Package 'tinyplot'

February 5, 2025

Type Package

Title Lightweight Extension of the Base R Graphics System

Version 0.3.0

Date 2025-02-05

Description Lightweight extension of the base R graphics system, with support for automatic legends, facets, themes, and various other enhancements.

License Apache License (>= 2)

Depends R (>= 4.0)

Imports graphics, grDevices, stats, tools, utils

Suggests altdoc (>= 0.5.0), fontquiver, png, rsvg, svglite, tinytest, tinysnapshot (>= 0.0.3), knitr

Encoding UTF-8

RoxygenNote 7.3.2

URL https://grantmcdermott.com/tinyplot/

BugReports https://github.com/grantmcdermott/tinyplot/issues

NeedsCompilation no

Author Grant McDermott [aut, cre] (<https://orcid.org/0000-0001-7883-8573>), Vincent Arel-Bundock [aut] (<https://orcid.org/0000-0003-1995-6531>), Achim Zeileis [aut] (<https://orcid.org/0000-0003-0918-3766>), Etienne Bacher [ctb]

Maintainer Grant McDermott <gmcd@amazon.com>

Repository CRAN

Date/Publication 2025-02-05 19:10:02 UTC

Contents

draw_legend 2
get_saved_par 5
tinyplot 7

tinyplot_add . . . 19
tinytheme . . . 20
tpar . . . 23
type_abline . . . 26
type_area . . . 27
type_boxplot . . . 28
type_density . . . 29
type_errorbar . . . 32
type_function . . . 33
type_glm . . . 34
type_histogram . . . 34
type_hline . . . 36
type_jitter . . . 37
type_lines . . . 38
type_lm . . . 38
type_loess . . . 39
type_points . . . 40
type_polygon . . . 40
type_polypath . . . 41
type_qq . . . 42
type_rect . . . 42
type_ridge . . . 43
type_rug . . . 47
type_segments . . . 49
type_spineplot . . . 49
type_spline . . . 51
type_summary . . . 53
type_text . . . 54
type_vline . . . 55

Index **56**

`draw_legend` *Calculate placement of legend and draw it*

Description

Internal function used to calculate the placement of (including outside the plotting area) and drawing of legend.

Usage

```
draw_legend(
  legend = NULL,
  legend_args = NULL,
  by_dep = NULL,
  lgnd_labs = NULL,
  type = NULL,
```

```
  pch = NULL,
  lty = NULL,
  lwd = NULL,
  col = NULL,
  bg = NULL,
  cex = NULL,
  gradient = FALSE,
  lmar = NULL,
  has_sub = FALSE,
  new_plot = TRUE
)
```

Arguments

legend	Legend placement keyword or list, passed down from tinyplot.
legend_args	Additional legend arguments to be passed to `legend()`.
by_dep	The (deparsed) "by" grouping variable name.
lgnd_labs	The labels passed to `legend(legend = ...)`.
type	Plotting type(s), passed down from tinyplot.
pch	Plotting character(s), passed down from tinyplot.
lty	Plotting linetype(s), passed down from tinyplot.
lwd	Plotting line width(s), passed down from tinyplot.
col	Plotting colour(s), passed down from tinyplot.
bg	Plotting character background fill colour(s), passed down from tinyplot.
cex	Plotting character expansion(s), passed down from tinyplot.
gradient	Logical indicating whether a continuous gradient swatch should be used to represent the colors.
lmar	Legend margins (in lines). Should be a numeric vector of the form `c(inner, outer)`, where the first number represents the "inner" margin between the legend and the plot, and the second number represents the "outer" margin between the legend and edge of the graphics device. If no explicit value is provided by the user, then reverts back to `tpar("lmar")` for which the default values are `c(1.0, 0.1)`.
has_sub	Logical. Does the plot have a sub-caption. Only used if keyword position is "bottom!", in which case we need to bump the legend margin a bit further.
new_plot	Logical. Should we be calling plot.new internally?

Value

No return value, called for side effect of producing a(n empty) plot with a legend in the margin.

Examples

```
oldmar = par("mar")

draw_legend(
  legend = "right!", ## default (other options incl, "left(!)", ""bottom(!)", etc.)
  legend_args = list(title = "Key", bty = "o"),
  lgnd_labs = c("foo", "bar"),
  type = "p",
  pch = 21:22,
  col = 1:2
)

# The legend is placed in the outer margin...
box("figure", col = "cyan", lty = 4)
# ... and the plot is proportionally adjusted against the edge of this
# margin.
box("plot")
# You can add regular plot objects per normal now
plot.window(xlim = c(1,10), ylim = c(1,10))
points(1:10)
points(10:1, pch = 22, col = "red")
axis(1); axis(2)
# etc.

# Important: A side effect of draw_legend is that the inner margins have been
# adjusted. (Here: The right margin, since we called "right!" above.)
par("mar")

# To reset you should call `dev.off()` or just reset manually.
par(mar = oldmar)

# Note that the inner and outer margin of the legend itself can be set via
# the `lmar` argument. (This can also be set globally via
# `tpar(lmar = c(inner, outer))`.)
draw_legend(
  legend_args = list(title = "Key", bty = "o"),
  lgnd_labs = c("foo", "bar"),
  type = "p",
  pch = 21:22,
  col = 1:2,
  lmar = c(0, 0.1) ## set inner margin to zero
)
box("figure", col = "cyan", lty = 4)

par(mar = oldmar)

# Continuous (gradient) legends are also supported
draw_legend(
  legend = "right!",
  legend_args = list(title = "Key"),
  lgnd_labs = LETTERS[1:5],
  col = hcl.colors(5),
```

```
  gradient = TRUE ## enable gradient legend
)

par(mar = oldmar)
```

`get_saved_par` *Retrieve the saved graphical parameters*

Description

Convenience function for retrieving the graphical parameters (i.e., the full list of `tag = value` pairs held in `par`) from either immediately before or immediately after the most recent tinyplot call.

Usage

```
get_saved_par(when = c("before", "after", "first"))
```

Arguments

`when`	character. From when should the saved parameters be retrieved? Either "before" (the default) or "after" the preceding `tinyplot` call.

Details

A potential side-effect of tinyplot is that it can change a user's `par` settings. For example, it may adjust the inner and outer plot margins to make space for an automatic legend; see draw_legend. While it is possible to immediately restore the original `par` settings upon exit via the `tinyplot(..., restore.par = TRUE)` argument, this is not the default behaviour. The reason being that we need to preserve the adjusted parameter settings in case users want to add further graphical annotations to their plot (e.g., `abline`, `text`, etc.) Nevertheless, it may still prove desirable to recall and reset these original graphical parameters after the fact (e.g., once all these extra annotations have been added). That is the purpose of this get_saved_par function.

Of course, users may prefer to manually capture and reset graphical parameters, as per the standard method described in the `par` documentation. For example:

```
op = par(no.readonly = TRUE)  # save current par settings
# <do lots of (tiny)plotting>
par(op)                       # reset original pars
```

This standard manual approach may be safer than get_saved_par because it offers more precise control. Specifically, the value of get_saved_par itself will be reset after ever new tinyplot call; i.e. it may inherit an already-changed set of parameters. Users should bear these trade-offs in mind when deciding which approach to use. As a general rule, get_saved_par offers the convenience of resetting the original `par` settings even if a user forgot to save them beforehand. But one should avoid invoking it after a series of consecutive tinyplot calls.

Finally, note that users can always call `dev.off` to reset all `par` settings to their defaults.

Value

A list of par settings.

Examples

```
#
# Contrived example where we draw a grouped scatterplot with a legend and
# manually add corresponding best fit lines for each group...
#

# First draw the grouped scatterplot
tinyplot(Sepal.Length ~ Petal.Length | Species, iris)

# Preserving adjusted par settings is good for adding elements to our plot
for (s in levels(iris$Species)) {
  abline(
    lm(Sepal.Length ~ Petal.Length, iris, subset = Species==s),
    col = which(levels(iris$Species)==s)
  )
}

# Get saved par from before the preceding tinyplot call (but don't use yet)
sp = get_saved_par("before")

# Note the changed margins will affect regular plots too, which is probably
# not desirable
plot(1:10)

# Reset the original parameters (could use `par(sp)` here)
tpar(sp)
# Redraw our simple plot with our corrected right margin
plot(1:10)

#
# Quick example going the other way, "correcting" for par.restore = TRUE...
#

tinyplot(Sepal.Length ~ Petal.Length | Species, iris, restore.par = TRUE)
# Our added best lines will be wrong b/c of misaligned par
for (s in levels(iris$Species)) {
  abline(
    lm(Sepal.Length ~ Petal.Length, iris, subset = Species==s),
    col = which(levels(iris$Species)==s), lty = 2
  )
}
# grab the par settings from the _end_ of the preceding tinyplot call to fix
tpar(get_saved_par("after"))
# now the best lines are correct
for (s in levels(iris$Species)) {
  abline(
    lm(Sepal.Length ~ Petal.Length, iris, subset = Species==s),
    col = which(levels(iris$Species)==s)
```

```
  )
}

# reset again to original saved par settings before exit
tpar(sp)
```

`tinyplot` *Lightweight extension of the base R plotting function*

Description

Enhances the base `plot` function. Supported features include automatic legends and facets for grouped data, additional plot types, theme customization, and so on. Users can call either `tinyplot()`, or its shorthand alias `plt()`.

Usage

```
tinyplot(x, ...)

## Default S3 method:
tinyplot(
  x = NULL,
  y = NULL,
  by = NULL,
  facet = NULL,
  facet.args = NULL,
  data = NULL,
  type = NULL,
  xlim = NULL,
  ylim = NULL,
  log = "",
  main = NULL,
  sub = NULL,
  xlab = NULL,
  ylab = NULL,
  ann = par("ann"),
  axes = TRUE,
  frame.plot = NULL,
  asp = NA,
  grid = NULL,
  palette = NULL,
  legend = NULL,
  pch = NULL,
  lty = NULL,
  lwd = NULL,
  col = NULL,
```

```
  bg = NULL,
  fill = NULL,
  alpha = NULL,
  cex = 1,
  restore.par = FALSE,
  xmin = NULL,
  xmax = NULL,
  ymin = NULL,
  ymax = NULL,
  add = FALSE,
  draw = NULL,
  file = NULL,
  width = NULL,
  height = NULL,
  empty = FALSE,
  xaxt = NULL,
  yaxt = NULL,
  flip = FALSE,
  xaxs = NULL,
  yaxs = NULL,
  ...
)

## S3 method for class 'formula'
tinyplot(
  x = NULL,
  data = parent.frame(),
  facet = NULL,
  facet.args = NULL,
  type = NULL,
  xlim = NULL,
  ylim = NULL,
  main = NULL,
  sub = NULL,
  xlab = NULL,
  ylab = NULL,
  ann = par("ann"),
  axes = TRUE,
  frame.plot = NULL,
  asp = NA,
  grid = NULL,
  pch = NULL,
  col = NULL,
  lty = NULL,
  lwd = NULL,
  restore.par = FALSE,
  formula = NULL,
  subset = NULL,
```

```
  na.action = NULL,
  drop.unused.levels = TRUE,
  ...
)

## S3 method for class 'density'
tinyplot(x = NULL, type = c("l", "area"), ...)

plt(x, ...)
```

Arguments

x, y	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable; most likely the names of existing vectors or columns of data frames. See the 'Examples' section below, or the function `xy.coords` for details. If supplied separately, `x` and `y` must be of the same length.
...	other graphical parameters. If `type` is a character specification (such as `"hist"`) then any argument names that match those from the corresponding `type_*()` function (such as `type_hist`) are passed on to that. All remaining arguments from `...` can be further graphical parameters, see `par`).
by	grouping variable(s). The default behaviour is for groups to be represented in the form of distinct colours, which will also trigger an automatic legend. (See `legend` below for customization options.) However, groups can also be presented through other plot parameters (e.g., `pch` or `lty`) by passing an appropriate "by" keyword; see Examples. Note that continuous (i.e., gradient) colour legends are also supported if the user passes a numeric or integer to `by`. To group by multiple variables, wrap them with `interaction`.
facet	the faceting variable(s) that you want arrange separate plot windows by. Can be specified in various ways: • In "atomic" form, e.g. `facet = fvar`. To facet by multiple variables in atomic form, simply interact them, e.g. `interaction(fvar1, fvar2)` or `factor(fvar1):factor(fvar2)`. • As a one-sided formula, e.g. `facet = ~fvar`. Multiple variables can be specified in the formula RHS, e.g. `~fvar1 + fvar2` or `~fvar1:fvar2`. Note that these multi-variable cases are *all* treated equivalently and converted to `interaction(fvar1, fvar2, ...)` internally. (No distinction is made between different types of binary operators, for example, and so `f1+f2` is treated the same as `f1:f2`, is treated the same as `f1*f2`, etc.) • As a two-side formula, e.g. `facet = fvar1 ~ fvar2`. In this case, the facet windows are arranged in a fixed grid layout, with the formula LHS defining the facet rows and the RHS defining the facet columns. At present only single variables on each side of the formula are well supported. (We don't recommend trying to use multiple variables on either the LHS or RHS of the two-sided formula case.) • As a special `"by"` convenience keyword, in which case facets will match the grouping variable(s) passed to by above.

`facet.args` an optional list of arguments for controlling faceting behaviour. (Ignored if `facet` is NULL.) Supported arguments are as follows:

- `nrow`, `ncol` for overriding the default "square" facet window arrangement. Only one of these should be specified, but `nrow` will take precedence if both are specified together. Ignored if a two-sided formula is passed to the main `facet` argument, since the layout is arranged in a fixed grid.
- `free` a logical value indicating whether the axis limits (scales) for each individual facet should adjust independently to match the range of the data within that facet. Default is `FALSE`. Separate free scaling of the x- or y-axis (i.e., whilst holding the other axis fixed) is not currently supported.
- `fmar` a vector of form `c(b,l,t,r)` for controlling the base margin between facets in terms of lines. Defaults to the value of `tpar("fmar")`, which should be `c(1,1,1,1)`, i.e. a single line of padding around each individual facet, assuming it hasn't been overridden by the user as part their global `tpar` settings. Note some automatic adjustments are made for certain layouts, and depending on whether the plot is framed or not, to reduce excess whitespace. See `tpar` for more details.
- `cex`, `font`, `col`, `bg`, `border` for adjusting the facet title text and background. Default values for these arguments are inherited from `tpar` (where they take a "facet." prefix, e.g. `tpar("facet.cex")`). The latter function can also be used to set these features globally for all `tinyplot` plots.

`data` a data.frame (or list) from which the variables in formula should be taken. A matrix is converted to a data frame.

`type` character string or call to a `type_*()` function giving the type of plot desired.

- NULL (default): Choose a sensible type for the type of `x` and `y` inputs (i.e., usually `"p"`).
- 1-character values supported by `plot`:
  - `"p"` Points
  - `"l"` Lines
  - `"b"` Both points and lines
  - `"c"` Empty points joined by lines
  - `"o"` Overplotted points and lines
  - `"s"` Stair steps
  - `"S"` Stair steps
  - `"h"` Histogram-like vertical lines
  - `"n"` Empty plot over the extent of the data
- `tinyplot`-specific types. These fall into several categories:
  - Shapes:
    - `"area"` / `type_area()`: Plots the area under the curve from y = 0 to y = f(x).
    - `"errorbar"` / `type_errorbar()`: Adds error bars to points; requires `ymin` and `ymax`.
    - `"pointrange"` / `type_pointrange()`: Combines points with error bars.

    - "polygon" / type_polygon(): Draws polygons.
    - "polypath" / type_polypath(): Draws a path whose vertices are given in x and y.
    - "rect" / type_rect(): Draws rectangles; requires xmin, xmax, ymin, and ymax.
    - "ribbon" / type_ribbon(): Creates a filled area between ymin and ymax.
    - "segments" / type_segments(): Draws line segments between pairs of points.
    - "text" / type_text(): Add text annotations.
  - Visualizations:
    - "boxplot" / type_boxplot(): Creates a box-and-whisker plot.
    - "density" / type_density(): Plots the density estimate of a variable.
    - "histogram" / type_histogram(): Creates a histogram of a single variable.
    - "jitter" / type_jitter(): Jittered points.
    - "qq" / type_qq(): Creates a quantile-quantile plot.
    - "ridge" / type_ridge(): Creates a ridgeline (aka joy) plot.
    - "rug" / type_rug(): Adds a rug to an existing plot.
    - "spineplot" / type_spineplot(): Creates a spineplot or spinogram.
  - Models:
    - "loess" / type_loess(): Local regression curve.
    - "lm" / type_lm(): Linear regression line.
    - "glm" / type_glm(): Generalized linear model fit.
    - "spline" / type_spline(): Cubic (or Hermite) spline interpolation.
  - Functions:
    - type_abline(): line(s) with intercept and slope.
    - type_hline(): horizontal line(s).
    - type_vline(): vertical line(s).
    - type_function(): arbitrary function.
    - type_summary(): summarize y by unique values of x.

xlim — the x limits (x1, x2) of the plot. Note that x1 > x2 is allowed and leads to a 'reversed axis'. The default value, NULL, indicates that the range of the finite values to be plotted should be used.

ylim — the y limits of the plot.

log — a character string which contains "x" if the x axis is to be logarithmic, "y" if the y axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.

main — a main title for the plot, see also title.

sub — a subtitle for the plot.

xlab — a label for the x axis, defaults to a description of x.

`ylab` a label for the y axis, defaults to a description of y.

`ann` a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.

`axes` logical or character. Should axes be drawn (`TRUE` or `FALSE`)? Or alternatively what type of axes should be drawn: `"standard"` (with axis, ticks, and labels; equivalent to `TRUE`), `"none"` (no axes; equivalent to `FALSE`), `"ticks"` (only ticks and labels without axis line), `"labels"` (only labels without ticks and axis line), `"axis"` (only axis line and labels but no ticks). To control this separately for the two axes, use the character specifications for `xaxt` and/or `yaxt`.

`frame.plot` a logical indicating whether a box should be drawn around the plot. Can also use `frame` as an acceptable argument alias. The default is to draw a frame if both axis types (set via `axes`, `xaxt`, or `yaxt`) include axis lines.

`asp` the y/xy/x aspect ratio, see `plot.window`.

`grid` argument for plotting a background panel grid, one of either:

- a logical (i.e., `TRUE` to draw the grid), or
- a panel grid plotting function like `grid()`. Note that this argument replaces the `panel.first` and `panel.last` arguments from base `plot()` and tries to make the process more seamless with better default behaviour. The default behaviour is determined by (and can be set globally through) the value of `tpar("grid")`.

`palette` one of the following options:

- NULL (default), in which case the palette will be chosen according to the class and cardinality of the "by" grouping variable. For non-ordered factors or strings with a reasonable number of groups, this will inherit directly from the user's default `palette` (e.g., "R4"). In other cases, including ordered factors and high cardinality, the "Viridis" palette will be used instead. Note that a slightly restricted version of the "Viridis" palette—where extreme color values have been trimmed to improve visual perception—will be used for ordered factors and continuous variables. In the latter case of a continuous grouping variable, we also generate a gradient legend swatch.
- A convenience string corresponding to one of the many palettes listed by either `palette.pals()` or `hcl.pals()`. Note that the string can be case-insensitive (e.g., "Okabe-Ito" and "okabe-ito" are both valid).
- A palette-generating function. This can be "bare" (e.g., `palette.colors`) or "closed" with a set of named arguments (e.g., `palette.colors(palette = "Okabe-Ito", alpha = 0.5)`). Note that any unnamed arguments will be ignored and the key `n` argument, denoting the number of colours, will automatically be spliced in as the number of groups.

`legend` one of the following options:

- NULL (default), in which case the legend will be determined by the grouping variable. If there is no group variable (i.e., `by` is NULL) then no legend is drawn. If a grouping variable is detected, then an automatic legend is drawn to the *outer* right of the plotting area. Note that the legend title and categories will automatically be inferred from the `by` argument and underlying data.

- A convenience string indicating the legend position. The string should correspond to one of the position keywords supported by the base `legend` function, e.g. "right", "topleft", "bottom", etc. In addition, `tinyplot` supports adding a trailing exclamation point to these keywords, e.g. "right!", "topleft!", or "bottom!". This will place the legend *outside* the plotting area and adjust the margins of the plot accordingly. Finally, users can also turn off any legend printing by specifying "none".
- Logical value, where TRUE corresponds to the default case above (same effect as specifying NULL) and FALSE turns the legend off (same effect as specifying "none").
- A list or, equivalently, a dedicated `legend()` function with supported legend arguments, e.g. "bty", "horiz", and so forth.

`pch` plotting "character", i.e., symbol to use. Character, integer, or vector of length equal to the number of categories in the `by` variable. See `pch`. In addition, users can supply a special `pch = "by"` convenience argument, in which case the characters will automatically loop over the number groups. This automatic looping will begin at the global character value (i.e., `par("pch")`) and recycle as necessary.

`lty` line type. Character, integer, or vector of length equal to the number of categories in the `by` variable. See `lty`. In addition, users can supply a special `lty = "by"` convenience argument, in which case the line type will automatically loop over the number groups. This automatic looping will begin at the global line type value (i.e., `par("lty")`) and recycle as necessary.

`lwd` line width. Numeric scalar or vector of length equal to the number of categories in the `by` variable. See `lwd`. In addition, users can supply a special `lwd = "by"` convenience argument, in which case the line width will automatically loop over the number of groups. This automatic looping will be centered at the global line width value (i.e.,

`col` plotting color. Character, integer, or vector of length equal to the number of categories in the `by` variable. See `col`. Note that the default behaviour in `tinyplot` is to vary group colors along any variables declared in the `by` argument. Thus, specifying colors manually should not be necessary unless users wish to override the automatic colors produced by this grouping process. Typically, this would only be done if grouping features are deferred to some other graphical parameter (i.e., passing the "by" keyword to one of `pch`, `lty`, `lwd`, or `bg`; see below.)

`bg` background fill color for the open plot symbols 21:25 (see `points.default`), as well as ribbon and area plot types. Users can also supply either one of two special convenience arguments that will cause the background fill to inherit the automatic grouped coloring behaviour of `col`:

- `bg = "by"` will insert a background fill that inherits the main color mappings from `col`.
- `by = <numeric[0,1]>` (i.e., a numeric in the range `[0,1]`) will insert a background fill that inherits the main color mapping(s) from `col`, but with added alpha-transparency.

For both of these convenience arguments, note that the (grouped) `bg` mappings will persist even if the (grouped) `col` defaults are themselves overridden. This

can be useful if you want to preserve the grouped palette mappings by background fill but not boundary color, e.g. filled points. See examples.

`fill` alias for `bg`. If non-NULL values for both `bg` and `fill` are provided, then the latter will be ignored in favour of the former.

`alpha` a numeric in the range `[0,1]` for adjusting the alpha channel of the color palette, where 0 means transparent and 1 means opaque. Use fractional values, e.g. `0.5` for semi-transparency.

`cex` character expansion. A numerical vector (can be a single value) giving the amount by which plotting characters and symbols should be scaled relative to the default. Note that NULL is equivalent to 1.0, while NA renders the characters invisible.

`restore.par` a logical value indicating whether the `par` settings prior to calling `tinyplot` should be restored on exit. Defaults to FALSE, which makes it possible to add elements to the plot after it has been drawn. However, note the the outer margins of the graphics device may have been altered to make space for the `tinyplot` legend. Users can opt out of this persistent behaviour by setting to TRUE instead. See also get_saved_par for another option to recover the original `par` settings, as well as longer discussion about the trade-offs involved.

`xmin`, `xmax`, `ymin`, `ymax`
minimum and maximum coordinates of relevant area or interval plot types. Only used when the `type` argument is one of `"rect"` or `"segments"` (where all four min-max coordinates are required), or `"pointrange"`, `"errorbar"`, or `"ribbon"` (where only `ymin` and `ymax` required alongside `x`).

`add` logical. If TRUE, then elements are added to the current plot rather than drawing a new plot window. Note that the automatic legend for the added elements will be turned off. See also tinyplot_add, which provides a convenient wrapper around this functionality for layering on top of an existing plot without having to repeat arguments.

`draw` a function that draws directly on the plot canvas (before `x` and `y` are plotted). The `draw` argument is primarily useful for adding common elements to each facet of a faceted plot, e.g. `abline` or `text`. Note that this argument is somewhat experimental and that *no* internal checking is done for correctness; the provided argument is simply captured and evaluated as-is. See Examples.

`file` character string giving the file path for writing a plot to disk. If specified, the plot will not be displayed interactively, but rather sent to the appropriate external graphics device (i.e., `png`, `jpeg`, `pdf`, or `svg`). As a point of convenience, note that any global parameters held in `(t)par` are automatically carried over to the external device and don't need to be reset (in contrast to the conventional base R approach that requires manually opening and closing the device). The device type is determined by the file extension at the end of the provided path, and must be one of ".png", ".jpg" (".jpeg"), ".pdf", or ".svg". (Other file types may be supported in the future.) The file dimensions can be controlled by the corresponding `width` and `height` arguments below, otherwise will fall back to the `"file.width"` and `"file.height"` values held in `tpar` (i.e., both defaulting to 7 inches, and where the default resolution for bitmap files is also specified as 300 DPI).

`width` numeric giving the plot width in inches. Together with `height`, typically used in conjunction with the `file` argument above, overriding the default values held in `tpar("file.width", "file.height")`. If either `width` or `height` is specified, but a corresponding `file` argument is not provided as well, then a new interactive graphics device dimensions will be opened along the given dimensions. Note that this interactive resizing may not work consistently from within an IDE like RStudio that has an integrated graphics windows.

`height` numeric giving the plot height in inches. Same considerations as `width` (above) apply, e.g. will default to `tpar("file.height")` if not specified.

`empty` logical indicating whether the interior plot region should be left empty. The default is `FALSE`. Setting to `TRUE` has a similar effect to invoking `type = "n"` above, except that any legend artifacts owing to a particular plot type (e.g., lines for `type = "l"` or squares for `type = "area"`) will still be drawn correctly alongside the empty plot. In contrast,`type = "n"` implicitly assumes a scatterplot and so any legend will only depict points.

`xaxt`, `yaxt` character specifying the type of x-axis and y-axis, respectively. See `axes` for the possible values.

`flip` logical. Should the plot orientation be flipped, so that the y-axis is on the horizontal plane and the x-axis is on the vertical plane? Default is FALSE.

`xaxs`, `yaxs` character specifying the style of the interval calculation used for the x-axis and y-axis, respectively. See `par` for the possible values.

`formula` a `formula` that optionally includes grouping variable(s) after a vertical bar, e.g. `y ~ x | z`. One-sided formulae are also permitted, e.g. `~ y | z`. Multiple grouping variables can be specified in different ways, e.g. `y ~ x | z1:z2` or `y ~ x | z1 + z2`. (These two representations are treated as equivalent; both are parsed as `interaction(z1, z2)` internally.) Note that the `formula` and `x` arguments should not be specified in the same call.

`subset`, `na.action`, `drop.unused.levels`
arguments passed to `model.frame` when extracting the data from `formula` and `data`.

Details

Disregarding the enhancements that it supports, `tinyplot` tries as far as possible to mimic the behaviour and syntax logic of the original base `plot` function. Users should therefore be able to swap out existing `plot` calls for `tinyplot` (or its shorthand alias `plt`), without causing unexpected changes to the output.

Value

No return value, called for side effect of producing a plot.

Examples

```
#'
aq = transform(
  airquality,
```

```
  Month = factor(Month, labels = month.abb[unique(Month)])
)

# In most cases, `tinyplot` should be a drop-in replacement for regular
# `plot` calls. For example:

op = tpar(mfrow = c(1, 2))
plot(0:10, main = "plot")
tinyplot(0:10, main = "tinyplot")
tpar(op) # restore original layout

# Aside: `tinyplot::tpar()` is a (near) drop-in replacement for `par()`

# Unlike vanilla plot, however, tinyplot allows you to characterize groups
# using either the `by` argument or equivalent `|` formula syntax.

with(aq, tinyplot(Day, Temp, by = Month)) ## atomic method
tinyplot(Temp ~ Day | Month, data = aq) ## formula method

# (Notice that we also get an automatic legend.)

# You can also use the equivalent shorthand `plt()` alias if you'd like to
# save on a few keystrokes

plt(Temp ~ Day | Month, data = aq) ## shorthand alias

# Use standard base plotting arguments to adjust features of your plot.
# For example, change `pch` (plot character) to get filled points and `cex`
# (character expansion) to increase their size.

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  pch = 16,
  cex = 2
)

# We can add alpha transparency for overlapping points

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  pch = 16,
  cex = 2,
  alpha = 0.3
)

# To get filled points with a common solid background color, use an
# appropriate plotting character (21:25) and combine with one of the special
# `bg` convenience arguments.
tinyplot(
  Temp ~ Day | Month,
  data = aq,
```

```
  pch = 21, # use filled circles
  cex = 2,
  bg = 0.3, # numeric in [0,1] adds a grouped background fill with transparency
  col = "black" # override default color mapping; give all points a black border
)

# Converting to a grouped line plot is a simple matter of adjusting the
# `type` argument.

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "l"
)

# Similarly for other plot types, including some additional ones provided
# directly by tinyplot, e.g. density plots or internal plots (ribbons,
# pointranges, etc.)

tinyplot(
  ~ Temp | Month,
  data = aq,
  type = "density",
  fill = "by"
)

# Facet plots are supported too. Facets can be drawn on their own...

tinyplot(
  Temp ~ Day,
  facet = ~Month,
  data = aq,
  type = "area",
  main = "Temperatures by month"
)

# ... or combined/contrasted with the by (colour) grouping.

aq = transform(aq, Summer = Month %in% c("Jun", "Jul", "Aug"))
tinyplot(
  Temp ~ Day | Summer,
  facet = ~Month,
  data = aq,
  type = "area",
  palette = "dark2",
  main = "Temperatures by month and season"
)

# Users can override the default square window arrangement by passing `nrow`
# or `ncol` to the helper facet.args argument. Note that we can also reduce
# axis label repetition across facets by turning the plot frame off.

tinyplot(
```

```
  Temp ~ Day | Summer,
  facet = ~Month, facet.args = list(nrow = 1),
  data = aq,
  type = "area",
  palette = "dark2",
  frame = FALSE,
  main = "Temperatures by month and season"
)

# Use a two-sided formula to arrange the facet windows in a fixed grid.
# LHS -> facet rows; RHS -> facet columns

aq$hot = ifelse(aq$Temp >= 75, "hot", "cold")
aq$windy = ifelse(aq$Wind >= 15, "windy", "calm")
tinyplot(
  Temp ~ Day,
  facet = windy ~ hot,
  data = aq
)

# To add common elements to each facet, use the `draw` argument

tinyplot(
  Temp ~ Day,
  facet = windy ~ hot,
  data = aq,
  draw = abline(h = 75, lty = 2, col = "hotpink")
)

# The (automatic) legend position and look can be customized using
# appropriate arguments. Note the trailing "!" in the `legend` position
# argument below. This tells `tinyplot` to place the legend _outside_ the plot
# area.

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "l",
  legend = legend("bottom!", title = "Month of the year", bty = "o")
)

# The default group colours are inherited from either the "R4" or "Viridis"
# palettes, depending on the number of groups. However, all palettes listed
# by `palette.pals()` and `hcl.pals()` are supported as convenience strings,
# or users can supply a valid palette-generating function for finer control

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "l",
  palette = "tableau"
)
```

```
# It's possible to customize the look of your plots by setting graphical
# parameters (e.g., via `(t)par`)... But a more convenient way is to just use
# built-in themes (see `?tinytheme`).

tinytheme("clean2")
tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "b",
  alpha = 0.5,
  main = "Daily temperatures by month",
  sub = "Brought to you by tinyplot"
)
# reset the theme
tinytheme()

# For more examples and a detailed walkthrough, please see the introductory
# tinyplot tutorial available online:
# https://grantmcdermott.com/tinyplot/vignettes/introduction.html
```

`tinyplot_add` *Add new elements to the current* `tinyplot`

Description

This convenience function grabs the preceding `tinyplot` call and updates it with any new arguments that have been explicitly provided by the user. It then injects `add=TRUE` and evaluates the updated call, thereby drawing a new layer on top of the existing plot. `plt_add()` is a shorthand alias for `tinyplot_add()`.

Usage

```
tinyplot_add(...)

plt_add(...)
```

Arguments

`...`	All named arguments override arguments from the previous calls. Arguments not supplied to tinyplot_add remain unchanged from the previous call.

Value

No return value, called for side effect of producing a plot.

Limitations

- Currently, `tinyplot_add` only works reliably if you are adding to a plot that was originally constructed with the tinyplot.formula method (and passed an appropriate `data` argument). In contrast, we cannot guarantee that using `tinyplot_add` will work correctly if your original plot was constructed with the atomic tinyplot.default method. The reason has to do with potential environment mismatches. (An exception is thus if your plot arguments (`x`, `y`, etc.) are attached to your global R environment.)
- Automatic legends for the added elements will be turned off.

Examples

```
library(tinyplot)

tinyplot(Sepal.Width ~ Sepal.Length | Species,
  facet = ~Species,
  data = iris)

tinyplot_add(type = "lm") ## or : plt_add(type = "lm")

## Note: the previous function is equivalent to (but much more convenient
## than) re-writing the full call with the new type and `add=TRUE`:

# tinyplot(Sepal.Width ~ Sepal.Length | Species,
#          facet = ~Species,
#          data = iris,
#          type = "lm",
#          add = TRUE)
```

`tinytheme` *Set or Reset Plot Themes for* `tinyplot`

Description

The `tinytheme` function sets or resets the theme for plots created with `tinyplot`. Themes control the appearance of plots, such as text alignment, font styles, axis labels, and even dynamic margin adjustment to reduce whitespace.

Usage

```
tinytheme(
 theme = c("default", "basic", "clean", "clean2", "bw", "classic", "minimal", "ipsum",
    "dark", "ridge", "ridge2", "tufte", "void"),
  ...
)
```

Arguments

`theme` A character string specifying the name of the theme to apply. Themes are arranged in an approximate hierarchy, adding or subtracting elements in the order presented below. Note that several themes are *dynamic*, in the sense that they attempt to reduce whitespace in a way that is responsive to the length of axes labels, tick marks, etc. These dynamic plots are marked with an asterisk (*) below.

- `"default"`: inherits the user's default base graphics settings.
- `"basic"`: light modification of `"default"`, only adding filled points, a panel background grid, and light gray background to facet titles.
- `"clean"` (*): builds on `"basic"` by moving the subtitle above the plotting area, adding horizontal axis labels, employing tighter default plot margins and title gaps to reduce whitespace, and setting different default palettes ("Tableau 10" for discrete colors and "agSunset" for gradient colors). The first of our dynamic themes and the foundation for several derivative themes that follow below.
- `"clean2"` (*): removes the plot frame (box) from `"clean"`.
- `"classic"` (*): connects the axes in a L-shape, but removes the other top and right-hand edges of the plot frame (box). Also sets the "Okabe-Ito" palette as a default for discrete colors. Inspired by the **ggplot2** theme of the same name.
- `"bw"` (*): similar to `"clean"`, except uses thinner lines for the plot frame (box), solid grid lines, and sets the "Okabe-Ito" palette as a default for discrete colors. Inspired by the **ggplot2** theme of the same name.
- `"minimal"` (*): removes the plot frame (box) from `"bw"`, as well as the background for facet titles. Inspired by the **ggplot2** theme of the same name.
- `"ipsum"` (*): similar to `"minimal"`, except subtitle is italicised and axes titles are aligned to the far edges. Inspired by the **hrbrthemes** theme of the same name for **ggplot2**.
- `"dark"` (*): similar to `"minimal"`, but set against a dark background with foreground and a palette colours lightened for appropriate contrast.
- `"ridge"` (*): a specialized theme for ridge plots (see `type_ridge()`). Builds off of `"clean"`, but adds ridge-specific tweaks (e.g. default "Zissou 1" palette for discrete colors, solid horizontal grid lines, and minor adjustments to y-axis labels). Not recommended for non-ridge plots.
- `"ridge2"` (*): removes the plot frame (box) from `"ridge"`, but retains the x-axis line. Again, not recommended for non-ridge plots.
- `"tufte"`: floating axes and minimalist plot artifacts in the style of Edward Tufte.
- `"void"`: switches off all axes, titles, legends, etc.

`...` Named arguments to override specific theme settings. These arguments are passed to `tpar()` and take precedence over the predefined settings in the selected theme.

Details

Sets a list of graphical parameters using `tpar()`

To reset the theme to default settings (no customization), call `tinytheme()` without arguments.

Caveat emptor: Themes are a somewhat experimental feature of `tinyplot`. While we feel confident that themes should work as expected for most "standard" cases, there may be some sharp edges. Please report any unexpected behaviour to our GitHub repo: https://github.com/grantmcdermott/tinyplot/issues

Known current limitations include:

- Themes do not work well when `legend = "top!"`.
- Dynamic margin spacing does not account for multi-line strings (e.g., axes or main titles that contain "\n").

Value

The function returns nothing. It is called for its side effects.

See Also

`tpar` which does the heavy lifting under the hood.

Examples

```
# Reusable plot function
p = function() tinyplot(
  lat ~ long | depth, data = quakes,
  main = "Earthquakes off Fiji",
  sub = "Data courtesy of the Harvard PRIM-H project"
)
p()

# Set a theme
tinytheme("bw")
p()

# Try a different theme
tinytheme("dark")
p()

# Customize the theme by overriding default settings
tinytheme("bw", fg = "green", font.main = 2, font.sub = 3, family = "Palatino")
p()

# Aside: One or two specialized themes are only meant for certain plot types
tinytheme("ridge2")
tinyplot(I(cut(lat, 10)) ~ depth, data = quakes, type = "ridge")

# Reset the theme
tinytheme()
p()
```

```
# Themes showcase
## We'll use a slightly more intricate plot (long y-axis labs and facets)
## to demonstrate dynamic margin adjustment etc.

thms = eval(formals(tinytheme)$theme)

for (thm in thms) {
  tinytheme(thm)
  tinyplot(
    I(Sepal.Length*1e4) ~ Petal.Length | Species, facet = "by", data = iris,
    main = "Demonstration of tinyplot themes",
    sub = paste0('tinytheme("', thm, '")')
  )
}

# Reset
tinytheme()
```

`tpar` *Set or query graphical parameters*

Description

Extends `par`, serving as a (near) drop-in replacement for setting or querying graphical parameters. The key differences is that, beyond supporting the standard group of R graphical parameters in `par`, `tpar` also supports additional graphical parameters that are provided by `tinyplot`. Similar to `par`, parameters are set by passing appropriate `key = value` argument pairs, and multiple parameters can be set or queried at the same time.

Usage

```
tpar(..., hook = FALSE)
```

Arguments

`...`	arguments of the form `key = value`. This includes all of the parameters typically supported by `par`, as well as the `tinyplot`-specific ones described in the 'Graphical Parameters' section below.
`hook`	Logical. If `TRUE`, base graphical parameters persist across plots via a hook applied before each new plot (see `?setHook`).

Details

The `tinyplot`-specific parameters are saved in an internal environment called `.tpar` for performance and safety reasons. However, they can also be set at package load time via `options`, which may prove convenient for users that want to enable different default behaviour at startup (e.g.,

through an .Rprofile file). These options all take a tinyplot_* prefix, e.g. options(tinyplot_grid = TRUE, tinyplot_facet.bg = "grey90").

For their part, any "base" graphical parameters are caught dynamically and passed on to par as appropriate. Technically, only parameters that satisfy par(..., no.readonly = TRUE) are evaluated.

However, note the important distinction: tpar only evaluates parameters from par if they are passed *explicitly* by the user. This means that tpar should not be used to capture the (invisible) state of a user's entire set of graphics parameters, i.e. tpar() != par(). If you want to capture the *all* existing graphics settings, then you should rather use par() instead.

Value

When parameters are set, their previous values are returned in an invisible named list. Such a list can be passed as an argument to tpar to restore the parameter values.

When just one parameter is queried, the value of that parameter is returned as (atomic) vector. When two or more parameters are queried, their values are returned in a list, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns a vector.

Additional Graphical Parameters

- adj.xlab: Numeric value between 0 and 1 controlling the alignment of the x-axis label.
- adj.ylab: Numeric value between 0 and 1 controlling the alignment of the y-axis label.
- dynmar: Logical indicating whether tinyplot should attempt dynamic adjustment of margins to reduce whitespace and/or account for spacing of text elements (e.g., long horizontal y-axis labels). Note that this parameter is tightly coupled to internal tinythemes() logic and should *not* be adjusted manually unless you really know what you are doing or don't mind risking unintended consequences to your plot.
- facet.bg: Character or integer specifying the facet background colour. If an integer, will correspond to the user's default colour palette (see palette). Passed to rect. Defaults to NULL (none).
- facet.border: Character or integer specifying the facet border colour. If an integer, will correspond to the user's default colour palette (see palette). Passed to rect. Defaults to NA (none).
- facet.cex: Expansion factor for facet titles. Defaults to 1.
- facet.col: Character or integer specifying the facet text colour. If an integer, will correspond to the user's default global colour palette (see palette). Defaults to NULL, which is equivalent to "black".
- facet.font: An integer corresponding to the desired font face for facet titles. For most font families and graphics devices, one of four possible values: 1 (regular), 2 (bold), 3 (italic), or 4 (bold italic). Defaults to NULL, which is equivalent to 1 (i.e., regular).
- file.height: Numeric specifying the height (in inches) of any plot that is written to disk using the tinyplot(..., file = X) argument. Defaults to 7.
- file.res: Numeric specifying the resolution (in dots per square inch) of any plot that is written to disk in bitmap format (i.e., PNG or JPEG) using the tinyplot(..., file = X) argument. Defaults to 300.

- `file.width`: Numeric specifying the width (in inches) of any plot that is written to disk using the `tinyplot(..., file = X)` argument. Defaults to `7`.
- `fmar`: A numeric vector of form `c(b,l,t,r)` for controlling the (base) margin padding, in terms of lines, between the individual facets in a faceted plot. Defaults to `c(1,1,1,1)`. If more than three facets are detected, the `fmar` parameter is scaled by 0.75 to reduce excess whitespace. For 2x2 plots, the padding better matches the `cex` expansion logic of base graphics.
- `grid.col`: Character or (integer) numeric that specifies the color of the panel grid lines. Defaults to `"lightgray"`.
- `grid.lty`: Character or (integer) numeric that specifies the line type of the panel grid lines. Defaults to `"dotted"`.
- `grid.lwd`: Non-negative numeric giving the line width of the panel grid lines. Defaults to `1`.
- `grid`: Logical indicating whether a background panel grid should be added to plots automatically. Defaults to `NULL`, which is equivalent to `FALSE`.
- `lmar`: A numeric vector of form `c(inner, outer)` that gives the margin padding, in terms of lines, around the automatic `tinyplot` legend. Defaults to `c(1.0, 0.1)`. The inner margin is the gap between the legend and the plot region, and the outer margin is the gap between the legend and the edge of the graphics device.
- `palette.qualitative`: Palette for qualitative colors. See the `palette` argumetn in `?tinyplot`.
- `palette.sequential`: Palette for sequential colors. See the `palette` argumetn in `?tinyplot`.
- `ribbon.alpha`: Numeric factor in the range `[0,1]` for modifying the opacity alpha of "ribbon" and "area" type plots. Default value is `0.2`.

See Also

`graphics::par` which `tpar` builds on top of. `get_saved_par` is a convenience function for retrieving graphical parameters at different stages of a `tinyplot` call (and used for internal accounting purposes). `tinytheme` allows users to easily set a group of graphics parameters in a single function call, according to a variety of predefined themes.

Examples

```
# Return a list of existing base and tinyplot graphic params
tpar("las", "pch", "facet.bg", "facet.cex", "grid")

# Simple facet plot with these default values
tinyplot(mpg ~ wt, data = mtcars, facet = ~am)

# Set params to something new. Similar to graphics::par(), note that we save
# the existing values at the same time by assigning to an object.
op = tpar(
  las       = 1,
  pch       = 2,
  facet.bg  = "grey90",
  facet.cex = 2,
  grid      = TRUE
)
```

```
# Re-plot with these new params
tinyplot(mpg ~ wt, data = mtcars, facet = ~am)

# Reset back to original values
tpar(op)

# Important: tpar() only evalutes parameters that have been passed explicitly
#   by the user. So it it should not be used to query and set (restore)
#   parameters that weren't explicitly requested, i.e. tpar() != par().

# Note: The tinyplot-specific parameters can also be be set via `options`
#   with a `tinyplot_*` prefix, which can be convenient for enabling
#   different default behaviour at startup time (e.g., via an .Rprofile
#   file). Example:
# options(tinyplot_grid = TRUE, tinyplot_facet.bg = "grey90")
```

`type_abline` *Add straight lines to a plot*

Description

Add straight lines to a plot

Usage

```
type_abline(a = 0, b = 1)
```

Arguments

a, b	the intercept and slope, single values.

Examples

```
mod = lm(mpg ~ hp, data = mtcars)
y = mtcars$mpg
yhat = predict(mod)
tinyplot(y, yhat, xlim = c(0, 40), ylim = c(0, 40))
tinyplot_add(type = type_abline(a = 0, b = 1))
```

type_area *Ribbon and area plot types*

Description

Type constructor functions for producing polygon ribbons, which define a y interval (usually spanning from ymin to ymax) for each x value. Area plots are a special case of ribbon plot where ymin is set to 0 and ymax is set to y.

Usage

```
type_area(alpha = NULL)

type_ribbon(alpha = NULL)
```

Arguments

alpha	numeric value between 0 and 1 specifying the opacity of ribbon shading If no alpha value is provided, then will default to tpar("ribbon.alpha") (i.e., probably 0.2 unless this has been overridden by the user in their global settings.)

Examples

```
x = 1:100/10
y = sin(x)

#
## Ribbon plots

# "ribbon" convenience string
tinyplot(x = x, ymin = y-1, ymax = y+1, type = "ribbon")
# Same result with type_ribbon()
tinyplot(x = x, ymin = y-1, ymax = y+1, type = type_ribbon())

# y will be added as a line if it is specified
tinyplot(x = x, y = y, ymin = y-1, ymax = y+1, type = "ribbon")

#
## Area plots

# "area" type convenience string
tinyplot(x, y, type = "area")

# Same result with type_area()
tinyplot(x, y, type = type_area())

# Area plots are often used for time series charts
tinyplot(AirPassengers, type = "area")
```

type_boxplot *Boxplot type*

Description

Type function for producing box-and-whisker plots. Arguments are passed to boxplot, although tinyplot scaffolding allows added functionality such as grouping and faceting. Box-and-whisker plots are the default plot type if x is a factor and y is numeric.

Usage

```
type_boxplot(
  range = 1.5,
  width = NULL,
  varwidth = FALSE,
  notch = FALSE,
  outline = TRUE,
  boxwex = 0.8,
  staplewex = 0.5,
  outwex = 0.5
)
```

Arguments

range	this determines how far the plot whiskers extend out from the box. If range is positive, the whiskers extend to the most extreme data point which is no more than range times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.
width	a vector giving the relative widths of the boxes making up the plot.
varwidth	if varwidth is TRUE, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
notch	if notch is TRUE, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ (Chambers et al., 1983, p. 62). See boxplot.stats for the calculations used.
outline	if outline is not true, the outliers are not drawn (as points whereas S+ uses lines).
boxwex	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.
staplewex	staple line width expansion, proportional to box width.
outwex	outlier line width expansion, proportional to box width.

Examples

```
# "boxplot" type convenience string
tinyplot(count ~ spray, data = InsectSprays, type = "boxplot")

# Note: Specifying the type here is redundant. Like base plot, tinyplot
# automatically produces a boxplot if x is a factor and y is numeric
tinyplot(count ~ spray, data = InsectSprays)

# Grouped boxplot example
tinyplot(len ~ dose | supp, data = ToothGrowth, type = "boxplot")

# Use `type_boxplot()` to pass extra arguments for customization
tinyplot(
  len ~ dose | supp, data = ToothGrowth, lty = 1,
  type = type_boxplot(boxwex = 0.3, staplewex = 0, outline = FALSE)
)
```

---

type_density *Density plot type*

Description

Type function for density plots.

Usage

```
type_density(
  bw = "nrd0",
  joint.bw = c("mean", "full", "none"),
  adjust = 1,
 kernel = c("gaussian", "epanechnikov", "rectangular", "triangular", "biweight",
    "cosine", "optcosine"),
  n = 512,
  alpha = NULL
)
```

Arguments

bw	the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below.) bw can also be a character string giving a rule to choose the bandwidth. See bw.nrd. The default, "nrd0", has remained the default for historical and compatibility reasons, rather than as a general recommendation, where e.g., "SJ" would rather fit, see also Venables and Ripley (2002). The specified (or computed) value of bw is multiplied by adjust.

`joint.bw` character string indicating whether (and how) the smoothing bandwidth should be computed from the joint data distribution when there are multiple subgroups. The options are `"mean"` (the default), `"full"`, and `"none"`. Also accepts a logical argument, where `TRUE` maps to `"mean"` and `FALSE` maps to `"none"`. See the "Bandwidth selection" section below for a discussion of practical considerations.

`adjust` the bandwidth used is actually `adjust*bw`. This makes it easy to specify values like 'half the default' bandwidth.

`kernel` a character string giving the smoothing kernel to be used. This must partially match one of `"gaussian"`, `"rectangular"`, `"triangular"`, `"epanechnikov"`, `"biweight"`, `"cosine"` or `"optcosine"`, with default `"gaussian"`, and may be abbreviated to a unique prefix (single letter).

`"cosine"` is smoother than `"optcosine"`, which is the usual 'cosine' kernel in the literature and almost MSE-efficient. However, `"cosine"` is the version used by S.

`n` the number of equally spaced points at which the density is to be estimated. When `n > 512`, it is rounded up to a power of 2 during the calculations (as `fft` is used) and the final result is interpolated by `approx`. So it almost always makes sense to specify `n` as a power of two.

`alpha` numeric value between 0 and 1 specifying the opacity of ribbon shading If no `alpha` value is provided, then will default to `tpar("ribbon.alpha")` (i.e., probably `0.2` unless this has been overridden by the user in their global settings.)

Details

The algorithm used in `density.default` disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.

The statistical properties of a kernel are determined by $\sigma_K^2 = \int t^2 K(t) dt$ which is always $= 1$ for our kernels (and hence the bandwidth `bw` is the standard deviation of the kernel) and $R(K) = \int K^2(t) dt$.
MSE-equivalent bandwidths (for different kernels) are proportional to $\sigma_K R(K)$ which is scale invariant and for our kernels equal to $R(K)$. This value is returned when `give.Rkern = TRUE`. See the examples for using exact equivalent bandwidths.

Infinite values in `x` are assumed to correspond to a point mass at `+/-Inf` and the density estimate is of the sub-density on `(-Inf, +Inf)`.

Bandwidth selection

While the choice of smoothing bandwidth will always stand to affect a density visualization, it gains an added importance when multiple densities are drawn simultaneously (e.g., for subgroups with respect to `by` or `facet`). Allowing each subgroup to compute its own separate bandwidth independently offers greater flexibility in capturing the unique characteristics of each subgroup, particularly when distributions differ substantially in location and/or scale. However, this approach may overemphasize small random variations and make it harder to visually compare densities across subgroups. Hence, it is often useful to employ the same ("joint") bandwidth across all subgroups. The following strategies are available via the `joint.bw` argument:

- The default `joint.bw = "mean"` first computes the individual bandwidths for each group but then computes their mean, weighted by the number of observations in each group. This will work well when all groups have similar amounts of scatter (similar variances), even when they have potentially rather different locations. The weighted averaging stabilizes potential fluctuations in the individual bandwidths, especially when some subgroups are rather small.
- Alternatively, `joint.bw = "full"` can be used to compute the joint bandwidth from the full joint distribution (merging all groups). This will yield an even more robust bandwidth, especially when the groups overlap substantially (i.e., have similar locations and scales). However, it may lead to too large bandwidths and thus too much smoothing, especially when the locations of the groups differ substantially.
- Finally, `joint.bw = "none"` disables the joint bandwidth so that each group just employs its individual bandwidth. This is often the best choice if the amounts of scatter differ substantially between the groups, thus necessitating different amounts of smoothing.

Titles

This tinyplot method for density plots differs from the base `plot.density` function in its treatment of titles. The x-axis title displays only the variable name, omitting details about the number of observations and smoothing bandwidth. Additionally, the main title is left blank by default for a cleaner appearance.

Examples

```
# "density" type convenience string
tinyplot(~Sepal.Length, data = iris, type = "density")

# grouped density example
tinyplot(~Sepal.Length | Species, data = iris, type = "density")

# use `bg = "by"` (or, equivalent `fill = "by"`) to get filled densities
tinyplot(~Sepal.Length | Species, data = iris, type = "density", fill = "by")

# use `type_density()` to pass extra arguments for customization
tinyplot(
  ~Sepal.Length | Species, data = iris,
  type = type_density(bw = "SJ"),
  main = "Bandwidth computed using Sheather & Jones (1991)"
)

# The default for grouped density plots is to use the mean of the
# individual subgroup bandwidths (weighted by group size) as the
# joint bandwidth. Alternatively, the bandwidth from the "full"
# data or separate individual bandwidths ("none") can be used.
tinyplot(~Sepal.Length | Species, data = iris,
    ylim = c(0, 1.25), type = "density")        # mean (default)
tinyplot_add(joint.bw = "full", lty = 2)        # full data
tinyplot_add(joint.bw = "none", lty = 3)        # none (individual)
legend("topright", c("Mean", "Full", "None"), lty = 1:3, bty = "n", title = "Joint BW")
```

type_errorbar *Error bar and pointrange plot types*

Description

Type function(s) for producing error bar and pointrange plots.

Usage

```
type_errorbar(length = 0.05)

type_pointrange()
```

Arguments

length	length of the edges of the arrow head (in inches).

Examples

```
mod = lm(mpg ~ wt * factor(am), mtcars)
coefs = data.frame(names(coef(mod)), coef(mod), confint(mod))
colnames(coefs) = c("term", "est", "lwr", "upr")

op = tpar(pch = 19)

# "errorbar" and "pointrange" type convenience strings
with(
  coefs,
  tinyplot(x = term, y = est, ymin = lwr, ymax = upr, type = "errorbar")
)
with(
  coefs,
  tinyplot(x = term, y = est, ymin = lwr, ymax = upr, type = "pointrange")
)

# Use `type_errorbar()` to pass extra arguments for customization
with(
  coefs,
  tinyplot(x = term, y = est, ymin = lwr, ymax = upr, type = type_errorbar(length = 0.2))
)

tpar(op)
```

type_function *Plot a function*

Description

Plot a function

Usage

```
type_function(fun = dnorm, args = list(), n = 101, ...)
```

Arguments

fun	Function of x to plot. Defaults to dnorm.
args	List of additional arguments to be passed to fun.
n	Number of points to interpolate on the x axis.
...	Additional arguments are passed to the lines() function, ex: type="p", col="pink".

Details

When using type_function() in a tinyplot() call, the x value indicates the range of values to plot on the x-axis.

Examples

```
# Plot the normal density
tinyplot(x = -4:4, type = type_function(dnorm))

# Extra arguments for the function to plot
tinyplot(x = -1:10, type = type_function(
  fun = dnorm, args = list(mean = 3)
))

# Additional arguments are passed to the `lines()` function.
tinyplot(x = -4:4, type = type_function(
  fun = dnorm,
  col = "pink", type = "p", pch = 3
))

# Custom function example
# (Here using the `\()` anonymous function syntax introduced in R 4.1.0)
tinyplot(x = -4:4, type = type_function(fun = \(x) 0.5 * exp(-abs(x))))
```

type_glm *Generalized linear model plot type*

Description

Type function for plotting a generalized model fit. Arguments are passed to `glm`.

Usage

```
type_glm(family = "gaussian", se = TRUE, level = 0.95, type = "response")
```

Arguments

`family`	a description of the error distribution and link function to be used in the model. For `glm` this can be a character string naming a family function, a family function or the result of a call to a family function. For `glm.fit` only the third option is supported. (See `family` for details of family functions.)
`se`	logical. If TRUE, confidence intervals are drawn.
`level`	the confidence level required.
`type`	character, partial matching allowed. Type of weights to extract from the fitted model object. Can be abbreviated.

Examples

```
# "glm" type convenience string
tinyplot(am ~ mpg, data = mtcars, type = "glm")

# Use `type_glm()` to pass extra arguments for customization
tinyplot(am ~ mpg, data = mtcars, type = type_glm(family = "binomial"))
```

type_histogram *Histogram plot type*

Description

Type function for histogram plots. `type_hist` is an alias for `type_histogram`.

Usage

```
type_histogram(
  breaks = "Sturges",
  freq = NULL,
  right = TRUE,
  free.breaks = FALSE,
  drop.zeros = TRUE
```

```
)

type_hist(
  breaks = "Sturges",
  freq = NULL,
  right = TRUE,
  free.breaks = FALSE,
  drop.zeros = TRUE
)
```

Arguments

breaks	Passed to hist. One of: • a vector giving the breakpoints between histogram cells, • a function to compute the vector of breakpoints, • a single number giving the number of cells for the histogram, • a character string naming an algorithm to compute the number of cells (see 'Details' of hist), • a function to compute the number of cells. In the last three cases the number is a suggestion only; as the breakpoints will be set to pretty values, the number is limited to 1e6 (with a warning if it was larger). If breaks is a function, the x vector is supplied to it as the only argument (and the number of breaks is only limited by the amount of available memory).
freq	logical; if TRUE, the histogram graphic is a representation of frequencies, the counts component of the result; if FALSE, probability densities, component density, are plotted (so that the histogram has a total area of one). Defaults to TRUE *if and only if* breaks are equidistant (and probability is not specified).
right	logical; if TRUE, the histogram cells are right-closed (left open) intervals.
free.breaks	Logical indicating whether the breakpoints should be computed separately for each group or facet? Default is FALSE, meaning that the breakpoints are computed from the full dataset; thus ensuring common bin widths across each group/facet. Can also use free as an acceptable argument alias. Ignored if there are no groups and/or facets.
drop.zeros	Logical indicating whether bins with zero counts should be dropped before plotting. Default is TRUE. Note that switching to FALSE may interfere with faceted plot behaviour if facet.args = list(free), since the x variable is effectively recorded over the full range of the x-axis (even if it does not extend over this range for every group).

Examples

```
# "histogram"/"hist" type convenience string(s)
tinyplot(Nile, type = "histogram")

# Use `type_histogram()` to pass extra arguments for customization
tinyplot(Nile, type = type_histogram(breaks = 30))
```

```
tinyplot(Nile, type = type_histogram(breaks = 30, freq = FALSE))
# etc.

# Grouped histogram example
tinyplot(
    ~Petal.Width | Species,
    type = "histogram",
    data = iris
)

# Faceted version
tinyplot(
    ~Petal.Width, facet = ~Species,
    type = "histogram",
    data = iris
)

# For visualizing faceted histograms across varying scales, you may also wish
# to impose free histogram breaks too (i.e., calculate breaks separately for
# each group). Compare:

# free facet scales + shared histogram breaks, versus...
tinyplot(
    ~Petal.Width, facet = ~Species,
    facet.args = list(free = TRUE),
    type = type_histogram(),
    data = iris
)
# ... free facet scales + free histogram breaks
tinyplot(
    ~Petal.Width, facet = ~Species,
    facet.args = list(free = TRUE),
    type = type_histogram(free = TRUE),
    data = iris
)
```

`type_hline` *Trace a horizontal line on the plot*

Description

Trace a horizontal line on the plot

Usage

```
type_hline(h = 0)
```

Arguments

h	y-value(s) for horizontal line(s). Numeric of length 1 or equal to the number of facets.

Examples

```
tinyplot(mpg ~ hp | factor(cyl), facet = ~ factor(cyl), data = mtcars)
tinyplot_add(type = type_hline(h = 12), col = "pink", lty = 3, lwd = 3)
```

---

type_jitter *Jittered points plot type*

Description

Type function for plotting jittered points. Arguments are passed to `jitter`.

Usage

```
type_jitter(factor = 1, amount = NULL)
```

Arguments

factor	numeric.
amount	numeric; if positive, used as *amount* (see below), otherwise, if = 0 the default is factor * z/50. Default (NULL): factor * d/5 where d is about the smallest difference between x values.

Details

The result, say r, is r <- x + runif(n, -a, a) where n <- length(x) and a is the amount argument (if specified).

Let z <- max(x) - min(x) (assuming the usual case). The amount a to be added is either provided as *positive* argument amount or otherwise computed from z, as follows:

If amount == 0, we set a <- factor * z/50 (same as S).

If amount is NULL (*default*), we set a <- factor * d/5 where *d* is the smallest difference between adjacent unique (apart from fuzz) x values.

Examples

```
# "jitter" type convenience string
tinyplot(Sepal.Length ~ Species, data = iris, type = "jitter")

# Use `type_jitter()` to pass extra arguments for customization
tinyplot(Sepal.Length ~ Species, data = iris, type = type_jitter(factor = 0.5))
```

type_lines *Lines plot type*

Description

Type function for plotting lines.

Usage

```
type_lines(type = "l")
```

Arguments

type	1-character string giving the type of plot desired. The following values are possible, for details, see plot: "p" for points, "l" for lines, "b" for both points and lines, "c" for empty points joined by lines, "o" for overplotted points and lines, "s" and "S" for stair steps and "h" for histogram-like vertical lines. Finally, "n" does not produce any points or lines.

Examples

```
# "l" type convenience character string
tinyplot(circumference ~ age | Tree, data = Orange, type = "l")

# Use `type_lines()` to pass extra arguments for customization
tinyplot(circumference ~ age | Tree, data = Orange, type = type_lines(type = "s"))
```

type_lm *Linear model plot type*

Description

Type function for plotting a linear model fit. Arguments are passed to lm.

Usage

```
type_lm(se = TRUE, level = 0.95)
```

Arguments

se	logical. If TRUE, confidence intervals are drawn.
level	the confidence level required.

Examples

```
# "lm" type convenience string
tinyplot(Sepal.Width ~ Petal.Width, data = iris, type = "lm")

# Grouped model fits (here: illustrating an example of Simpson's paradox)
tinyplot(Sepal.Width ~ Petal.Width | Species, data = iris, type = "lm")
tinyplot_add(type = "p")

# Use `type_lm()` to pass extra arguments for customization
tinyplot(Sepal.Width ~ Petal.Width, data = iris, type = type_lm(level = 0.8))
```

---

type_loess *Local polynomial regression plot type*

Description

Type function for plotting a LOESS (LOcal regrESSion) fit. Arguments are passed to `loess`.

Usage

```
type_loess(
  span = 0.75,
  degree = 2,
  family = "gaussian",
  control = loess.control(),
  se = TRUE,
  level = 0.95
)
```

Arguments

span	the parameter α which controls the degree of smoothing.
degree	the degree of the polynomials to be used, normally 1 or 2. (Degree 0 is also allowed, but see the 'Note'.)
family	if `"gaussian"` fitting is by least-squares, and if `"symmetric"` a re-descending M estimator is used with Tukey's biweight function. Can be abbreviated.
control	control parameters: see `loess.control`.
se	logical. If `TRUE` (the default), confidence intervals are drawn.
level	the confidence level required if `se = TRUE`. Default is 0.95.

Examples

```
# "loess" type convenience string
tinyplot(dist ~ speed, data = cars, type = "loess")

# Use `type_loess()` to pass extra arguments for customization
tinyplot(dist ~ speed, data = cars, type = type_loess(span = 0.5, degree = 1))
```

type_points *Points plot type*

Description

Type function for plotting points, i.e. a scatter plot.

Usage

```
type_points()
```

Examples

```
# "p" type convenience character string
tinyplot(Sepal.Length ~ Petal.Length, data = iris, type = "p")

# Same result with type_points()
tinyplot(Sepal.Length ~ Petal.Length, data = iris, type = type_points())

# Note: Specifying the type here is redundant. Like base plot, tinyplot
# automatically produces a scatter plot if x and y are numeric
tinyplot(Sepal.Length ~ Petal.Length, data = iris)

# Grouped scatter plot example
tinyplot(Sepal.Length ~ Petal.Length | Species, data = iris)

# Continuous grouping (with gradient legend)
tinyplot(Sepal.Length ~ Petal.Length | Sepal.Width, data = iris, pch = 19)
```

type_polygon *Polygon plot type*

Description

Type function for plotting polygons. Arguments are passed to polygon.

Usage

```
type_polygon(density = NULL, angle = 45)
```

Arguments

density	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. A zero value of density means no shading nor filling whereas negative values and NA suppress shading (and so allow color filling).
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise).

Examples

```
# "polygon" type convenience character string
tinyplot(1:9, c(2,1,2,1,NA,2,1,2,1), type = "polygon")

# Use `type_polygon()` to pass extra arguments for customization
tinyplot(1:9, c(2,1,2,1,NA,2,1,2,1), type = type_polygon(density = c(10, 20)))
```

`type_polypath` *Polypath polygon type*

Description

Type function for plotting polygons. Arguments are passed to `polypath`.

Usage

```
type_polypath(rule = "winding")
```

Arguments

`rule`	character value specifying the path fill mode: either `"winding"` or `"evenodd"`.

Examples

```
# "polypath" type convenience character string
tinyplot(
    c(.1, .1, .6, .6, NA, .4, .4, .9, .9),
    c(.1, .6, .6, .1, NA, .4, .9, .9, .4),
    type = "polypath", fill = "grey"
)

# Use `type_polypath()` to pass extra arguments for customization
tinyplot(
    c(.1, .1, .6, .6, NA, .4, .4, .9, .9),
    c(.1, .6, .6, .1, NA, .4, .9, .9, .4),
    type = type_polypath(rule = "evenodd"), fill = "grey"
)
```

type_qq *Quantile-Quantile plot (QQ)*

Description

Plots the theoretical quantiles of x on the horizontal axis against observed values of x on the vertical axis.

Usage

```
type_qq(distribution = qnorm)
```

Arguments

distribution	Distribution function to use.

Examples

```
tinyplot(~mpg, data = mtcars, type = type_qq())

# suppress the line
tinyplot(~mpg, data = mtcars, lty = 0, type = type_qq())
```

type_rect *Rectangle plot type*

Description

Type function for plotting rectangles.

Usage

```
type_rect()
```

Details

Contrary to base rect, rectangles in tinyplot must be specified using the xmin, ymin,xmax, and ymax arguments.

Examples

```
i = 4*(0:10)

# "rect" type convenience character string
tinyplot(
  xmin = 100+i, ymin = 300+i, xmax = 150+i, ymax = 380+i,
  by = i, fill = 0.2,
  type = "rect"
)

# Same result with type_rect()
tinyplot(
  xmin = 100+i, ymin = 300+i, xmax = 150+i, ymax = 380+i,
  by = i, fill = 0.2,
  type = type_rect()
)
```

---

type_ridge *Ridge plot type*

Description

Type function for producing ridge plots (also known as joy plots), which display density distributions for multiple groups with vertical offsets. This function uses `tinyplot` scaffolding, which enables added functionality such as grouping and faceting.

The line color is controlled by the `col` argument in the `tinyplot()` call. The fill color is controlled by the bg argument in the `tinyplot()` call.

Usage

```
type_ridge(
  scale = 1.5,
  joint.max = c("all", "facet", "by"),
  breaks = NULL,
  probs = NULL,
  ylevels = NULL,
  bw = "nrd0",
  joint.bw = c("mean", "full", "none"),
  adjust = 1,
 kernel = c("gaussian", "epanechnikov", "rectangular", "triangular", "biweight",
    "cosine", "optcosine"),
  n = 512,
  gradient = FALSE,
  raster = FALSE,
  col = NULL,
  alpha = NULL
)
```

Arguments

scale	Numeric. Controls the scaling factor of each plot. Values greater than 1 means that plots overlap.
joint.max	character indicating how to scale the maximum of the densities: The default "all" indicates that all densities are scaled jointly relative to the same maximum so that the areas of all densities are comparable. Alternatively, "facet" indicates that the maximum is computed within each facet so that the areas of the densities are comparable within each facet but not necessarily across facets. Finally, "by" indicates that each row (in each facet) is scaled separately, so that the areas of the densities for by groups in the same row are comparable but not necessarily across rows.
breaks	Numeric. If a color gradient is used for shading, the breaks between the colors can be modified. The default is to use equidistant breaks spanning the range of the x variable.
probs	Numeric. Instead of specifying the same breaks on the x-axis for all groups, it is possible to specify group-specific quantiles at the specified probs. The quantiles are computed based on the density (rather than the raw original variable). Only one of breaks or probs must be specified.
ylevels	a character or numeric vector specifying in which order the levels of the y-variable should be plotted.
bw	the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below.) bw can also be a character string giving a rule to choose the bandwidth. See bw.nrd. The default, "nrd0", has remained the default for historical and compatibility reasons, rather than as a general recommendation, where e.g., "SJ" would rather fit, see also Venables and Ripley (2002). The specified (or computed) value of bw is multiplied by adjust.
joint.bw	character string indicating whether (and how) the smoothing bandwidth should be computed from the joint data distribution. The default of "mean" will compute the joint bandwidth as the mean of the individual subgroup bandwidths (weighted by their number of observations). Choosing "full" will result in a joint bandwidth computed from the full distribution (merging all subgroups). For "none" the individual bandwidth will be computed independently for each subgroup. Also accepts a logical argument, where TRUE maps to "mean" and FALSE maps to "none". See type_density for some discussion of practical considerations.
adjust	the bandwidth used is actually adjust*bw. This makes it easy to specify values like 'half the default' bandwidth.
kernel	a character string giving the smoothing kernel to be used. This must partially match one of "gaussian", "rectangular", "triangular", "epanechnikov", "biweight", "cosine" or "optcosine", with default "gaussian", and may be abbreviated to a unique prefix (single letter).

"cosine" is smoother than "optcosine", which is the usual 'cosine' kernel in the literature and almost MSE-efficient. However, "cosine" is the version used by S.

n — the number of equally spaced points at which the density is to be estimated. When n > 512, it is rounded up to a power of 2 during the calculations (as fft is used) and the final result is interpolated by approx. So it almost always makes sense to specify n as a power of two.

gradient — Logical or character. Should a gradient fill be used to shade the area under the density? If a character specification is used, then it can either be of length 1 and specify the palette to be used with gradient = TRUE corresponding to gradient = "viridis". If a character vector of length greater than 1 is used, then it should specify the colors in the palette, e.g., gradient = hcl.colors(512).

raster — Logical. Should the gradient fill be drawn using rasterImage? Defaults to FALSE, in which case the gradient fill will instead be drawn using polygon. See the Technical note on gradient fills section below.

col — Character string denoting the outline (border) color for all of the ridge densities. Note that a singular value is expected; if multiple colors are provided then only the first will be used. This argument is mostly useful for the aesthetic effect of drawing a common outline color in combination with gradient fills. See Examples.

alpha — Numeric in the range [0,1] for adjusting the alpha transparency of the density fills. In most cases, will default to a value of 1, i.e. fully opaque. But for some by grouped plots (excepting the special cases where by==y or by==x), will default to 0.6.

Technical note on gradient fills

tinyplot uses two basic approaches for drawing gradient fills in ridge line plots, e.g., if type_ridge(gradient = TRUE).

The first (and default) polygon-based approach involves dividing up the main density region into many smaller polygons along the x-axis. Each of these smaller polygons inherits a different color "segment" from the underlying palette swatch, which in turn creates the effect of a continuous gradient when they are all plotted together. Internally, this polygon-based approach is vectorized (i.e., all of the sub-polygons are plotted simultaneously). It is thus efficient from a plotting perspective and generally also performs well from an aesthetic perspective. However, it can occasionally produce undesirable plotting artifacts on some graphics devices—e.g., thin but visible vertical lines—if alpha transparency is being used at the same time.

For this reason, we also offer an alternative raster-based approach for gradient fills that users can invoke via type_ridge(gradient = TRUE, raster = TRUE). The essential idea is that we coerce the density polygon into a raster representation (using rasterImage) and achieve the gradient effect via color interpolation. The trade-off this time is potential smoothness artifacts around the top of the ridge densities at high resolutions, since we have converted a vector object into a raster object.

Again, we expect that the choice between these two approaches will only matter for ridge plots that combine gradient fills with alpha transparency (and on certain graphics devices). We recommend that users experiment to determine which approach is optimal for their device.

Examples

```
aq = transform(
  airquality,
  Month = factor(month.abb[Month], levels = month.abb[5:9]),
  Month2 = factor(month.name[Month], levels = month.name[5:9]),
  Late = ifelse(Day > 15, "Late", "Early")
  )

# default ridge plot (using the "ridge" convenience string)
tinyplot(Month ~ Temp, data = aq, type = "ridge")

# for ridge plots, we recommend pairing with the dedicated theme(s), which
# facilitate nicer y-axis labels, grid lines, etc.

tinytheme("ridge")
tinyplot(Month ~ Temp, data = aq, type = "ridge")

tinytheme("ridge2") # removes the plot frame (but keeps x-axis line)
tinyplot(Month ~ Temp, data = aq, type = "ridge")

# the "ridge(2)" themes are especially helpful for long y labels, due to
# dyanmic plot adjustment
tinyplot(Month2 ~ Temp, data = aq, type = "ridge")

# pass customization arguments through type_ridge()... for example, use
# the scale argument to change/avoid overlap of densities (more on scaling
# further below)

tinyplot(Month ~ Temp, data = aq, type = type_ridge(scale = 1))

## by grouping is also supported. two special cases of interest:

# 1) by == y (color by y groups)
tinyplot(Month ~ Temp | Month, data = aq, type = "ridge")

# 2) by == x (gradient coloring along x)
tinyplot(Month ~ Temp | Temp, data = aq, type = "ridge")

# aside: pass explicit `type_ridge(col = <col>)` arg to set a different
# border color
tinyplot(Month ~ Temp | Temp, data = aq, type = type_ridge(col = "white"))

# gradient coloring along the x-axis can also be invoked manually without
# a legend (the next two tinyplot calls are equivalent)

# tinyplot(Month ~ Temp, data = aq, type = type_ridge(gradient = "agsunset"))
tinyplot(Month ~ Temp, data = aq, type = type_ridge(gradient = TRUE))

# aside: when combining gradient fill with alpha transparency, it may be
# better to use the raster-based approach (test on your graphics device)

tinyplot(Month ~ Temp, data = aq,
```

```
  type = type_ridge(gradient = TRUE, alpha = 0.5),
  main = "polygon fill (default)")
tinyplot(Month ~ Temp, data = aq,
  type = type_ridge(gradient = TRUE, alpha = 0.5, raster = TRUE),
  main = "raster fill")

# highlighting only the center 50% of the density (i.e., 25%-75% quantiles)
tinyplot(Month ~ Temp, data = aq, type = type_ridge(
  gradient = hcl.colors(3, "Dark Mint")[c(2, 1, 2)],
  probs = c(0.25, 0.75), col = "white"))

# highlighting the probability distribution by color gradient
# (darkest point = median)
tinyplot(Month ~ Temp, data = aq, type = type_ridge(
  gradient = hcl.colors(250, "Dark Mint")[c(250:1, 1:250)],
  probs = 0:500/500))

# faceting also works, although we recommend switching (back) to the "ridge"
# theme for faceted ridge plots

tinytheme("ridge")
tinyplot(Month ~ Ozone, facet = ~ Late, data = aq,
  type = type_ridge(gradient = TRUE))

## use the joint.max argument to vary the maximum density used for
## determining relative scaling...

# jointly across all densities (default) vs. per facet
tinyplot(Month ~ Temp, facet = ~ Late, data = aq,
  type = type_ridge(scale = 1))
tinyplot(Month ~ Temp, facet = ~ Late, data = aq,
  type = type_ridge(scale = 1, joint.max = "facet"))

# jointly across all densities (default) vs. per by row
tinyplot(Month ~ Temp | Late, data = aq,
  type = type_ridge(scale = 1))
tinyplot(Month ~ Temp | Late, data = aq,
  type = type_ridge(scale = 1, joint.max = "by"))

# restore the default theme
tinytheme()
```

---

type_rug *Add a rug to a plot*

Description

Adds a rug representation (1-d plot) of the data to the plot.

Usage

```
type_rug(
  ticksize = 0.03,
  side = 1,
  quiet = getOption("warn") < 0,
  jitter = FALSE,
  amount = NULL
)
```

Arguments

ticksize	The length of the ticks making up the 'rug'. Positive lengths give inwards ticks.
side	On which side of the plot box the rug will be plotted. Normally 1 (bottom) or 3 (top).
quiet	logical indicating if there should be a warning about clipped values.
jitter	Logical. Add jittering to separate ties? Default is FALSE.
amount	Numeric. Amount of jittering (see jitter). Only used if jitter is TRUE.

Details

This function should only be used as part of tinyplot_add(), i.e. adding to an existing plot.

In most cases, determining which variable receives the rug representation will be based on the side argument (i.e., x-variable if side is 1 or 3, and y-variable if side is 2 or 4). An exception is if the preceding plot type was either "density" or "histogram"; for these latter cases, the x-variable will always be used. See Examples.

Examples

```
tinyplot(~wt | am, data = mtcars, type = "density", facet = "by", fill = "by")
tinyplot_add(type = "rug")
# use type_rug() to pass extra options
tinyplot_add(type = type_rug(side = 3, ticksize = 0.05))

# For ties, use jittering
tinyplot(eruptions ~ waiting, data = faithful, type = "lm")
tinyplot_add(type = type_rug(jitter = TRUE, amount = 0.3))
tinyplot_add(type = type_rug(jitter = TRUE, amount = 0.1, side = 2))
# Add original points just for reference
tinyplot_add(type = "p")
```

type_segments *Line segments plot type*

Description

Type function for plotting line segments.

Usage

```
type_segments()
```

Details

Contrary to base `segments`, line segments in tinyplot must be specified using the `xmin`, `ymin`,`xmax`, and `ymax` arguments.

Examples

```
# "segments" type convenience character string
tinyplot(
  xmin = c(0,.1), ymin = c(.2,1), xmax = c(1,.9), ymax = c(.75,0),
  type = "segments"
)

# Same result with type_segments()
tinyplot(
  xmin = c(0,.1), ymin = c(.2,1), xmax = c(1,.9), ymax = c(.75,0),
  type = type_segments()
)
```

type_spineplot *Spineplot and spinogram types*

Description

Type function(s) for producing spineplots and spinograms, which are modified versions of histograms or mosaic plots, and particularly useful for visualizing factor variables. Note that `tinyplot` defaults to `type_spineplot()` if `y` is a factor variable.

Usage

```
type_spineplot(
  breaks = NULL,
  tol.ylab = 0.05,
  off = NULL,
  ylevels = NULL,
  col = NULL,
  xaxlabels = NULL,
  yaxlabels = NULL,
  weights = NULL
)
```

Arguments

`breaks`	if the explanatory variable is numeric, this controls how it is discretized. `breaks` is passed to `hist` and can be a list of arguments.
`tol.ylab`	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
`off`	vertical offset between the bars (in per cent). It is fixed to `0` for spinograms and defaults to `2` for spine plots.
`ylevels`	a character or numeric vector specifying in which order the levels of the dependent variable should be plotted.
`col`	a vector of fill colors of the same length as `levels(y)`. The default is to call `gray.colors`.
`xaxlabels, yaxlabels`	character vectors for annotation of x and y axis. Default to `levels(y)` and `levels(x)`, respectively for the spine plot. For `xaxlabels` in the spinogram, the breaks are used.
`weights`	numeric. A vector of frequency weights for each observation in the data. If `NULL` all weights are implicitly assumed to be 1. If `x` is already a 2-way table, the weights are ignored.

Examples

```
# "spineplot" type convenience string
tinyplot(Species ~ Sepal.Width, data = iris, type = "spineplot")

# Aside: specifying the type is redundant for this example, since tinyplot()
# defaults to "spineplot" if y is a factor (just like base plot).
tinyplot(Species ~ Sepal.Width, data = iris)

# Use `type_spineplot()` to pass extra arguments for customization
tinyplot(Species ~ Sepal.Width, data = iris, type = type_spineplot(breaks = 4))

p = palette.colors(3, "Pastel 1")
tinyplot(Species ~ Sepal.Width, data = iris, type = type_spineplot(breaks = 4, col = p))
rm(p)
```

```
# More idiomatic tinyplot way of drawing the previous plot: use y == by
tinyplot(
  Species ~ Sepal.Width | Species, data = iris, type = type_spineplot(breaks = 4),
  palette = "Pastel 1", legend = FALSE
)

# Grouped and faceted spineplots

ttnc = as.data.frame(Titanic)

tinyplot(
  Survived ~ Sex, facet = ~ Class, data = ttnc,
  type = type_spineplot(weights = ttnc$Freq)
)

# For grouped "by" spineplots, it's better visually to facet as well
tinyplot(
  Survived ~ Sex | Class, facet = "by", data = ttnc,
  type = type_spineplot(weights = ttnc$Freq)
)

# Fancier version. Note the smart inheritance of spacing etc.
tinyplot(
  Survived ~ Sex | Class, facet = "by", data = ttnc,
  type = type_spineplot(weights = ttnc$Freq),
  palette = "Dark 2", facet.args = list(nrow = 1), axes = "t"
)

# Note: It's possible to use "by" on its own (without faceting), but the
# overlaid result isn't great. We will likely overhaul this behaviour in a
# future version of tinyplot...
tinyplot(Survived ~ Sex | Class, data = ttnc,
  type = type_spineplot(weights = ttnc$Freq), alpha = 0.3
)
```

`type_spline` *Spline plot type*

Description

Type function for plotting a cubic (or Hermite) spline interpolation. Arguments are passed to `spline`; see this latter function for default argument values.

Usage

```
type_spline(
  n = NULL,
  method = "fmm",
  xmin = NULL,
```

```
  xmax = NULL,
  xout = NULL,
  ties = mean
)
```

Arguments

n	if xout is left unspecified, interpolation takes place at n equally spaced points spanning the interval [xmin, xmax].
method	specifies the type of spline to be used. Possible values are "fmm", "natural", "periodic", "monoH.FC" and "hyman". Can be abbreviated.
xmin, xmax	left-hand and right-hand endpoint of the interpolation interval (when xout is unspecified).
xout	an optional set of values specifying where interpolation is to take place.
ties	handling of tied x values. The string "ordered" or a function (or the name of a function) taking a single vector argument and returning a single number or a length-2 list of both, see approx and its 'Details' section, and the example below.

Details

The inputs can contain missing values which are deleted, so at least one complete (x, y) pair is required. If method = "fmm", the spline used is that of Forsythe, Malcolm and Moler (an exact cubic is fitted through the four points at each end of the data, and this is used to determine the end conditions). Natural splines are used when method = "natural", and periodic splines when method = "periodic".

The method "monoH.FC" computes a *monotone* Hermite spline according to the method of Fritsch and Carlson. It does so by determining slopes such that the Hermite spline, determined by (x_i, y_i, m_i), is monotone (increasing or decreasing) **iff** the data are.

Method "hyman" computes a *monotone* cubic spline using Hyman filtering of an method = "fmm" fit for strictly monotonic inputs.

These interpolation splines can also be used for extrapolation, that is prediction at points outside the range of x. Extrapolation makes little sense for method = "fmm"; for natural splines it is linear using the slope of the interpolating curve at the nearest data point.

Examples

```
# "spline" type convenience string
tinyplot(dist ~ speed, data = cars, type = "spline")

# Use `type_spline()` to pass extra arguments for customization
tinyplot(dist ~ speed, data = cars, type = type_spline(method = "natural", n = 25),
    add = TRUE, lty = 2)
```

type_summary *Plot summary values of* y *at unique values of* x

Description

Applies a summary function to y along unique values of x. This is useful, say, for quickly plotting mean values of your dataset. Internally, type_summary() applies a thin wrapper around ave and then passes the result to type_lines for drawing.

Usage

```
type_summary(fun = mean, ...)
```

Arguments

fun	summarizing function. Should be compatible with ave. Defaults to mean.
...	Additional arguments are passed to the lines() function, ex: type="p", col="pink".

See Also

ave which performs the summarizing (averaging) behind the scenes.

Examples

```
# Plot the mean chick weight over time
tinyplot(weight ~ Time, data = ChickWeight, type = type_summary())

# mean is the default function, so the above is equivalent to
tinyplot(weight ~ Time, data = ChickWeight, type = type_summary(mean))

# Plot the median instead
tinyplot(weight ~ Time, data = ChickWeight, type = type_summary(median))

# Works with groups and/or facets too
tinyplot(weight ~ Time | Diet, facet = "by", data = ChickWeight, type = type_summary())

# Custom/complex function example
tinyplot(
  weight ~ Time | Diet, facet = "by", data = ChickWeight,
  type = type_summary(function(y) quantile(y, probs = 0.9)/max(y))
)
```

type_text *Text annotations plot type*

Description

Type function for adding text annotations to a plot. This function allows you to draw text at specified (x,y) coordinates.

Usage

```
type_text(
  labels,
  adj = NULL,
  pos = NULL,
  offset = 0.5,
  vfont = NULL,
  font = NULL
)
```

Arguments

labels	Character vector of length 1 or of the same length as the number of x,y coordinates.
adj	one or two values in $[0, 1]$ which specify the x (and optionally y) adjustment ('justification') of the labels, with 0 for left/bottom, 1 for right/top, and 0.5 for centered. On most devices values outside $[0, 1]$ will also work. See below.
pos	a position specifier for the text. If specified this overrides any adj value given. Values of 1, 2, 3 and 4, respectively indicate positions below, to the left of, above and to the right of the specified (x,y) coordinates.
offset	when pos is specified, this value controls the distance ('offset') of the text label from the specified coordinate in fractions of a character width.
vfont	NULL for the current font family, or a character vector of length 2 for Hershey vector fonts. The first element of the vector selects a typeface and the second element selects a style. Ignored if labels is an expression.
font	Font to be used, following graphics::par()

Examples

```
tinyplot(mpg ~ hp | factor(cyl),
  data = mtcars,
  type = type_text(
    labels = row.names(mtcars),
    font = 2,
    adj = 0))
```

type_vline *Trace a vertical line on the plot*

Description

Trace a vertical line on the plot

Usage

```
type_vline(v = 0)
```

Arguments

v	x-value(s) for vertical line(s). Numeric of length 1 or equal to the number of facets.

Examples

```
tinyplot(mpg ~ hp, data = mtcars)
tinyplot_add(type = type_vline(150))

# facet-specify location and colors
cols = c("black", "green", "orange")
tinyplot(mpg ~ hp | factor(cyl), facet = ~ factor(cyl), data = mtcars, col = cols)
tinyplot_add(type = type_vline(v = c(100, 150, 200)), lty = 3, lwd = 3)
```

Index

abline, *5*, *14*
approx, *30*, *45*, *52*
ave, *53*

boxplot, *28*
boxplot.stats, *28*
bw.nrd, *29*, *44*

dev.off, *5*
dnorm, *33*
draw_legend, 2, *5*

family, *34*
fft, *30*, *45*
formula, *15*

get_saved_par, *5*, 5, *14*, *25*
glm, *34*
graphics::par, *25*
graphics::par(), *54*
gray.colors, *50*

Hershey, *54*
hist, *35*, *50*

interaction, *9*

jitter, *37*, *48*
jpeg, *14*

list, *52*
lm, *38*
loess, *39*
loess.control, *39*

mean, *53*

options, *23*

palette, *12*
par, *5, 6*, *9*, *14, 15*, *23, 24*
pdf, *14*

plot, *7*, *10*, *15*, *38*
plot.density, *31*
plt *(*tinyplot*)*, 7
plt_add *(*tinyplot_add*)*, 19
png, *14*
polygon, *40*, *45*
polypath, *41*

rasterImage, *45*
rect, *42*

segments, *49*
spline, *51*
svg, *14*

text, *5*, *14*
tinyplot, *3*, *5*, 7, *42*, *49*
tinyplot.default, *20*
tinyplot.formula, *20*
tinyplot_add, *14*, *19*, 19
tinyplot_add(), *48*
tinytheme, 20, *25*
tpar, *10*, *14*, *22*, 23
type_abline, 26
type_abline(), *11*
type_area, 27
type_area(), *10*
type_boxplot, 28
type_boxplot(), *11*
type_density, 29, *44*
type_density(), *11*
type_errorbar, 32
type_errorbar(), *10*
type_function, 33
type_function(), *11*
type_glm, 34
type_glm(), *11*
type_hist, *9*
type_hist *(*type_histogram*)*, 34
type_histogram, 34

type_histogram(), *11*
type_hline, 36
type_hline(), *11*
type_jitter, 37
type_jitter(), *11*
type_lines, 38, *53*
type_lm, 38
type_lm(), *11*
type_loess, 39
type_loess(), *11*
type_pointrange *(*type_errorbar*)*, 32
type_pointrange(), *10*
type_points, 40
type_polygon, 40
type_polygon(), *11*
type_polypath, 41
type_polypath(), *11*
type_qq, 42
type_qq(), *11*
type_rect, 42
type_rect(), *11*
type_ribbon *(*type_area*)*, 27
type_ribbon(), *11*
type_ridge, 43
type_ridge(), *11*, *21*
type_rug, 47
type_rug(), *11*
type_segments, 49
type_segments(), *11*
type_spineplot, 49
type_spineplot(), *11*
type_spline, 51
type_spline(), *11*
type_summary, 53
type_summary(), *11*
type_text, 54
type_text(), *11*
type_vline, 55
type_vline(), *11*

xy.coords, *9*