, of integer, numeric, character or logical. See playwith for details.
- **label**: label for the widget.
- **handler**: a function.
- **horizontal**: FALSE to make a tool for a vertical toolbar.

Details

Note: this is only to be called inside a tool constructor function. It is not intended to be called directly.

Value

- a gtkToolButton.

Author(s)

Felix Andrews <felix@nfrac.org>
See Also

playwith

Examples

### none yet

---

**playDo**

*Do something in a specified plot space*

---

**Description**

Part of the **playwith** Application Programming Interface.

**Usage**

```r
playDo(playState, expr, space = "plot",
       clip.off = !isTRUE(playState$clip.annotations),
       return.code = FALSE)
```

**Arguments**

- `playState` a **playState** object representing the plot, window and device.
- `expr` an expression, typically a drawing operation, to evaluate in the plot space. It will be quoted (see examples).
- `space` the plot space (viewport) to go to before evaluating `expr`. This can be "plot", "page", or for lattice plots "packet N" (where N is a packet number), or for grid plots the name of a viewport passed to the `viewport` argument of `playwith`.
- `clip.off` enforce no clipping of drawing operations: see clip argument to `viewport`.
- `return.code` if TRUE, return code (as an expression) for the given actions, rather than executing them.

**Details**

This function allows an arbitrary expression to be evaluated while some part of the plot has been made active (i.e. moving to a grid graphics viewport). Grid viewports are used also to represent spaces in a base graphics plot (using the gridBase package). That means `expr` can always use Grid drawing operations.

The default `space="plot"` will work for base graphics, grid graphics and for single-panel lattice plots. It will also work for multi-panel lattice plots when one panel is in focus (see `trellis.focus`). Using `space="page"` will apply to the whole device space in normalised device coordinates (0–1).

Other functions such as **playSelectData** and **playPointInput** return values that can be used directly for the `space` argument.
Value

the value returned by expr.

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

playwith.API, playSelectData, playPointInput

Examples

```r
if (interactive()) {
  library(lattice)
  library(grid)

  packs <- paste("packet", rep(1:4, each=4))
  playwith(xypolygon(1:16 ~ 1:16 | packs))
  myGp <- gpar(fill="red", col="black", alpha=0.5)

  ## draw in a specific packet
  playDo(playDevCur(), grid.circle(gp=myGp), space="packet 2")

  ## draw in default space="plot" after focussing on one panel
  trellis.focus("panel", 1, 1)
  packet.number() # 1, same as space="packet 1"
  playDo(playDevCur(), grid.circle(gp=myGp))
  trellis.unfocus()

  ## space="plot" does not work in a multi-panel plot
  ## unless one panel is in focus
  try(playDo(playDevCur(), grid.circle(gp=myGp)))

  ## draw on the whole page
  playDo(playDevCur(), grid.circle(gp=myGp), space="page")
}
```

---

**playGetIDs**

Get or set IDs of selected points

**Description**

Part of the **play** Application Programming Interface.
**Usage**

```r
playGetIDs(playState = playDevCur(),
            type = c("labelled", "brushed"),
            labels = FALSE)
```

```r
playSetIDs(playState = playDevCur(), value,
           type = "brushed", space = "plot",
           add = FALSE, redraw = NA, pos = 1)
```

```r
playClear(playState = playDevCur(),
           type = c("annotations", "labelled", "brushed"),
           redraw = TRUE)
```

**Arguments**

- `playState`: a `playState` object representing the plot, window and device.
- `type`: which type of points to get or set subscripts for.
- `labels`: TRUE to return the labels as displayed; otherwise the subscripts into the original data object.
- `value`: specifies the set of points to be selected. Can be an integer vector of subscripts, or a logical vector (not recycled).
- `space`: the space to draw labels in if `type = "ids"`.
- `add`: if TRUE, add to any existing set of points; otherwise replace them.
- `redraw`: whether to redraw the plot. The default NA only redraws if necessary (i.e. if an existing selection was replaced). If FALSE nothing is drawn.
- `pos`: position specifier for labels.

**Details**

Not yet.

**Value**

`playGetIDs` returns an integer vector, or if `labels = TRUE` a character vector.

**Author(s)**

Felix Andrews <felix@nfrac.org>

**See Also**

`playwith.API`
Examples

```r
if (interactive()) {

  playwith(xyplot(1:100 - 1:100 | 1:2, subscripts = TRUE),
           labels = paste("label", 1:100, sep=""))
  playSetIDs(value = c(50:60, 100))
  playGetIDs()
  playSetIDs(value = c(10, 20, 30), type = "labelled",
              space = "packet 1")
  playGetIDs(labels = TRUE)
  playClear()
  playGetIDs()

}
```

---

**playPointInput**

Get point, line or rect input from playwith user

---

**Description**

Part of the **playwith** Application Programming Interface.

**Usage**

```r
playPointInput(playState = playDevCur(), prompt)
playLineInput(playState = playDevCur(), prompt, scales = "dynamic")
playRectInput(playState = playDevCur(), prompt, scales = "dynamic")
```

**Arguments**

- **playState** a **playState** object representing the plot, window and device.
- **prompt** text to display in the prompt.
- **scales** the default allows the user to hold Shift while dragging, to constrain the selection to x or y scales. Otherwise this should be one or more of "x" and "y", specifying which axes to select along.

**Details**

**playPointInput** is similar to **locator**, but returns native coordinates in whichever plot space was clicked on. Device coordinates and normalised device coordinates are also available.

**playRectInput** and **playLineInput** allow the user to interactively draw a rectangle or line.
Value

All these functions return NULL if the user cancelled (e.g. by right-clicking). Otherwise a list with:

- **space** character, specifies the plot space in which the user clicked or dragged. See the space argument to **playDo**.
- **coords** native coordinates of the point or shape in space. A list with numeric vectors x and y. For a rectangle or line, these are length 2 where the first element refers to the start of the drag motion. For a point these are length 1. If space is "page", this is NULL.
- **dc** device coordinates of the point or shape (pixels).
- **ndc** normalised device coordinates of the point or shape.
- **is.click** logical, whether it was a click (so probably should not be treated as a rectangle or line). This is not returned by **playPointInput**.
- **modifiers** a flag representing which modifier keys were pressed during the click (or at the end of a drag). E.g. to test for Shift key: if (foo$modifiers & GdkModifierType["shift-mask"]).

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

**playSelectData, playwith.API**

Examples

```r
if (interactive()) {

library(lattice)
playwith(xyplot(Sepal.Width ~ Petal.Width | Species, data = iris))
playPointInput()
playRectInput()
}
```

---

**playSelectData**  
*Let playwith user select data points*

Description

Part of the **playwith** Application Programming Interface.
Usage

```r
playSelectData(playState = playDevCur(),
   prompt = paste("Click or drag to select data points; ",
                      "Right-click or Esc to cancel.");,
   scales = "dynamic",
   multiview = TRUE,
   foo = playRectInput(playState, prompt = prompt, scales = scales))
```

Arguments

- `playState`: a `playState` object representing the plot, window and device.
- `prompt`: text to display in the prompt.
- `scales`: the default allows the user to hold Shift while dragging, to constrain the selection to x or y scales. Otherwise this should be one or more of "x" and "y", specifying which axes to select along.
- `multiview`: to return a matrix of x and y when the plot contains multiple positions for the same points (e.g. splom).
- `foo`: rectangular plot region structure, usually generated interactively.

Details

`playSelectData` is similar to `identify`. The user is prompted to click or drag to select data points. If a click, the nearest point is selected, if it is within 11 points. If it is a drag, all points within the rectangular region are selected. Note that data points can be selected from one panel of a multi-panel lattice plot without specifying the panel beforehand.

Value

`playSelectData` returns `NULL` if the user cancelled (e.g. by right-clicking). Otherwise a list with:

- `space`: character, specifies the plot space in which data points were selected. See the `space` argument to `playDo`.
- `subscripts`: indices of the data points selected. This relies on the data being guessed correctly from the plot call, unless `data_points` was specified explicitly.
- `x, y`: values of the selected data points.
- `is.click`: logical, whether it was a click (otherwise a drag).
- `pos`: position of click relative to the closest point, as in the `pos` argument to `text`. If `is.click` is false, this is `NULL`.
- `...`: as well as all the other elements returned by `playRectInput`.

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

`playRectInput`, `playwith.API`
Examples

```r
if (interactive()) {

  library(lattice)
  playwith(xyplot(Sepal.Width ~ Petal.Width | Species, data = iris))
  playSelectData()
}
```

**playState**

Object representing the plot, window and device.

Description

The `playState` object is created by `playwith` to represent the state of the plot, window and device. It is central to the `playwith.API`.

Details

A `playState` object is an `environment` (of class "playState") containing many other objects, including:

- `win` the plot window (`gtkWindow`).
- `dev` the plot device, as returned by `dev.cur`.
- `call` the current plot call.
- `env` local environment used to store plot data objects.
- `accepts.arguments` whether the current main plot function accepts arguments.
- `callName` name of the current main plot function.
- `is.lattice, is.ggplot, is.vcd, is.base` whether the current plot is a Lattice / ggplot / base graphics plot. `is.base` is defined as TRUE if neither of the others is TRUE and `viewport` is undefined.
- `result, trellis` result is the result of evaluating the plot call. If `is.lattice`, the trellis object is also stored in `trellis`.
- `viewport` a named list of viewport paths (or names). One of these must be named "plot". NULL if the plot is a base graphics or Lattice plot.
- `spaces` a character vector listing all spaces defined in the current plot, not including "page".
- `labels` labels for data points, either given or guessed.
- `data.points` given data points or NULL.
- `ids` a named list of currently labelled data points. Each name corresponds to a "space", which can be "page" (positioned on page) or "plot" (positioned in plot coordinates). With Lattice graphics the space can be "packet 1" etc. Each list element is a data frame of numeric subscripts and `pos` (for label placement).
- `annotations` a named list of calls to be evaluated in a target viewport; each name corresponds to a "space", as in `ids`.
linked an environment, containing a list "ids" and a list "subscribers". Elements of the former are subscripts of brushed data points. Elements of the latter are the playState objects of all linked plots in this group. This linked object is shared by all subscribers.

tools hmm...
uiManager, actionGroups the GtkUIManager and a named list of its action groups.

widgets A list of the GUI widgets. The most useful one is drawingArea (the plot device widget).
tmp a list of temporary objects, including:
   plot.ready whether the plot has been drawn and is ready for interaction.
   identify.ok whether data points and labels are defined (or a plausible guess could be made), allowing points to be identified.

There are several other standard objects which can be inspected with ls.str(playDevCur()). Other objects can be passed in through the ... argument to playwith, or defined by tools.

Author(s)
Felix Andrews <felix@nfrac.org>

See Also
playwith, playwith.API

---

Description
A GTK+ graphical user interface for exploring and editing R plots.

Usage

```r
playwith(expr, 
  new = playwith.getOption("new"),
  title = NULL,
  labels = NULL,
  data.points = NULL,
  viewport = NULL,
  parameters = list(),
  tools = list(),
  init.actions = list(),
  preplot.actions = list(),
  update.actions = list(),
  ..., 
  width = playwith.getOption("width"),
  height = playwith.getOption("height"),
  pointsize = playwith.getOption("pointsize"),
```
eval.args = playwith.getOption("eval.args"),
on.close = playwith.getOption("on.close"),
modal = FALSE,
link.to = NULL,
playState = if (!new) playDevCur(),
plot.call,
main.function)

Arguments

expr  an expression to create a plot, like plot(mydata). Note, arguments and nested
calls are allowed, just like a normal plot call (see examples). Could also be a
chunk of code in {braces}. For quoted calls, use the plot.call argument.

new  if TRUE open in a new window, otherwise replace the current window (if one
exists).

title  optional window title; otherwise derived from the plot call.

labels  a character vector of labels for data points. If missing, it will be guessed from
the plot call arguments if possible.

data.points  a data frame (or other suitable plotting structure: see xy.coords) giving loca-
tions of data points, in case these can not be guessed from the plot call argu-
ments. If a data frame, extra variables may be included; these can be used to
label or locate points in the GUI. Note, if a suitable data argument is found in
the plot call, that plays the same role.

viewport  name or vpPath of the viewport representing the data space. This allows inter-
action with grid graphics plots (but ignore this for Lattice plots). Experimental:
can also be a named list.

parameters  defines simple tools for controlling values of any parameters appearing in the
plot call. This must be a named list, where the value given for each name defines
the possible or initial values of that parameter. The supported values are:

• integer or AsIs(I()): creates a numeric spinbutton.
• numeric scalar: creates a text entry box for numeric values.
• numeric vector: creates a slider with given range.
• character: creates a text entry box.
• character vector: creates a combo box (including text entry).
• logical: creates a checkbox.
• function: creates a button, which calls the given function with a single
argument playState.

These can also be lists, where the first item is the value as above. In this case
an item named label can specify a label for the widget, and an item named
handler can specify a function to run when the widget is changed. This func-
tion should be a function(playState, value); the parameter values are then
accessed from playState$env. If the function returns FALSE the plot is not
redrawn.

tools  a list of tool specifications. These are technically GtkActionEntries but should
be specified as lists with the following structure. Elements can be specified in
this order, or named (as with a function call).
name  The name of the action (used internally to control the action state, or in a
custom UI XML file). This item is required and must be the first element.
All other elements are optional.

stock_id  The stock icon ID, or the name of an icon from the icon theme. See
unlist(gtkStockListIds()) or http://library.gnome.org/devel/gtk/
unstable/gtk-Stock-Items.html for a list.

label  The label for the action. If label is NULL, the default label for the given
stock.id is used.

accelerator  The accelerator for the action, in the format understood by gtkAcceleratorParse.
See gdkKeySyms.

tooltip  The tooltip for the action.

callback  The function to call when the action is activated.

is_active  Only for toggle actions: sets the initial state (TRUE / FALSE).

update.action, init.action  If present these items must be named. Their
values are included in the update.actions and init.actions lists.

preplot.actions, update.actions
a list of actions to be run, respectively, before and after the plot is drawn (and
each time it is redrawn). Note that preplot.actions can not assume that
playState$s.is.lattice (or other state values) are set. They can, however,
modify the plot call or its data before the plot is drawn.
These may be functions, or names of functions, or expressions. Functions are
passed one argument, which is the playState. Note, these are in addition to
any given in playwith.options("update.actions").

init.actions
init.actions are run whenever the plot type changes or its data changes. They
are not run when only simple arguments to the call change, but they are run
whenever the plot call is edited manually. Same format as update.actions.

... extra arguments are stored in the playState object. These can then be accessed
by tools. The default tools will recognise the following extra arguments:

click.mode  sets the initial action when clicking and dragging on the plot: one
of "Zoom", "Identify", "Brush", "Annotation", or "Arrow".

time.mode  whether the plot is to start in "time mode", with navigation along the
x-axis. If NA, it will guess whether to start in time.mode based on whether
the current plot looks like a time series plot (but this can chew some extra
memory). The default is taken from playwith.options("time.mode").

time.vector  a vector defining discrete times, as numeric, Date or POSIXt.
It must be sorted, increasing. If given, then the "time mode" is used to
navigate along these discrete times, rather than along the continuous x-
axis. Special objects cur.index and cur.time will be provided in the
plot environment, so the plot call can refer to these. cur.index is the cur-
tent time step, between 1 and length(time.vector), and cur.time is
time.vector[cur.index]. In this case time.mode will be on by default.

cur.index, cur.time, time.mode, page.incr  If time.vector is given, either
of cur.index or cur.time will set the initial time step. time.mode, page.incr
sets the number of steps to jump if the user clicks on the scroll bar.
	page  In multi-page Lattice plots, this will set the initial page to display.
label.offset  the distance from a data point to its identifying label. Numeric, in units of character widths.

arrow  a list with arguments to panel.arrows, specifying the type of arrows to draw. e.g. list(ends="both", type="closed").

show.tooltips  show tooltips for toolbar items. This uses the GTK event loop internally, which might, occasionally, cause the R terminal to freeze.

show.toolbars, show.statusbar, page.annotation, clip.annotations, keep, stay.on.top  set the corresponding window options. All are logical. Defaults are taken from playwith.options.

width, height  initial size of the plot device in inches.

pointsize  default point size for text in the Cairo device.

eval.args  whether to evaluate the plot call arguments: can be TRUE, FALSE, NA (don’t eval global vars) or a regular expression matching symbols to evaluate. Or a list. See below.

on.close  a function to be called when the user closes the plot window. The playState object will passed to the function. If the function returns TRUE, the window will not be closed.

modal  whether the window is modal: if TRUE, the session will freeze until the window is closed.

link.to  an existing playState (i.e. playwith plot) to link to. The set of brushed data points will then be synchronised between them. It is assumed that the data subscripts of the two plots correspond directly. Links can be broken with playUnlink.

playState  the playState object for an existing plot window. If given, the new plot will appear in that window, replacing the old plot. This over-rides the new argument.

plot.call  a plot call (call object), if given this is used instead of expr.

main.function  the function (or its name) appearing in the call which accepts typical plot arguments like xlim or ylab. This will only be needed in unusual cases when the default guess fails.

Details

This function opens a GTK+ window containing a plot device (from the cairoDevice package), a menubar and toolbars. There is a call toolbar (similar to the “address bar” of a web browser) at the top, showing the current plot call, which can be edited in-place. Then there are up to four toolbars, one on each side of the plot. The user interface is customisable: see playwith.options.

With the autoplay facility, playwith can function like a default graphics device (although it is not technically a graphics device itself, it is a wrapper around one).

See playwith.API for help on controlling the plot once open, as well as defining new tools. For the special case of tools to control parameter values, it is possible to create the tools automatically using the parameters argument.

Four types of plots are handled somewhat differently:

- **Lattice** graphics: recognised by returning an object of class trellis. This is the best-supported case.
• **ggplot**2 graphics: recognised by returning an object of class `ggplot`. This case is rather poorly supported.

• other **grid** graphics: you must give the `viewport` argument to enable interaction.

• base graphics: this is the default case. If a multiple-plot layout is used, interaction can only work in the last sub-plot, i.e. the settings defined by `par()`.

Some forms of interaction are based on evaluating and changing arguments to the plot call. This is designed to work in common cases, but could never work for all types of plots. To enable zooming, ensure that the main call accepts `xlim` and `ylim` arguments. Furthermore, you may need to specify `main.function` if the relevant high-level call is nested in a complex block of expressions.

To enable identification of data points, the locations of data points are required, along with appropriate labels. By default, these locations and labels will be guessed from the plot call, but this may fail. You can pass the correct values in as `data.points` and/or `labels`. Please also contact the maintainer to help improve the guesses. If identification of data points is not required, passing `data.points = NA, labels = NA` may speed things up.

Some lattice functions need to be called with subscripts = TRUE in order to correctly identify points in a multiple-panel layout. Otherwise the subscripts used will then refer to the data in each panel separately, rather than the original dataset. In this case a warning dialog box will be shown.

In order to interact with a plot, its supporting data needs to be stored: i.e. all variables appearing in the plot call must remain accessible. By default (eval.args = NA), objects that are not globally accessible will be copied into an attached environment and stored with the plot window. I.e. objects are stored unless they exist in the global environment (user workspace) or in an attached namespace. This method should work in most cases. However, it may end up copying more data than is really necessary, potentially using up memory. Note that if e.g. `foo$bar` appears in the call, the whole of `foo` will be copied.

If eval.args = TRUE then variables appearing in the plot call will be evaluated and stored even if they are defined in the global environment. Use this if the global variables might change (or be removed) before the plot is destroyed.

If eval.args = FALSE then the plot call will be left alone and no objects will be copied. This is OK if all the data are globally accessible, and will speed things up.

If a regular expression is given for eval.args then only variables whose names match it will be evaluated, and this includes global variables, as with eval.args=TRUE. In this case you can set invert.match=TRUE to store variables that are not matched. For example eval.args="^tmp" will store variables whose names begin with "tmp"; eval.args=list("'foo$", invert.match=TRUE) will store everything except foo.

*Note:* function calls appearing in the plot call will be evaluated each time the plot is updated – so random data as in `plot(rnorm(100))` will keep changing, with confusing consequences! You should therefore generate random data prior to the plot call. Changes to variables in the workspace (if they are not stored locally) may also cause inconsistencies in previously generated plots.

*Warning:* the playwith device will tend to make itself the active device any time it is clicked on, so be careful if any other devices are left open.

**Value**

`playwith` invisibly returns the `playState` object representing the plot, window and device. The result of the plot call is available as component `$result`. 
Author(s)
Felix Andrews <felix@nfrac.org>

See Also
playwith.options, autoplay, playwith.API

Examples

if (interactive()) {
  options(device.ask.default = FALSE)

  ## Scatterplot (Lattice graphics).
  ## Labels are taken from rownames of data.
  ## Right-click on the plot to identify points.
  playwith(xypplot(Income ~ log(Population / Area),
      data = data.frame(state.x77, groups = state.region,
      type = c("p", "smooth"), span = 1, auto.key = TRUE,
      xlab = "Population density, 1974 (log scale)",
      ylab = "Income per capita, 1974")

  ## Scatterplot (base graphics); similar.
  ## Note that label style can be set from a menu item.
  urbAss <- USArrests[,c("UrbanPop", "Assault")]
  playwith(plot(urbAss, panel.first = lines(lowess(urbAss)),
      col = "blue", main = "Assault vs urbanisation",
      xlab = "Percent urban population, 1973",
      ylab = "Assault arrests per 100k, 1973")

  ## Time series plot (Lattice).
  ## Date-time range can be entered directly in "time mode"
  ## (supports numeric, Date, POSIXct, yearmon and yearqtr).
  ## Click and drag to zoom in, holding Shift to constrain;
  ## or use the scrollbar to move along the x-axis.
  library(zoo)
  playwith(xypplot(sunspots ~ yearmon(time(sunspots)),
      xlim = c(1900, 1930), type = "l"),
      time.mode = TRUE)

  ## Time series plot (base graphics); similar.
  ## Custom labels are passed directly to playwith.
  tt <- time(treeering)
  treeyears <- paste(abs(tt) + (tt <= 0),
    ifelse(tt > 0, "CE", "BCE"))
  playwith(plot(treeering, xlim = c(1000, 1300),
      labels = treeyears, time.mode = TRUE)

  ## Multi-panel Lattice plot.
  ## Need subscripts = TRUE to correctly identify points.
  ## Scales are "same" so zooming applies to all panels.
  ## Use the 'Panel' tool to expand a single panel, then use
  ## the vertical scrollbar to change pages.
Depth <- equal.count(quakes$depth, number = 3, overlap = 0.1)
playwith(xypplot(lat ~ long | Depth, data = quakes,
    subscripts = TRUE, aspect = "iso", pch = ".", cex = 2),
    labels = paste("mag", quakes$mag))

## Spin and brush for a 3D Lattice plot.
## Drag on the plot to rotate in 3D (can be confusing).
## Brushing is linked to the previous xypplot (if still open).
## Note, brushing 'cloud' requires a recent version of Lattice.
playwith(cloud(-depth ~ long * lat, quakes, zlab = "altitude"),
    new = TRUE, link.to = playDevCur(), click.mode = "Brush")

## Set brushed points according to a logical condition.
playSetIDs(value = which(quakes$mag >= 6))

## Interactive control of a parameter with a slider.
xx <- rnorm(50)
playwith(plot(density(xx, bw = bandwidth), panel.last = rug(xx)),
    parameters = list(bandwidth = seq(0.05, 1, by = 0.01)))

## The same with a spinbutton (use I() to force spinbutton).
## Initial value is set as the first in the vector of values.
## This also shows a combobox for selecting text options.
xx <- rnorm(50)
kernels <- c("gaussian", "epanechnikov", "rectangular",
    "triangular", "biweight", "cosine", "optcosine"
) playwith(plot(density(xx, bw = bandwidth, kern = kernel), lty = 1ty),
    parameters = list(bandwidth = I(c(0.1, 1:50/50)),
        kernel = kernels, lty = 1:6))

## More parameters (logical, numeric, text).
playwith(stripplot(yield ~ site, data = barley,
    jitter = TRUE, type = c("p", "a"),
    aspect = aspect, groups = barley[[groups]],
    scales = list(abbreviate = abbrev),
    par.settings = list(plot.line = list(col = linecol))),
    parameters = list(abbrev = FALSE, aspect = 0.5,
        groups = c("none", "year", "variety"),
        linecol = "red"))

## Looking through 100 time series and comparing to a reference;
## Use buttons to save the current series number or its mean value.
dat <- ts(matrix(cumsum(rnorm(100*100)), ncol = 100), start = 1900)
colnames(dat) <- paste("Series", 1:100)
ref <- (dat[,3] + dat[,4]) / 2
playwith(xypplot(cbind(dat[,i], ref = ref)),
    parameters = list(i = 1:100,
        print.i = function(playState) print(playState$env$i),
        print_mean = function(p) print(mean(dat[,p$env$i]))),
    save_to_i = function(playState)
        .GlobalEnv$i <- playState$env$i,
    append_to_i = function(playState) {
        if (!exists("ii")) ii <- c()
playwith

.playwith {
  plot(x, sin(x), type = "l", ylim = c(-1.2, 1.8), col = 3, lty = 2)
  points(x, cos(x), pch = 3, col = 4)
  lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
  legend("topright", c("sin", "cos", "tan"), col = c(3,4,6),
         lty = c(2, -1, 1), pch = c(-1, 3, 4),
         merge = TRUE, bg = 'gray98')
}, data.points = pts, labels = labs)

A ggplot example.
# NOTE: only qplot()-based calls will work.
# Labels are taken from rownames of the data.
if (require(ggplot2)) {
  playwith(qplot(qsec, wt, data = mtcars) + stat_smooth())
}

A minimalist grid plot.
# This shows how to get playwith to work with custom plots:
# accept xlim/ylim and pass "viewport" to enable zooming.
myGridPlot <- function(x, y, xlim = NULL, ylim = NULL, ...) {
  if (is.null(xlim)) xlim <- extendrange(x)
  if (is.null(ylim)) ylim <- extendrange(y)
  grid.newpage()
  pushViewport(plotViewport())
  grid.rect()
  pushViewport(viewport(xscale = xlim, yscale = ylim,
                       name = "theData"))
  grid.points(x, y, ...)
  grid.xaxis()
  grid.yaxis()
  upViewport()
}
playwith(myGridPlot(1:10, 11:20, pch = 17), viewport = "theData")

Presenting the window as a modal dialog box.
# When the window is closed, ask user to confirm.
confirmClose <- function(playState) {
  if (gconfirm("Close window and report IDs?",
               input = TRUE, preview = TRUE))
    return(TRUE)
  }

GlobalEnv$ii <- c(ii, playState$env$i)
)

## Composite plot (base graphics).
## Adapted from an example in help("legend").
## In this case, the initial plot() call is detected correctly;
## in more complex cases may need e.g. main.function="plot".
## Here we also construct data points and labels manually.
x <- seq(-4*pi, 4*pi, by = pi/24)
pts <- data.frame(x = x, y = c(sin(x), cos(x), tan(x)))
labs <- rep(c("sin", "cos", "tan"), each = length(x))
labs <- paste(labs, round(180 * x / pi) %% 360)
playwith( {
  plot(x, sin(x), type = "l", xlab = c(-pi, pi)),
       ylim = c(-1.2, 1.8), col = 3, lty = 2)
  points(x, cos(x), pch = 3, col = 4)
  lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
  legend("topright", c("sin", "cos", "tan"), col = c(3,4,6),
          lty = c(2, -1, 1), pch = c(-1, 3, 4),
          merge = TRUE, bg = 'gray98')
}, data.points = pts, labels = labs)
```r
parent = playState$win)) {
  cat("Indices of identified data points:\n")
  print(playGetIDs(playState))
  return(FALSE) ## close
} else TRUE ## don't close
}

xy <- data.frame(x = 1:20, y = rnorm(20),
  row.names = letters[1:20])

playwith(xyplot(y ~ x, xy, main = "Select points, then close"),
  width = 4, height = 3.5, show.toolbars = FALSE,
  on.close = confirmPassword, modal = TRUE,
  click.mode = "Brush")

## Ask user to save plot to PNG when window is closed:
saveOnClose <- function(playState) {
  playDevSet(playState)
  if (!confirm("Save plot to PNG file? (Cancel = no)")") return(FALSE)
  fname <- gfile("Save PNG file as:", type = "save")
  if (is.na(fname)) return(TRUE) ## cancel
  dev.off(dev.copy(Cairo_png, file = fname, 
    width = dev.size()[1], height = dev.size()[2]))
  FALSE
}

#playwith.options(on.close = saveOnClose)

## Demonstrate cacheing of objects in local environment.
## By default, only local variables in the plot call are stored.

x_global <- rnorm(100)
doLocalStuff <- function("") {
  y_local <- rnorm(100)
  angle <- (atan2(y_local, x_global) / (2*pi)) + 0.5
  color <- hsv(h = angle, v = 0.75)
  doRays <- function(x, y, col) {
    segments(0, 0, x, y, col = col)
  }

  playwith(plot(x_global, y_local, pch = 8, col = color,
    panel.first = doRays(x_global, y_local, color)),
    ...)}

doLocalStuff(title = "locals only") ## eval.args = NA is default
## List objects that have been copied and stored:
## Note: if you rm(x_global) now, redraws will fail.
ls(playDevCur()$env)
## Next: store all data objects (in a new window):
doLocalStuff(title = "all stored", eval.args = TRUE, new = TRUE)
ls(playDevCur()$env)
## Now there are two devices open:
str(playDevList())
playDevCur()
playDevOff()
playDevCur()
```

## Not run:
## Big data example, do not try to guess labels or time.mode.
gc()
bigobj <- rpois(5000000, 1)
print(object.size(bigobj), units = "Mb")
gc()
playwith(qqmath(~ bigobj, f.value = ppoints(500)),
  data.points = NA, labels = NA, time.mode = FALSE)
playDevOff()
gc()
## or generate the trellis object first:
trel <- qqmath(~ bigobj, f.value = ppoints(500))
playwith(trel)
m(trel)
## in this case, it is much better to compute the sample first:
subobj <- quantile(bigobj, ppoints(500), na.rm = TRUE)
playwith(qqmath(~ subobj))
m(subobj)
m(bigobj)

## End(Not run)

## See demo(package = "playwith") for examples of new tools.
}

---

### Description

The **playwith** Application Programming Interface.

### Details

**playwith** plots (incorporating a plot, window and device) are represented by a **playState** object.

The following sections list the API functions that can be used to work with the plot, and to write new interactive tools. See the links to specific help pages for details. In case these are insufficient, you may work with the **playState** object itself.

### Device management

These are similar to **dev.set** etc.

**playDevCur()** returns the current or last active **playState** – this is not necessarily the active graphics device.

**playDevList()** lists all open **playStates**.

**playDevSet(playState)** sets the current **playState**, and sets the active graphics device.

**playDevOff(playState)** closes the device and window (**dev.off()** also works).
Common user commands

These functions are also available as menu items.

`playGetIDs(playState, type, labels)` returns indices (or labels) of currently brushed or labelled data points.

`playSetIDs(playState, value, type, space, add, redraw, pos)` sets which data points are brushed or labelled. `type` defaults to "brushed"; `space` is ignored unless `type = "labelled"`.

`playClear(playState, type, redraw)` remove one or more of the types "annotations", "labelled", "brushed". The latter will also apply to any linked plots. If `redraw = FALSE` the display will not be updated.

`playUndo(playState)` reverts the last change any annotations (including the set of labelled and brushed data points).

`updateLinkedSubscribers(playState, redraw)` triggers a redraw of any linked plots.

`playUnLink(playState)` removes links from the given plot to any other plots (for linked brushing).

`playSourceCode(playState)` returns (deparsed) R code to reproduce the current plot display.

Interaction

These functions allow the user to click or drag on the plot. Click or drag locations are converted into the native coordinates of whatever plot space they occurred in (but are available as device coordinates too).

`playSelectData(playState, prompt, scales, foo)` interactively select data from the plot (one point or a whole region).

`playPointInput(playState, prompt)` prompt for a click on the plot. Similar to `locator`.

`playLineInput(playState, prompt, scales)` prompt to drag a line.

`playRectInput(playState, prompt, scales)` prompt to drag a rectangular region.

`playPrompt(playState, text)` sets the statusbar text. Pass `NULL` to reset. The GUI is frozen when the prompt is set and unfrozen when reset.

`playFreezeGUI(playState), playThawGUI(playState)` disables or re-enables the GUI.

Working with the display

`playDo(playState, expr, space, clip.off, return.code)` evaluates the given expression in the given plot space, i.e. after moving to the corresponding grid viewport.

`playAnnotate(playState, annot, space, add, redraw)` adds the annotation (a call or expression to draw on the plot) to the list of persistent annotations, and draws them. These will always be drawn after plotting (unless they are removed by the user).

`rawxlim(playState, space)` gets or sets the x axis limits, in native coordinates of the given space (viewport).

`rawylim(playState, space)` same as `rawxlim`, for y axis limits.

`spaceCoordsToDataCoords(playState, xy)` converts raw space (viewport) coordinates to the data scale by applying an inverse log transformation if necessary. If there are no log scales this does nothing.
dataCoordsToSpaceCoords(playState, xy) converts data coordinates to raw space (viewport) coordinates by applying a log transformation if necessary. If there are no log scales this does nothing.

Working with the call

`callArg(playState, arg, eval, data)` gets or sets arguments to the main plot call.

`mainCall(playState)` gets or replaces the main plot call (which is not necessarily the same as the top-level call, `playState$call`).

`updateMainCall(playState)` locates the main plot call within the top-level call – by matching against the given `main.function`, or by guessing. This allows `callArg()` to work correctly: it should be called if `playState$call` is replaced.

`playReplot(playState)` redraws the plot by evaluating the plot call, and runs update actions. This should be called after changing plot arguments (or annotations if that requires a redraw). However, if the data has changed or the type of plot (e.g. the high-level plot function) has changed, `playNewPlot` should be used instead. `playReplot` is triggered when zooming, editing plot settings, removing annotations, etc.

`playNewPlot(playState)` redraws the plot by evaluating the plot call, updates the main call, runs init actions (such as detecting whether zooming, identifying or brushing data points is possible), as well as running update actions as with `playReplot`. Note: `playNewPlot` is triggered when the call text is edited manually in the GUI.

Working with data

`xyData(playState, space)` attempts to extract plot data in terms of x and y locations, but in the original data form (such as factor, date or time). This uses the generic function `plotCoords` to generate plot locations; new methods can be defined for non-standard plot types.

`xyCoords(playState, space)` same as `xyData`, converted to numeric coordinates.

`getDataArg(playState, eval)` attempts to extract the underlying data set (typically a `data.frame`), which may contain more variables than those currently plotted. This may come from a `data` argument to the plot call, or from a `with()` block.

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

`playwith`, `playState`, `convertFromDevicePixels`

Examples

```r
if (interactive()) {

demo(package = "playwith")
}
```
playwith.history  Session history for playwith

Description

A basic history of plot calls in playwith, like history.

Usage

playwith.history(max.show = 100, ...)

Arguments

max.show maximum number of history items to show.
... ignored.

Details

The returned history is a combination of the stored session history (stored in playwith:::PlaywithEnv$history and the current history for all open playwith windows.

See Also

playwith

Examples

if (interactive()) {
  playwith.history()
}

playwith.options  User default settings for playwith

Description

A basic user settings facility, like options and lattice.options.

Usage

playwith.options(...)  
playwith.getOption(name)
playwith.options

Arguments

name character giving the name of a setting.

... new options can be defined, or existing ones modified, using one or more arguments of the form 'name = value' or by passing a list of such tagged values. Existing values can be retrieved by supplying the names (as character strings) of the components as unnamed arguments.

Details

These functions are direct copies of the lattice equivalents: see lattice.options.
The available options can be seen with str(playwith.options()). Many of these simply provide defaults for corresponding arguments to the playwith function.

See Also

playwith

Examples

oopt <- playwith.options()
str(oopt)

playwith.options(time.mode = FALSE)

## list options are merged, not replaced
playwith.getOption("arrow")
playwith.options(arrow = list(type = "closed", length = 0.1))
playwith.getOption("arrow")

playwith.options(save.as.format = "png")
playwith.options(toolbar.style = "icons")
playwith.options(deparse.options =
  c("keepInteger", "showAttributes", "keepNA"))

## make a new "style shortcut" (an arbitrary expression)
## to add a standard sub-title to the plot:
doMySub <- quote({
  txt <- input("Enter subtitle text: ",
                text = paste(Sys.time(), Sys.info()"login",
                             R.version.string, sep = ", "))
  if (!is.na(txt))
    callArg(playState, "sub") <- if (nchar(txt) > 0) txt
})
playwith.options(styleShortcuts = list("mySub" = doMySub))

## try it:
if (interactive())
  playwith(plot(1:10))

## reset
playwith.options(oopt)
**plotCoords**

API for defining data coordinates of a plot

**Description**

Given a call to a plot function, return the data coordinates.

**Usage**

```r
plotCoords(name, object, call, envir, ...)  

## Default S3 method:
plotCoords(name, object, call, envir, data, panel.args, ...)  

## S3 method for class 'qqnorm'
plotCoords(name, object, call, envir, ...)  

## S3 method for class 'qqplot'
plotCoords(name, object, call, envir, ...)  

plotCoords.plot(name, object, call, envir, ...)  

## Default S3 method:
plotCoords.plot(name, object, call, envir, data, ...)  

## S3 method for class 'dendrogram'
plotCoords.plot(name, object, call, envir, ...)  

## S3 method for class 'mca'
plotCoords.plot(name, object, call, envir, ...)  

plotCoords.biplot(name, object, call, envir, ...)  

## Default S3 method:
plotCoords.biplot(name, object, call, envir, ...)  

## S3 method for class 'prcomp'
plotCoords.biplot(name, object, call, envir, ...)  

## S3 method for class 'princomp'
plotCoords.biplot(name, object, call, envir, ...)  

## S3 method for class 'qqmath'
plotCoords(name, object, call, envir, panel.args, ...)  

## S3 method for class 'cloud'
plotCoords(name, object, call, envir, panel.args, ...)  

## S3 method for class 'parallel'
plotCoords(name, object, call, envir, panel.args, ...)  

## S3 method for class 'splom'
plotCoords(name, object, call, envir, panel.args, packet, ...)
Arguments

name
  The class of this object is the name of the plot function. Hence methods can be
  defined for different plot functions.

object
  the object passed as first argument to the plot call.

call
  the plot call.

envir
  environment containing objects referenced by the call. Call arguments should
  be evaluated in this environment.

panel.args
  passed for Lattice plots only: panel arguments for the relevant panel.

packet
  passed for Lattice plots only: packet number for which to return data (corre-
  sponds to panel.args).

data
  passed for non-Lattice plots only: a "data" argument, or NULL, to be used in
  evaluating call arguments (in addition to envir).

... ignored.

Details

None yet...

Value

a list with components:

x, y
  data point coordinates (in native panel / user coordinates).

subscripts
  (optional) data point subscripts.

Author(s)

Felix Andrews <felix@nfrac.org>

See Also

xyData, xy.coords

Examples

## Note, these are not designed to be called directly;
## they are used internally in playwith().
## But for demonstration purposes:
pargs <- trellis.panelArgs(qqmath(rnorm(20)), packet = 1)
plotCoords(structure("qqmath", class = "qqmath"),
          call = quote(qqmath(rnorm(20))), envir = new.env(),
          panel.args = pargs)
rawXLim \hspace{1cm} \textit{Get or set current plot limits}

\textbf{Description}

Part of the \texttt{playwith} Application Programming Interface.

\textbf{Usage}

\begin{verbatim}
rawXLim(playState, space = "plot")
rawYLim(playState, space = "plot")
rawXLim(playState) <- value
rawYLim(playState) <- value

spaceCoordsToDataCoords(playState, xy)
dataCoordsToSpaceCoords(playState, xy)
\end{verbatim}

\textbf{Arguments}

\begin{description}
\item[playState] \hspace{1cm} a \texttt{playState} object representing the plot, window and device.
\item[space] \hspace{1cm} character, the plot space for which to get or set limits. See the \texttt{space} argument to \texttt{playDo}; however, in this case, \texttt{space="plot"} will always return a value: if it is a Lattice plot with multiple panels, one will be chosen arbitrarily.
\item[value] \hspace{1cm} numeric length 2, the new nominal x or y limits (for \texttt{xlim} or \texttt{ylim} plot arguments).
\item[xy] \hspace{1cm} list with at least one of the elements \texttt{x} and \texttt{y} (as numeric).
\end{description}

\textbf{Details}

\texttt{rawXLim} returns the current plot limits, on a numeric, linear scale. This is as simple as: \texttt{playDo(playState, space=space, x=convertX(unit(0:1, "npc"), "native", valueOnly=TRUE), y=convertY(unit(0:1, "npc"), "native", valueOnly=TRUE)}, except that the default \texttt{space="plot"} will always return a value: if it is a Lattice plot with multiple panels, one will be chosen arbitrarily.

The assignment form converts a numeric range, in the raw native plot coordinates, to values suitable for the plot \texttt{xlim} argument: it may convert back from log-transformed scales, and convert to factor levels if necessary. It then updates the current plot call with the new value.

\texttt{spaceCoordsToDataCoords} converts from the native viewport coordinates to the data coordinates, which simply involves converting from a log scale if necessary. \texttt{dataCoordsToSpaceCoords} is the inverse case: applying a log transformation if necessary. It used to refer to the position of data points in the viewport.

\textbf{Value}

the extractor form returns the x or y plot limits as numeric length 2.
Author(s)

Felix Andrews <felix@nfrac.org>

See Also

playwith.API

Examples

if (interactive()) {

  playwith(plot(1:10, log="x"))
  playState <- playDevCur()
  rawXLim(playState) <- -0.04 1.04
  rawXLim(playState) <- c(0, 2)
  playRepplot(playState)
  # now xlim=c(1, 100)
  (rawx <- rawXLim(playState)) <- -0.08 2.08

  spaceCoordsToDataCoords(playState, list(x=rawx))
  dataCoordsToSpaceCoords(playState, list(x=1:10))
}

xyCoords

Get playwith plot data points

Description

Part of the playwith Application Programming Interface.

Usage

xyCoords(playState = playDevCur(), space = "plot")
xyData(playState = playDevCur(), space = "plot")
getDataArg(playState = playDevCur(), eval = TRUE)

Arguments

playState

a playState object representing the plot, window and device.

space

character, the plot space for which to get data. This is only relevant to multi-panel lattice plots, where data is split across panels. In this case, if space="page", the combined data from all panels is returned. See the space argument to playDo.

eval

whether to evaluate the argument; otherwise return the it as it appears in the call.
xyCoords

Details
None yet.

Value
the returned value is a list with elements x and y, which are numeric vectors in the case of xyCoords. Can also include an element subscripts, and generally will for lattice plots (similar to the result of calling \texttt{trellis.panelArgs}).

Author(s)
Felix Andrews <felix@nfrac.org>

See Also
\texttt{playwith.API, plotCoords}

Examples

\begin{verbatim}
if (interactive()) {

library(lattice)
x <- as.Date("1990-01-01") + 1:20 - 1
ab <- rep(c("a", "b"), each=10)
playwith(xyplot(1:20 ~ x | ab, subscripts = TRUE))
playState <- playDevCur()
xyCoords(playState, space="packet 2")
xyData(playState, space="packet 2")
try(xyData(playState, space="plot"))
getDataArg(playState)

}
\end{verbatim}
Index

*Topic **aplot**
  - panel.user.text, 8
*Topic **device**
  - autoplay, 2
  - playwith, 17
*Topic **dplot**
  - convertFromDevicePixels, 5
  - plotCoords, 31
*Topic **dynamic**
  - playwith, 17
*Topic **iplot**
  - autoplay, 2
  - convertFromDevicePixels, 5
  - identifyGrob, 7
  - parameterControlTool, 9
  - playGetIDs, 11
  - playSelectData, 14
  - playwith, 17
*Topic **programming**
  - callArg, 3
  - parameterControlTool, 9
  - playDo, 10
  - playPointInput, 13
  - playState, 16
  - playwith.API, 26
  - playwith.options, 29
  - rawXlim, 33
  - xyCoords, 34
*Topic **utilities**
  - playwith.API, 26
  - playwith.history, 29

[[, 4

autoplay, 2, 20, 22

Cairo, 20
call, 20
callArg, 3, 28
callArg<- (callArg), 3
case.names.Date (plotCoords), 31
case.names.dendrogram (plotCoords), 31
case.names.hclust (plotCoords), 31
case.names.mca (plotCoords), 31
case.names.POSIXt (plotCoords), 31
case.names.prcomp (plotCoords), 31
case.names.princomp (plotCoords), 31
case.names.ts (plotCoords), 31
case.names.yearmon (plotCoords), 31
case.names.yearqtr (plotCoords), 31
case.names.zoo (plotCoords), 31
convertFromDevicePixels, 5, 28
convertToDevicePixels
  (convertFromDevicePixels), 5
dataCoordsToSpaceCoords (rawXlim), 33
Date, 19
dev.set, 26
environment, 16
gdkKeySyms, 19
getDataType, 28
getDataType (xyCoords), 34
grid.convert, 6
grid.edit, 7
grid.ls, 6, 7
grid.newpage, 2
grid.remove, 7
grid.show.layout, 6
grob, 6, 7
grobBBDevicePixels
  (convertFromDevicePixels), 5
grobBoundingBoxes, 7
grobBoundingBoxes
  (convertFromDevicePixels), 5
grobX, 6
gtkAcceleratorParse, 19
GtkActionEntry, 18
gtkToolButton, 9
GtkUIManager, 17

36
INDEX

gtkWindow, 16
history, 29
identify, 15
identifyGrob, 7
inviewport (convertFromDevicePixels), 5
lattice.options, 29, 30
llines, 8, 9
locator, 13, 27
mainCall, 28
mainCall (callArg), 3
mainCall<- (callArg), 3
options, 29
panel.arrows, 20
panel.brushlines (panel.usetext), 8
panel.brushpoints (panel.usetext), 8
panel.usetext, 8
parameterControlTool, 9
playAnnotate (playwith.API), 26
playClear, 27
playDevCur (playwith.API), 11
playDevList (playwith.API), 26
playDevOff (playwith.API), 26
playDevSet (playwith.API), 26
playDo, 10, 14, 15, 27, 33, 34
playFreezeGUI (playwith.API), 26
playGetIDs, 11, 27
playLineInput, 27
playLineInput (playPointInput), 13
playNewPlot (playwith.API), 26
playPointInput, 10, 11, 13, 27
playPrompt (playwith.API), 26
playRectInput, 15, 27
playRectInput (playPointInput), 13
playReplot (playwith.API), 26
playSelectData, 10, 11, 14, 14, 27
playSetIDs, 27
playSetIDs (playGetIDs), 11
playSourceCode (playwith.API), 26
playState, 4, 10, 12, 13, 15, 16, 19–21, 26, 28, 33, 34
playThawGUI (playwith.API), 26
playUndo (playwith.API), 26
playUnlink (playwith.API), 26
playwith, 2, 3, 8–10, 16, 17, 17, 26, 28–30
playwith.API, 4, 11, 12, 14–17, 20, 22, 26, 34, 35
playwith.getOption (playwith.options), 29
playwith.history, 29
playwith.options, 19, 20, 22, 29
plot, 2
plot.new, 2
plotCoords, 28, 31, 35
plotOnePage (playwith), 17
POSIXt, 19
print.playState (playState), 16
print.trellis, 2
rawXlim, 27, 33
rawXlim<- (rawXlim), 33
rawYlim (rawXlim), 33
rawYlim<- (rawXlim), 33
regular expression, 20
showGrosBB (convertFromDevicePixels), 5
spaceCoordsToDataCoords, 27
spaceCoordsToDataCoords (rawXlim), 33
text, 15
trellis.focus, 10
trellis.panelArgs, 35
trellis.par.get, 9
unit, 5
updateLinkedSubscribers (playwith.API), 26
updateMainCall, 28
updateMainCall (callArg), 3
viewport, 10, 18
vpPath, 5, 18
xy.coords, 18, 32
xyCoords, 28, 34
xyData, 28, 32
xyData (xyCoords), 34