# Package 'tinyplot'

July 3, 2026

**Type** Package

**Title** Lightweight Extension of the Base R Graphics System

**Version** 0.7.0

**Date** 2026-07-02

**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 (>= 4.0.0)

**Imports** graphics, grDevices, stats, tools, utils

**Suggests** fontquiver, png, rsvg, svglite (>= 2.2.0), tinytest, tinysnapshot (>= 0.0.3), knitr

**Config/Needs/website** altdoc (>= 0.7.2), future.apply

**Encoding** UTF-8

**URL** https://grantmcdermott.com/tinyplot/

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

**Config/roxygen2/version** 8.0.0

**NeedsCompilation** no

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

**Maintainer** Grant McDermott <contact@grantmcdermott.com>

**Repository** CRAN

**Date/Publication** 2026-07-03 06:40:02 UTC

# Contents

draw_legend . . . 3
get_saved_par . . . 5
tinylabel . . . 7
tinyplot . . . 9
tinyplot.data.frame . . . 25
tinyplot.matrix . . . 26
tinyplot.ts . . . 28
tinyplot_add . . . 29
tinytheme . . . 30
tinytheme_register . . . 35
tpar . . . 36
type_abline . . . 40
type_area . . . 42
type_barplot . . . 44
type_boxplot . . . 47
type_chull . . . 49
type_density . . . 50
type_ellipse . . . 52
type_errorbar . . . 53
type_function . . . 55
type_glm . . . 57
type_histogram . . . 58
type_jitter . . . 60
type_lines . . . 61
type_lm . . . 62
type_loess . . . 63
type_points . . . 64
type_polygon . . . 65
type_polypath . . . 66
type_qq . . . 66
type_rect . . . 67
type_ridge . . . 68
type_rug . . . 72
type_segments . . . 73
type_spineplot . . . 74
type_spline . . . 76
type_summary . . . 77
type_text . . . 78
type_violin . . . 81

**Index** **84**

`draw_legend` *Calculate placement and draw legend*

## Description

Main exported function for drawing legends. Supports:

- Inner and outer positioning (with "!" suffix)
- Discrete and continuous (gradient) legends
- Automatic margin adjustment

## Usage

```
draw_legend(
  legend = NULL,
  legend_args = NULL,
  by_dep = NULL,
  lgnd_labs = NULL,
  labeller = NULL,
  type = NULL,
  pch = NULL,
  lty = NULL,
  lwd = NULL,
  col = NULL,
  bg = NULL,
  cex = NULL,
  gradient = FALSE,
  lmar = NULL,
  has_sub = FALSE,
  has_cap = FALSE,
  cap_text = NULL,
  new_plot = TRUE,
  draw = TRUE,
  soma_target = NULL,
  dynmar_title_mar = NULL
)
```

## 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 = ...)`. |
| `labeller` | Character or function for formatting the labels (`lgnd_labs`). Passed down to `tinylabel`. |

| | |
|---|---|
| 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. |
| has_cap | Logical. Does the plot have a caption. Only used if keyword position is "bottom!", in which case we need to bump the legend margin a bit further. |
| cap_text | Character. The caption text to draw below the legend when position is "bottom!". Ignored otherwise. |
| new_plot | Logical. Should we be calling plot.new internally? |
| draw | Logical. If `FALSE`, no legend is drawn but the sizes are returned. Note that a new (blank) plot frame will still need to be started in order to perform the calculations. |
| soma_target | Numeric. Shared outer margin target (in lines) for multi-legend alignment. If `NULL`, each legend computes its own margin. |
| dynmar_title_mar | Numeric or `NULL`. The pre-computed `dynmar_computed[3]` value for "top!" legends under dynmar themes. When set, the legend margin formula uses this directly to ensure correct title positioning. |

## 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)
```

---

## tinylabel *Format labels*

---

## Description

Function for formatting label appearance, e.g. axis ticks labels. This is what the top-level `xaxl` and `yaxl` arguments from `tinyplot` ultimately get passed to.

## Usage

```
tinylabel(x, labeller = NULL, na.ignore = TRUE, na.rm = TRUE)
```

## Arguments

| | |
|---|---|
| `x` | a numeric or character vector |
| `labeller` | a formatting function to be applied to `x`, e.g. `format`, `toupper`, `abs`, or other custom function (including from the popular **scales** package). Can also be one of the following convenience strings (symbols), for which common formatting transformations are provided: `"percent"` (`"%"`), `"comma"` (`","`), `"log"` (`"l"`), `"dollar"` (`"$"`), `"euro"` (`"€"`), or `"sterling"` (`"£"`). |
| `na.ignore` | logical indicating whether the labelling function should ignore `NA` values in `x`. In other words, should the `NA` values be left as-is? Default is `TRUE`. |
| `na.rm` | logical indicating whether `NA` values should be removed from `x` and thus the return object too. Default is `TRUE`, but only evaluated if `na.ignore` is `FALSE`. |

## Value

a character vector of the same length as `x` with the transformed labels.

## See Also

`tinyplot`

## Examples

```
x = 1e4
tinylabel(x, "comma")
tinylabel(x, ",") # same
tinylabel(x, "$") # or "dollar"

# invoke tinylabel from a parent tinyplot call...
#   => x/yaxl for adjusting axes tick labels
#   => legend = list(labeller = ...) for adjusting the legend labels
s77 = transform(data.frame(state.x77), Illiteracy = Illiteracy / 100)
tinyplot(Life.Exp ~ Income | Illiteracy, data = s77,
         xaxl = '$',
         legend = list(labeller = '%'))

# log example (combined with axis scaling)
tinyplot(x = 10^c(10:0), y = 0:10, type = "b",
         log = "x", xaxl = "log")

# combine with `x/yaxb` to adjust the actual tick marks ("break points")
# at the same time
```

```
tinyplot(x = 10^c(10:0), y = 0:10, type = "b",
         log = "x", xaxl = "log", xaxb = 10^c(1,3,5,7,9))

#
## custom function examples

## example I: date formatting

dat = data.frame(
  date = seq(as.Date("2000/1/1"), by = "month", length.out = 12),
  trend = 1:12 + rnorm(12, sd = 1)
)

tinyplot(trend ~ date, data = dat,
         xaxl = function(x) format(x, "%b, %Y"))

## example II: string wrapping

# create a "vectorised" version of `base::strwrap` that breaks long
# strings into new lines every 18 characters
strwrap18 = function(x) sapply(
  strwrap(x, width = 18, simplify = FALSE),
  paste,
  collapse = "\n"
)

# now demonstrate on a dataset with long y-tick labels
dat2 = data.frame(
  x = rep(rnorm(100), 3),
  y = c(
    "tinyplot is a lightweight extension of the base R graphics system.",
    "R is a language for statistical computing.",
    "Data visualization is an essential skill."
  )
)

tinyplot(y ~ x, data = dat2, type = "j",
         yaxl = strwrap18,
         theme = "bw") # use theme for horizontal labels + dynamic margin
```

---

`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,
  xmin = NULL,
  xmax = NULL,
  ymin = NULL,
  ymax = NULL,
  weights = NULL,
  labels = NULL,
  by = NULL,
  facet = NULL,
  facet.args = NULL,
  data = NULL,
  type = NULL,
  legend = NULL,
  main = NULL,
  sub = NULL,
  cap = NULL,
  xlab = NULL,
  ylab = NULL,
  ann = par("ann"),
  xlim = NULL,
  ylim = NULL,
  axes = TRUE,
  xaxt = NULL,
  yaxt = NULL,
  xaxs = NULL,
  yaxs = NULL,
  xaxb = NULL,
  yaxb = NULL,
  xaxl = NULL,
  yaxl = NULL,
  log = "",
  flip = FALSE,
  frame.plot = NULL,
  grid = NULL,
  palette = NULL,
  pch = NULL,
  lty = NULL,
  lwd = NULL,
  col = NULL,
  bg = NULL,
  fill = NULL,
  alpha = NULL,
```

```
  cex = NULL,
  add = FALSE,
  draw = NULL,
  empty = FALSE,
  restore.par = FALSE,
  file = NULL,
  width = NULL,
  height = NULL,
  asp = NA,
  theme = NULL,
  ...
)

## S3 method for class 'formula'
tinyplot(
  x = NULL,
  data = parent.frame(),
  facet = NULL,
  facet.args = NULL,
  type = NULL,
  xmin = NULL,
  xmax = NULL,
  ymin = NULL,
  ymax = NULL,
  weights = NULL,
  labels = NULL,
  xlim = NULL,
  ylim = NULL,
  main = NULL,
  sub = NULL,
  cap = 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). |
| 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). Can be passed as unquoted (NSE) variable names when called from the tinyplot.formula method, e.g. ymin = varname where varname is a variable in data. |
| weights | an optional numeric vector of observation weights, of the same length as the number of x,y coordinates. Can be passed as an unquoted (NSE) variable name when called from the tinyplot.formula method, e.g. weights = varname where varname is a variable in data. Weights are only consumed by the plot types that support them; specifically, the model fit and distribution types like type_lm() and type_spineplot(). Because weights trigger a statistical transformation, a warning is emitted if this argument is supplied to a non-supported type (even though the actual weights will be ignored, regardless). |
| labels | an optional vector of text labels for use with type_text(), of length 1 or the same length as the number of x,y coordinates. Can be passed as an unquoted (NSE) variable name when called from the tinyplot.formula method, e.g. labels = varname where varname is a variable in data. Ignored by other plot types. Overrides the labels argument of type_text() if both are supplied. |
| 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, lty, or cex) 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:
    * `"barplot"` / `type_barplot()`: Creates a bar plot.
    * `"boxplot"` / `type_boxplot()`: Creates a box-and-whisker plot.
    * `"chull"` / `type_chull()`: Draws convex hull(s) around grouped points.
    * `"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.
    * `"violin"` / `type_violin()`: Creates a violin plot.
  - 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.

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. Supported keywords:
  - Standard position keywords from base legend(), e.g. "right", "topleft", "bottom", etc.
  - Outer positions via a trailing "!", e.g. "right!", "topleft!", or "bottom!". This places the legend *outside* the plotting area and adjusts the margins accordingly.
  - "direct": places text labels at the last point of each group's data, coloured to match. Best suited to line-based plots with x-sorted data, where "last" corresponds to "rightmost". The right margin is automatically expanded to prevent clipping. Requires discrete groups via by. For faceted plots, labels are drawn in each panel for the groups present there. Supports additional arguments when passed via legend(...):
    - nudge_x, nudge_y: numeric vectors of per-group offsets in data units. Recycled to the number of groups. Supports named vectors for targeted adjustment, e.g. nudge_y = c("Group A" = -10, "Group B" = 20).
    - repel: if TRUE, automatically separates overlapping labels vertically. If a positive number, sets the minimum gap between labels in data units.
  - "none": turns off legend printing.
- 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. To suppress the legend title, set title = NULL or title = FALSE, e.g. legend = list(title = FALSE).

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

sub — a subtitle for the plot.

cap — a caption for the plot, drawn at the bottom. Useful for annotations like data sources. Best paired with a dynamic tinytheme. For the default theme, should be seen as a substitute for sub, since these two will otherwise overlap. Appearance can be customized via tpar parameters adj.cap, cex.cap, col.cap, font.cap, and line.cap.

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.

xlim — the x limits (x1, x2) of the plot. Note that x1 > x2 is allowed and leads to a reversed axis (although see the "rev(erse)" keyword option below). The default value, NULL, indicates that the range of the finite values to be plotted should be used. Alongside the standard length-2 vector (e.g., xlim = c(0, 1)), tinyplot supports three further convenience forms:

- a single scalar (e.g. xlim = 0) ensures that the provided value is covered by the axis range, irrespective of the data extent.
- a length-2 vector with one NA (e.g. xlim = c(0, NA)) pins the non-NA limit and lets the data determine the other limit.
- the convenience string "rev" (or the longer "reverse") reverses the automatically-computed axis range. This is equivalent to passing a descending vector, but without having to know the data extent in advance.

ylim — the y limits of the plot. Accepts the same input forms as xlim.

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.

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

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.

xaxb, yaxb — numeric vector (or character vector, if appropriate) giving the break points at which the axis tick-marks are to be drawn. Break points outside the range of the data will be ignored if the associated axis variable is categorical, or an explicit x/ylim range is given.

xaxl, yaxl — a function or a character keyword specifying the format of the x- or y-axis tick labels. Note that this is a post-processing step that affects the *appearance* of the tick labels only; use in conjunction with x/yaxb if you would like to adjust the position of the tick marks too. In addition to user-supplied formatting functions (e.g., format, toupper, abs, or other custom function), several convenience keywords (or their symbol equivalents) are available for common formatting transformations: "percent" ("%"), "comma" (","), "log" ("l"), "dollar" ("$"), "euro" ("€"), or "sterling" ("£"). See the tinylabel documentation for examples.

`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.

`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.

`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.

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

- a logical (i.e., `TRUE` to draw the grid on both axes).
- a character string controlling which axes get grid lines and at what resolution. Uppercase letters (`"X"`, `"Y"`, `"XY"`) draw grid lines at the standard axis tick positions (with the latter equivalent to `TRUE`), while lowercase letters (`"x"`, `"y"`, `"xy"`) draw a finer grid with additional lines at the midpoints between ticks. (Note: the finer grid has no effect on log-scale axes, since tick positions already include intermediate values.)
- 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.
- A vector or list of colours, e.g. `c("darkorange", "purple", "cyan4")`. If too few colours are provided for a discrete (qualitative) set of groups, then the colours will be recycled with a warning. For continuous (sequential) groups, a gradient palette will be interpolated.

`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., `par("lwd")`) and pad on either side of that.

`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. There are two additional considerations, specifically for points-alike plot types (e.g. `"p"`):

- users can also supply a special `cex = "by"` convenience argument, in which case the character expansion will automatically adjust by group too. The range of this character expansion is controlled by the `clim` argument in the respective types; see `type_points()` for example.

- passing a cex vector of equal length to the main x and y variables (e.g., another column in the same dataset) will yield a "bubble"plot with its own dedicated legend. This can provide a useful way to visualize an extra dimension of the data; see Examples.

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 within tinyplot() and thus has access to the local definition of all variables such as x, y, etc. See Examples.

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.

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.

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.

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

`theme` keyword string (e.g. `"clean"`) or list defining a theme. Passed on to `tinytheme`, but reset upon exit so that the theme effect is only temporary. Useful for invoking ephemeral themes.

`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`. Only a single `y` and `x` variable (if any) must be specified but multiple grouping variables can be included 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.) If arithmetic operators are used for transforming variables, they should be wrapped in `I()`, e.g., `I(y1/y2) ~ x`. 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
)

# Use the special "by" convenience keyword if you would like to map these
# aesthetic features over groups too (i.e., in addition to the default
# colour grouping)

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  pch = "by",
  cex = "by"
)

# 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`/`fill` 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
```

```
)

# Aside: For "bubble" plots, pass an appropriate vector to the `cex` arg.
# This can be useful for depicting an additional dimension of the data (here:
# Wind).
tinyplot(
  Temp ~ Day | Month,
  data = aq,
  pch = 21,
  cex = aq$Wind, # map character size to another feature in the data
  bg = 0.3,
  col = "black"
)

# 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")
)

# Use legend = "direct" to place text labels at the last point of each
# group's data, coloured to match. Best suited to line-based plots with
# x-sorted data, where "last" corresponds to "rightmost". The right
# margin is automatically expanded to fit the labels. Pairs well with
```

```
# dynamic themes for tighter margins overall.

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "l",
  legend = "direct",
  theme = "clean2"
)

# Use `nudge_x/y`` to manually adjust label positions, or `repel` to
# auto-separate overlapping labels:
tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "l",
  legend = legend("direct", nudge_y = c(May = -1, Jun = 2)),
  # legend = legend("direct", repel = TRUE), # another option
  theme = "clean2"
)

# 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`).

tinyplot(
  Temp ~ Day | Month,
  data = aq,
  type = "b",
  alpha = 0.5,
  main = "Daily temperatures by month",
  sub = "Brought to you by tinyplot",
  cap = "Source: Base R airquality dataset",
  theme = "clean2"
)

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

`tinyplot.data.frame` *tinyplot Method for Plotting Data Frames*

## Description

Convenience interface for visualizing `data.frame` objects with tinyplot.

## Usage

```
## S3 method for class 'data.frame'
tinyplot(x, formula = NULL, by = NULL, labs = FALSE, frames = FALSE, ...)

tinypairs(x, by = NULL, labs = FALSE, frames = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `x` | an object of class `"data.frame"`. |
| `formula` | a `formula` that is passed on to `tinyplot.formula`. If `formula` is `NULL` a formula of type `y ~ 1` or `y ~ x` is set up for 1- and 2-variable data frames, respectively. For data frames with 3 or more variables, a `pairs`-style grid of all variable combinations is drawn instead. |
| `by` | (pairs-case only) optional grouping variable. Either 1) a character string giving a column name in `x`, or 2) a vector of equal length to the main data.frame (*caveat emptor*: this is not checked). The variable is spliced into each sub-plot's formula as `y ~ x \| by` so groups are distinguished (e.g. by colour). Note that the legend is deliberately suppressed. |
| `labs` | (pairs-case only) logical indicating whether the axes labels (titles) for each sub-plot in the pairs-style case should be shown. Default is `FALSE`. |
| `frames` | (pairs-case only) logical indicating whether each sub-plot should be framed by a box. Default is `FALSE`. Note the trailing "s" plural case to disambiguate from `frame.plot`. |
| `...` | further arguments passed to `tinyplot`. |

## Details

This is a convenience function for plotting data frames with or without a formula. The case with the formula mainly facilitates using `tinyplot()` in combination with pipes. The case without formula provides a quick way of plotting the variables in a data frame: a single variable, a pair of variables, or a `pairs`-style grid of all variable combinations for 3 or more variables. The latter can be called directly (also for less than 3 variables) via `tinypairs()`.

Note that, unlike a base `pairs` plot, these `tinyplot` variants (deliberately) produce a histogram for the univariate cases along the grid diagonal.

## Value

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

## Examples

```
## using tinyplot() with data frames
tinyplot(cars)
tinyplot(iris, Sepal.Length ~ Petal.Width | Species)
tinyplot(
  as.data.frame(UCBAdmissions),
  Admit ~ Gender, facet = 1 ~ Dept, weights = Freq
)

## note that this also enables usage with pipes (in R >= 4.1.0) such as
# cars |> tinyplot()
# iris |> tinyplot(Sepal.Length ~ Petal.Width | Species)
# UCBAdmissions |>
#   as.data.frame() |>
#   tinyplot(Admit ~ Gender, facet = 1 ~ Dept, weights = Freq)
# etc.

## pairs-style display for data frames with 3 or more variables
## (works best with a theme)
tinytheme("clean2")
tinyplot(iris)

## pass `by` arg to group the pairs display (legend is suppressed)
## here, we also add optional frames around the individual sub-plots
tinyplot(iris, by = "Species", frames = TRUE)

## another option (but assumes objects of equal length)
tinyplot(iris[, 1:4], by = iris$Species, frames = TRUE)

## tinypairs can also create 2x2 pairs display
tinypairs(iris[, c(3, 1)], by = iris$Species)

tinytheme() ## reset theme
```

---

# tinyplot.matrix *tinyplot Method for Plotting Matrices*

---

## Description

Convenience interface for visualizing `matrix` objects with tinyplot.

## Usage

```
## S3 method for class 'matrix'
tinyplot(
  x,
  type = NULL,
  legend = NULL,
```

```
  facet = NULL,
  xlab = NULL,
  ylab = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| x | an object of class "matrix". |
| type | plot type passed on to tinyplot. Defaults to "p" (points). |
| legend | specification passed on to tinyplot. The default is to draw a legend when the matrix has named columns, and to suppress it otherwise. |
| facet | specification of facet passed on to tinyplot. The only accepted non-NULL value is the "by" convenience string, which facets the plot by matrix column. |
| xlab, ylab | axis labels passed on to tinyplot. ylab defaults to the deparsed matrix name. xlab defaults to "Index" when the matrix has no row names; when it does, the row names already label the ticks so the x-axis title is suppressed. |
| ... | further arguments passed to tinyplot. |

## Details

Internally the matrix is converted to long form and visualized as a scatter (or other type) of each column's values against their row index. Each column is mapped to a separate by category, so a matrix with multiple columns produces a grouped plot. Optionally, it can also be faceted via facet = "by". This mirrors the base R matplot convention of plotting the columns of a matrix against the row numbers. If the matrix has column names, these are used as the group (and legend) labels. Single-column matrices are drawn as a simple index plot with no grouping or legend.

## Value

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

## See Also

matplot

## Examples

```
# basic use
tinyplot(VADeaths)
tinyplot(VADeaths, type = "b")
tinyplot(VADeaths, type = "b", legend = "direct", theme = "socviz")
tinyplot(VADeaths, type = "b", legend = FALSE, facet = "by", theme = "socviz")

# equivalent plot to an example in `?matplot`
sines = outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
tinyplot(sines, type = "o", pch = "by", lty = "by", col = rainbow(ncol(sines)))
```

`tinyplot.ts` *tinyplot Method for Plotting ts Objects (Time Series)*

## Description

Convenience interface for visualizing `ts` (time series) objects with tinyplot.

## Usage

```
## S3 method for class 'ts'
tinyplot(
  x,
  facet,
  type = "l",
  facet.args = list(free = TRUE),
  xlab = NULL,
  ylab = NA,
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | an object of class `"ts"`. |
| `facet` | specification of `facet` for `tinyplot.formula`. The default in the `tinyplot` method is to use `facet = NULL` for univariate series and `facet = ~ Series` (equivalent to `facet = "by"`) for multivariate series. |
| `type`, `facet.args`, `xlab`, `ylab`, `...` | further arguments passed to `tinyplot`. |

## Details

Internally the time series object is converted to a long data frame with columns `Time` (time index), `Value` (observations), and `Series` (factor with column labels). Depending on the settings of `facet` this data frame is visualized either with the formula `Value ~ Time` or `Value ~ Time | Series`. See the `facet` argument description for more details and the Examples below for some illustrations.

An exception is made if the user explicitly supplies a distribution `type` argument (i.e, one of `"histogram"`, `"density"`, `"boxplot"`, `"violin"`, `"qq"`, `"ridge"`, or `"rug"`). These summarize the series values and ignore the time index, so they are visualized with the one-sided formula `~ Value` (or `~ Value | Series` for multivariate series). This preserves base-R-compatible behaviour such as `tinyplot(Nile, type = "histogram")`.

## Value

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

## Examples

```
## univariate series
tinyplot(Nile)

# exception: expicitly passing a distribution type on a univariate series
# still triggers the corresponding transformation
tinyplot(Nile, type = "histogram")

## multivariate series (generally, these also look better with a theme)
tinytheme("clean2")
tinyplot(EuStockMarkets)                    ## faceted, free scales, same color
tinyplot(EuStockMarkets, facet.args = NULL) ## faceted, same scale, same color
tinyplot(EuStockMarkets, facet = "by")      ## faceted, free scales, diff colors
tinyplot(EuStockMarkets, facet = NULL)      ## single frame, diff colors

## further variations
tinyplot(EuStockMarkets, facet = NULL, legend = list("direct", repel = TRUE))
tinyplot(EuStockMarkets, facet = "by", facet.args = NULL)
tinyplot(EuStockMarkets, facet.args = list(free = TRUE, ncol = 1))

## pass additional tinyplot args through `...` for further customization
tinyplot(EuStockMarkets,
         facet.args = NULL,
         type = "area", xlab = NA, yaxl = ",",
         main = "European stock indices")

tinytheme() ## reset
```

---

`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. Arguments are captured unevaluated and spliced into the updated call, so those |

that rely on non-standard evaluation against `data`—e.g. `subset = cyl == 4` or `weights = wt`—behave the same as in a direct `tinyplot` call.

## Value

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

## Limitations

- `tinyplot_add()` works reliably only when adding to a plot originally created using the `tinyplot.formula` method with a valid `data` argument. We cannot guarantee correct behavior if the original plot was created with the atomic `tinyplot.default` method, due to potential environment mismatches. (An exception is when the original plot arguments—`x`, `y`, etc.—are located in the global environment.)
- Automatic legends for the added elements will be turned off.

## Examples

```
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)

## Arguments relying on non-standard evaluation (e.g. `subset`) work too:
tinyplot(mpg ~ wt, data = mtcars)
tinyplot_add(subset = cyl == 4, col = "red", pch = 16)
```

---

# 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", "dynamic", "clean", "clean2", "bw", "linedraw",
   "classic", "minimal", "ipsum", "ipsum2", "dark", "socviz", "broadsheet", "nber",
    "web", "ridge", "ridge2", "tufte", "float", "void"),
  ...,
  register = NULL
)
```

## 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.
- `"dynamic"` (*): builds on `"basic"` by enabling dynamic margin adjustment with tighter default margins, horizontal axis labels, and the subtitle moved above the plotting area. Turns off the panel grid. Provides the foundation for all other dynamic themes and is a good starting point for users who wish to build custom dynamic themes.
  - `"clean"` (*): builds on `"dynamic"` by re-enabling the panel background grid and setting different default palettes ("Tableau 10" for discrete colors and "agSunset" for gradient colors).
    - `"clean2"` (*): removes the plot frame (box) from `"clean"`.
  - `"classic"` (*): builds on `"dynamic"` with L-shaped axes (removing the top and right-hand edges of the plot frame), smaller axis text, tighter axis spacing, and 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 fine grid lines, smaller axis text, tighter axis spacing, and sets the "Okabe-Ito" palette as a default for discrete colors. Inspired by the **ggplot2** theme of the same name.
    - `"linedraw"` (*): builds on `"bw"` with black facet title strips and white strip text. 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"` (*): builds on `"minimal"` with bold titles, no ticks, fine grid, edge-aligned axis titles, and a custom muted palette. Inspired by the **hrbrthemes** theme of the same name for **ggplot2**.

    - "ipsum2" (*): a lighter variant of "ipsum" that retains the original italic subtitle and edge-aligned axis titles, but without the additional spacing and palette changes.
    - "dark" (*): similar to "minimal", but set against a dark background with foreground and a palette colours lightened for appropriate contrast.
    - "socviz" (*): builds on "minimal" with L-shaped axes, very light grid lines, and larger axis text. Inspired by Kieran Healy's **socviz** package theme for **ggplot2**.
  - "broadsheet" (*): a publication/newspaper style with only horizontal grid lines, no frame, short x-axis ticks, and muted secondary text (subtitle, caption). Compact axis spacing.
    - "nber" (*): builds on "broadsheet" for an NBER working paper style with a light blue-grey background, grey text, italic axis titles and captions, and a blue-grey discrete palette.
  - "web" (*): a FiveThirtyEight-inspired style with a light grey device background, no frame or axis lines, and bold grid lines. Suited to web/online publication.
  - "tufte" (*): floating axes and minimalist plot artifacts in the style of Edward Tufte.
    - "float" (*): builds on "tufte" with outward ticks, fewer tick marks, and a "dark" qualitative palette.
  - "void" (*): switches off all axes, titles, legends, etc.
  - "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.

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

`register` Optional character string. If provided, the theme (with any `...` overrides) is registered under this name via `tinytheme_register()` and simultaneously activated. This is a shortcut for calling `tinytheme_register()` and `tinytheme()` separately.

## Details

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

**Persistent vs. ephemeral themes.** Calling `tinytheme("<theme>")` triggers a persistent theme, which will be applied to all subsequent graphics (incl. base `plot`). To reset the theme to your default settings, call `tinytheme()` without arguments. Alternatively, invoke the `tinyplot(..., theme = <theme>)` argument for an ephemeral theme that is automatically reset at the end of the plot call.

```
# Set a persistent theme
```

```
tinytheme("clean")
tinyplot(1:10)  # uses "clean"
tinyplot(10:1)  # still uses "clean"
tinytheme()     # reset to default

# Use an ephemeral theme for a single plot (incl. added components)
tinyplot(1:10, theme = "dark")  # uses "dark" ephemerally
tinyplot_add(type = "S")        # same plot, so retains theme
tinyplot(10:1)                  # new plot, so back to default
```

**Custom overrides.** To customize a theme's parameters (e.g., `mar`, `las`, etc.), either pass them directly as additional args to `tinytheme(<theme>, ...)` or as a list arguments to `tinyplot(..., theme = list(<theme>, ..`. Please note that passing overrides through `tpar()` or `par()` *won't* work, because themes work by installing a persistent hook, which means that an active theme will override any subsequent calls for parameters that the theme controls.

```
# Do this
tinytheme("clean", mar = c(5, 5, 2, 2))
<some plot>

# Or this (ephemeral version)
tinyplot(..., theme = list("clean", mar = c(5, 5, 2, 2)))

# Don't do this (the theme hook will overwrite your changes)
tinytheme("clean")
tpar(mar = c(5, 5, 2, 2))
<some plot>
```

**Spacing primitives.** Dynamic themes compute `mgp` (margin line positions) automatically from two spacing primitives, rather than requiring users to reason about how `mar`, `mgp`, and `tcl` combine:

- `gap.axis`: the gap in margin lines between the tick tip and the near edge of the tick label. Default `0.2`.
- `gap.lab`: the gap in margin lines between the far edge of the tick label and the near edge of the axis title. Default `1.0`.

These primitives scale automatically with `cex.axis` and `cex.lab`, so the visible spacing between elements remains constant regardless of text size. To adjust spacing, pass them as overrides:

```
# Tighter spacing between tick labels and axis titles
tinytheme("clean", gap.lab = 0.5)

# More room between ticks and tick labels
tinytheme("clean", gap.axis = 0.5)
```

If you supply an explicit `mgp` value, it is used as-is and the primitives are ignored.

## Value

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

## See Also

tpar which does the heavy lifting under the hood; tinytheme_register() for registering custom named themes.

## 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("dark")
p()

#  A set theme is persistent and will apply to subsequent plots
tinyplot(0:10)

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

# Customize the theme by overriding default settings
tinytheme("clean",
          adj.xlab = 1, adj.ylab = 1,
          cex.lab = 0.75, cex.axis = 0.9,
          font.sub = 3,
          gap.axis = 0, gap.lab = 0.5,
          tcl = -0.1)
p()

# Another custom theme example, including a different font
tinytheme("bw", font.main = 2, col.axis = "darkcyan", family = "HersheyScript")
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()

# For an ephemeral theme, use `tinyplot(..., theme = <theme>)` directly
tinyplot(0:10, theme = "clean", main = "This theme is ephemeral")
tinyplot(10:0, main = "See, no more theme")

# Customize an ephemeral theme by passing arguments as a list
tinyplot(0:10, main = "Custom", theme = list("clean", grid.col = "hotpink"))
```

```
# 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,
    yaxl = ",",
    main = paste0('tinytheme("', thm, '")'),
    sub = "A subtitle",
    cap = "A caption"
  )
  box("outer", lty = 2)
}

# Reset
tinytheme()
```

---

`tinytheme_register` *Register, List, and Unregister Custom Themes*

---

## Description

`tinytheme_register()` registers a custom theme so it can be used by name with `tinytheme()` or `tinyplot(..., theme = )`. Custom themes inherit from a base theme and apply user-specified overrides. Registered themes are session-scoped: they persist across plots but not across R sessions. To make a custom theme permanently available, register it in your `.Rprofile`.

`tinytheme_list()` returns the names of all available themes (built-in and registered).

`tinytheme_unregister()` removes a previously registered theme from the registry. Does not reset an active theme.

## Usage

```
tinytheme_register(name, theme = "default", ...)

tinytheme_list()

tinytheme_unregister(name)
```

## Arguments

| | |
|---|---|
| `name` | Character string. The name for your custom theme. Cannot clash with or overwrite a built-in theme name (`"default"`, `"clean"`, etc.) |

| | |
|---|---|
| theme | Character string or list. The base theme to inherit from. If a string, it must reference a built-in or previously-registered theme. If a list, it is used directly as the base definition. Default is "default". |
| ... | Named arguments to override specific theme settings. These are the same parameters accepted by tpar(). |

## Value

tinytheme_register() returns the theme definition list (invisibly). tinytheme_list() returns a named list with character vectors builtin and registered. tinytheme_unregister() returns NULL (invisibly).

## See Also

tinytheme()

## Examples

```
# Register a custom theme based on "float" but with a grid
tinytheme_register("float2", theme = "float", grid = TRUE)

# Use it
tinyplot(1:5, theme = "float2")

# List all themes
tinytheme_list()

# Unregister
tinytheme_unregister("float2")
```

---

# 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.cap: Numeric value between 0 and 1 controlling the alignment of the plot caption. Defaults to 0.5 (centered) for the default theme, and 1 (right-aligned) for all other themes.
- 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.
- cairo: Logical indicating whether cairo_pdf should be used when writing plots to PDF. If FALSE, then pdf will be used instead, with implications for embedding (non-standard) fonts. Only used if tinyplot(..., file = "<filename>.pdf") is called. Defaults to the value of capabilities("cairo").
- cex.cap: Numeric expansion factor for the plot caption text. Defaults to 1 for the default, basic, and dynamic themes, and 0.8 for clean/classic and their descendants.
- col.cap: Character specifying the colour of the plot caption. Defaults to "black".

- `col.default`: Default colour for single-group displays (i.e. plots without a by grouping). Can be `NULL`, a length-1 character colour, or a length-1 numeric index into `palette.qualitative`. Defaults to `NULL`, in which case the first colour of the active qualitative palette is used (or base `palette()[1]`, typically black, if no theme is set). A character value sets the single-group colour independently of the multi-group palette. A numeric value `i` selects the `i`th colour of `palette.qualitative` as the single-group default; a *negative* index additionally drops that colour from the palette used for grouped plots. For example, `col.default = -1` paired with `palette.qualitative = "Okabe-Ito"` uses black (the leading palette colour) for single-group plots and an Okabe-Ito-minus-black palette for grouped plots, avoiding the need to maintain a separate "no-black" palette.
- `font.cap`: Integer specifying the font face for the plot caption (`1` = plain, `2` = bold, `3` = italic, `4` = bold italic). Defaults to `1`.
- `line.cap`: Numeric specifying the margin line on which to draw the caption. If `NULL` (default), computed automatically based on the available bottom margin.
- `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.

- gap.main: Numeric giving the gap (in margin lines) from the main title baseline to the element below it (plot box when no subtitle; subtitle top when subtitle is present). Only used when dynmar = TRUE. Defaults to 0.7.
- gap.sub: Numeric giving the gap (in margin lines) from the subtitle baseline to the plot box. Only used when dynmar = TRUE and side.sub = 3. Defaults to 0.7.
- 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 or character indicating whether a background panel grid should be added to plots automatically. Defaults to NULL, which is equivalent to FALSE. In addition to logical values, a character string can be used to control axis-specific grids: uppercase letters ("X", "Y", "XY") draw grid lines at the standard axis tick positions (equivalent to TRUE), while lowercase letters ("x", "y", "xy") draw a finer grid with additional lines at the midpoints between ticks.
- 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 argument in ?tinyplot.
- palette.sequential: Palette for sequential colors. See the palette argument 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 evaluates 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

These functions add straight line(s) through the current plot.

## Usage

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

type_hline(h = 0)

type_vline(v = 0)
```

## Arguments

| | |
|---|---|
| a, b | the intercept (default: `a = 0`) and slope (default: `b = 1`) terms. Numerics of length 1, or equal to the number of groups or number of facets (or the product thereof). |
| h | y-value(s) for horizontal line(s). Numeric of length 1, or equal to the number of groups or number of facets (or the product thereof). |
| v | x-value(s) for vertical line(s). Numeric of length 1, or equal to the number of groups or number of facets (or the product thereof). |

## Details

While `type_abline`, `type_hline`, and `type_vline` can be called in a base plot layer, we expect that they will typically be called as subsequent layers via `tinyplot_add`.

### Recycling logic

The recycling behaviour of the line parameters (i.e., a, b, h, or v) is adaptive, depending on whether by or facet grouping is detected. While this leads to different recycling scenarios, the underlying code logic follows sensible heuristics designed to match user expectations.

Parameter lengths must equal one of four options:

1. Single value (i.e., length = 1), i.e. simplest case where the same line is applied uniformly across all groups and facets. Uses the default user colour (e.g. "black", or tpar("palette.qualitative")[1] if a theme is set).
2. Number of by groups, i.e. one parameter per group. For example, tinyplot(mpg ~ wt | factor(cyl), data = mtcars, type = type_hline(h = 21:23)) will give three horizontal lines, with colours matching the user's qualitative palette.
3. Number of facet groups, i.e. one parameter per facet panel. For example: tinyplot(mpg ~ wt, facet = ~am, data = mtcars, type = type_hline(h = c(20,30))) would give separate horizontal lines per facet, but both using the same default color.
4. Product of by and facet groups, i.e. one parameter for each unique by-facet combination. Orders over facets first and then, within that, by group. For example: tinyplot(mpg ~ wt | factor(cyl), facet = ~am, data = mtcars, type = type_hline(h = 21:26)) will give six separate lines, with the first three (21:23) coloured by group in the first facet, and second three (24:26) coloured by by group in the second facet.

Alongside these general rules, we also try to accommodate special cases when other aesthetic parameters like lwd or lty are invoked by the user. See Examples.

### Examples

```
#
## abline

tinyplot(x = -10:10, y = rnorm(21) + -10:10, grid = TRUE)
tinyplot_add(type = "abline")
# same as...
# tinyplot_add(type = type_abline(a = 0, b = 1))

# customize by passing bespoke intercept and slope values
tinyplot_add(type = type_abline(a = -1, b = -0.5))

# note that calling as abline & co. as a base plot layer will still lead to
# axes limits that respect the range of the data
tinyplot(x = -10:10, y = -10:10, grid = TRUE, type = "abline")

#
## hline and vline

# Base plot layer
tinyplot(mpg ~ hp | cyl, facet = "by", data = mtcars, ylim = c(0, 40))

# Add horizontal lines at the (default) 0 y-intercept
tinyplot_add(type = "hline", col = "grey")
```

```
# Note that group+facet aesthetics will be inherited. We can use this to
# add customized lines (here: the mean `mpg` for each `cyl` group)
tinyplot_add(type = type_hline(with(mtcars, tapply(mpg, cyl, mean))), lty = 2)

# Similar idea for vline
tinyplot_add(type = type_vline(with(mtcars, tapply(hp, cyl, mean))), lty = 2)

#
## Recycling logic

# length(h) == no. of groups
tinyplot(mpg ~ wt | factor(cyl), data = mtcars, type = type_hline(h = 21:23))

# length(h) == no. of facets
tinyplot(mpg ~ wt, facet = ~am, data = mtcars, type = type_hline(h = c(20, 30)))

# length(h) == no. of groups x no. of facets
tinyplot(mpg ~ wt | factor(cyl),
  facet = ~am, data = mtcars,
  type = type_hline(h = 21:26))

# special adjustment case (here: lwd by group)
tinyplot(mpg ~ wt | factor(cyl),
  facet = ~am, data = mtcars,
  type = type_hline(c(20, 30)), lwd = c(21, 14, 7))
```

---

`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, dodge = 0, fixed.dodge = FALSE)
```

## 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.) |
| `dodge` | Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either: |

- numeric value in the range `[0,1)`. Note that values are scaled relative to the spacing of x-axis breaks, e.g. `dodge = 0.1` places the outermost groups one-tenth of the way to adjacent breaks, `dodge = 0.5` places them midway between breaks, etc. Values < 0.5 are recommended.
- logical. If `TRUE`, the dodge width is calculated automatically based on the number of groups (0.1 per group for 2-4 groups, 0.45 for 5+ groups). If `FALSE` or 0, no dodging is performed.

Default value is 0 (no dodging). While we do not check, it is *strongly* recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks.

`fixed.dodge` Logical. If `FALSE` (default), dodge positions are calculated independently for each `x` value, based only on the groups present at that position. If `TRUE`, dodge positions are based on all groups, ensuring "fixed" spacing across x-axis breaks (i.e., even if some groups are missing for a particular `x` value).

## Dodging ribbon plots

We support dodging for grouped ribbon plots, enabling similar functionality to dodged errorbar and pointrange plots. However, it is strongly recommended that dodging is only implemented for cases where the x-axis comprises a limited number of discrete cases (e.g., coefficient or event-study plots). See Examples.

## 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")
```

```
#
## Dodged ribbon/area plots

# Dodged ribbon or area plots can be useful in cases where there is strong
# overlap across groups (and a limited number of discrete x-axis values).

dat = data.frame(
  x = rep(c("Before", "After"), each = 2),
  grp = rep(c("A", "B"), 2),
  y = c(10, 10.5, 15, 15.3),
  lwr = c(8, 8.5, 13, 13.3),
  upr = c(12, 12.5, 17, 17.3)
)

tinyplot(
  y ~ x | grp,
  data = dat,
  ymin = lwr, ymax = upr,
  type = type_ribbon(),
  main = "Overlappling ribbons"
)

tinyplot(
  y ~ x | grp,
  data = dat,
  ymin = lwr, ymax = upr,
  type = type_ribbon(dodge = 0.1),
  main = "Dodged ribbons"
)
```

---

`type_barplot` *Barplot type*

---

## Description

Type function for producing barplots. For formulas of type ~ x (without left-hand side) the barplot visualizes the counts (absolute frequencies) of the levels of x. For formulas of type y ~ x the value of y within each level of x is visualized, if necessary aggregated using some function (default: mean).

## Usage

```
type_barplot(
  width = 5/6,
  beside = FALSE,
  center = FALSE,
  offset = NULL,
  FUN = NULL,
  xlevels = NULL,
```

```
  xaxlabels = NULL,
  drop.zeros = FALSE,
  lighten = TRUE
)
```

## Arguments

| | |
|---|---|
| width | numeric, optional vector of bar widths. (The distance between the midpoints of the bars is always 1.) |
| beside | logical. In case of a by grouping variable, should bars be juxtaposed? Default is to use stacked bars instead. |
| center | logical or numeric. In case of stacked barplots (beside = FALSE) should the bars be centered (or all start at zero, default)? If set to TRUE the center is at the mid-point of the middle category (in case of uneven number of categories) or between the two middle categories (in case of an even number). Additionally it is possible to set center = 2 or center = 2.5 to indicate that centering should be after the second category or the mid-way in the third category, respectively. |
| offset | optional specification for shifting bar baselines, accepting one of two distinct forms. See the Examples for illustrations of both.<br>• *Positions* via an unnamed numeric scalar or vector. Bars start at the offset value(s) rather than zero, matched per x-level after any xlevels reordering (a scalar is applied to all bars). Useful for waterfall charts. The positional form cannot be combined with center.<br>• *Category* via a character vector such as offset = "Unsure", or a named numeric vector such as offset = c(Unsure = 1.1). The named level(s) of the by grouping are "set aside", i.e. pulled out of the (optionally centered) stack and drawn as standalone bars. This is useful for diverging/Likert plots where a neutral category (e.g. "Unsure") is shown apart from the diverging stack. The category form requires a by grouping and beside = FALSE, but can be combined with center. |
| FUN | a function to compute the summary statistic for y within each group of x in case of using a two-sided formula y ~ x (default: mean). |
| xlevels | a character or numeric vector specifying the ordering of the levels of the x variable (if character) or the corresponding indexes (if numeric) for the plot. |
| xaxlabels | a character vector with the axis labels for the x variable, defaulting to the levels of x. |
| drop.zeros | logical. Should bars with zero height be dropped? If set to FALSE (default) a zero height bar is still drawn for which the border lines will still be visible. |
| lighten | logical. Should the fills use a lighter, opaque tint of the series colour(s)? Default is TRUE, which keeps single- and multi-group displays consistent and lets the fill read cleanly over grid lines. Set to FALSE to use the fully-saturated palette colour(s) instead. |

## Examples

```
# Basic examples of frequency tables (without y variable)
```

```
tinyplot(~ cyl, data = mtcars, type = "barplot")
tinyplot(~ cyl | vs, data = mtcars, type = "barplot")
tinyplot(~ cyl | vs, data = mtcars, type = "barplot", beside = TRUE)

# Reorder x variable categories either by their character levels or numeric indexes
tinyplot(~ cyl, data = mtcars, type = "barplot", xlevels = c("8", "6", "4"))
tinyplot(~ cyl, data = mtcars, type = "barplot", xlevels = 3:1)

# Note: Above we used automatic argument passing for `beside`. But this
# wouldn't work for `width`, since it would conflict with the top-level
# `tinyplot(..., width = <width>)` argument. It's safer to pass these args
# through the `type_barplot()` functional equivalent.
tinyplot(
  ~ cyl | vs, data = mtcars,
  type = type_barplot(beside = TRUE, drop.zeros = TRUE, width = 0.65)
)

# Example for numeric y aggregated by x (default: FUN = mean) + facets
tinyplot(
  extra ~ ID | group, facet = "by", data = sleep,
  type = "barplot",
  theme = "clean2"
)

# Fancy frequency table:
tinyplot(
  Freq ~ Sex | Survived, data = as.data.frame(Titanic),
  facet = ~ Class, facet.args = list(nrow = 1),
  type = "barplot", flip = TRUE,
  theme = "clean2"
)

# Centered barplot for conditional proportions of dark (black/brown) vs.
# light (red/blond) hair color, conditional on eye color and sex.
# Aside: use `lighten = FALSE` to avoid lightening the bar fill colors.
hec = as.data.frame(proportions(HairEyeColor, 2:3))
hcols = c("black", "sienna", "indianred", "goldenrod")
tinyplot(
  Freq ~ Eye | Hair, data = hec,
  facet = ~ Sex, facet.args = list(ncol = 1),
  type = type_barplot(center = TRUE, lighten = FALSE),
  flip = TRUE, yaxl = "percent",
  theme = list("clean2", palette.qualitative = hcols)
)

# Use cases for the `offset` argument

# 1. Waterfall plot
d = data.frame(item = c("Sales", "Services", "Costs", "Returns", "TOTAL"),
               value = c(100, 40, -80, -10, 50))
d$item = factor(d$item, levels = d$item)
d$offset = c(0, cumsum(d$value[1:3]), 0)
tinyplot(
```

```
  value ~ item | I(value < 0), data = d,
  type = type_barplot(offset = d$offset, lighten = FALSE),
  col = NA, # (optional: turn off border)
  legend = FALSE
)
tinyplot_add(type = type_vline(4.5), lty = 2, col = "grey50")

# 2. Diverging/Likert layout: a character (or named numeric) offset "sets
# aside" the named category, pulling it out of the centered stack and drawing
# it as a standalone bar. Here a neutral "Unsure" response is shown apart from
# the diverging agree/disagree scale.
lik = expand.grid(
  question = c("Pay", "Workload", "Manager", "Culture"),
  response = c("Strong disagree", "Disagree", "Agree", "Strong agree", "Unsure")
)
lik$response = factor(lik$response, levels = unique(lik$response))
lik$share = c( # proportions summing to 1 within each question
  .10, .25, .05, .15,
  .20, .30, .15, .20,
  .35, .20, .40, .30,
  .25, .15, .35, .20,
  .10, .10, .05, .15
)
# diverging palette: reds (disagree) -> blues (agree), grey for "Unsure"
pal = c("#b2182b", "#ef8a62", "#67a9cf", "#2166ac", "grey")
tinyplot(
  share ~ question | response, data = lik,
  type = type_barplot(center = TRUE, offset = "Unsure", lighten = FALSE),
  flip = TRUE, xlab = NA, ylab = NA, yaxl = "percent",
  legend = list("top!", title = FALSE),
  theme = list("clean2", palette.qualitative = pal),
  main = "Hypothetical Likert example with category offset"
)
tinyplot_add(type = "vline")
```

---

`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,
  lighten = TRUE
)
```

## 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. |
| lighten | logical. Should the fills use a lighter, opaque tint of the series colour(s)? Default is TRUE, which keeps single- and multi-group displays consistent and lets the fill read cleanly over grid lines. Set to FALSE to use the fully-saturated palette colour(s) instead. |

## Examples

```
# "boxplot" type convenience string
tinyplot(weight ~ feed, data = chickwts, 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(weight ~ feed, data = chickwts)

# For flipped boxplots, it's usually better to use a dynamic theme to
# accommodate (horizontal) y-axis labels
tinyplot(weight ~ feed, data = chickwts, flip = TRUE, theme = "dynamic")

# Grouped boxplot example using a different dataset
# (C.f., the final example in `?boxplot`)
tinyplot(
```

```
  len ~ dose | supp, data = ToothGrowth, type = "boxplot",
  main = "Guinea Pigs' Tooth Growth",
  legend = list(title = "Supplement"),
  xlab = "Vitamin C dose mg", ylab = "tooth length"
)

# Use `type_boxplot()` to pass extra arguments for customization
tinyplot(
  len ~ dose | supp, data = ToothGrowth,
  type = type_boxplot(boxwex = 0.3, staplewex = 0, outline = FALSE), lty = 1,
  legend = list(title = "Supplement"),
  main = "Guinea Pigs' Tooth Growth",
  xlab = "Vitamin C dose mg", ylab = "tooth length"
)
```

---

`type_chull` *Convex hull plot type*

---

## Description

Type function for drawing convex hulls around grouped points. Uses `chull` to compute the convex hull of each group and draws the result as a polygon.

## Usage

```
type_chull(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

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

# layer filled convex hull(s) on top of points
tinyplot(Sepal.Length ~ Petal.Length | Species, data = iris, theme = "basic")
tinyplot_add(type = "chull", fill = 0.2)
```

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, see density for details and options. |
| 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).<br>"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_ellipse *Confidence ellipse plot type*

---

## Description

Type function for drawing confidence ellipses around grouped points.

## Usage

```
type_ellipse(level = 0.95, segments = 100, density = NULL, angle = 45)
```

## Arguments

| | |
|---|---|
| level | Numeric confidence level for the ellipse. Default is 0.95. |
| segments | Integer number of line segments used to approximate the ellipse. Default is 100. |
| 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). |

## Details

A confidence ellipse summarizes the spread and correlation of two variables: its axes align with the principal directions of variation, their lengths reflect the standard deviations along those directions, and the overall size is scaled so that (assuming bivariate normality) the specified proportion of observations falls within the boundary.

Operationally, the ellipse is computed from the covariance matrix (cov) of each group via eigendecomposition (eigen), scaled by a chi-squared quantile (qchisq) for the specified confidence level. Groups with fewer than 3 points are silently dropped.

## Examples

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

# layer filled ellipse(s) on top of points
tinyplot(Sepal.Length ~ Petal.Length | Species, data = iris, theme = "basic")
tinyplot_add(type = "ellipse", fill = 0.2)

# to avoid clipping, it's better to reverse the layering (ellipse first)
tinyplot(Sepal.Length ~ Petal.Length | Species, data = iris, theme = "basic",
         type = "ellipse", fill = 0.2)
tinyplot_add(type = "p")

# custom confidence level
tinyplot(Sepal.Length ~ Petal.Length | Species, data = iris,
  type = type_ellipse(level = 0.5))
```

---

# 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, dodge = 0, fixed.dodge = FALSE)

type_pointrange(dodge = 0, fixed.dodge = FALSE)
```

## Arguments

| | |
|---|---|
| length | length of the edges of the arrow head (in inches). |
| dodge | Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either: |

- numeric value in the range `[0,1)`. Note that values are scaled relative to the spacing of x-axis breaks, e.g. `dodge = 0.1` places the outermost groups one-tenth of the way to adjacent breaks, `dodge = 0.5` places them midway between breaks, etc. Values < 0.5 are recommended.
- logical. If `TRUE`, the dodge width is calculated automatically based on the number of groups (0.1 per group for 2-4 groups, 0.45 for 5+ groups). If `FALSE` or 0, no dodging is performed.

Default value is 0 (no dodging). While we do not check, it is *strongly* recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks.

`fixed.dodge` Logical. If `FALSE` (default), dodge positions are calculated independently for each `x` value, based only on the groups present at that position. If `TRUE`, dodge positions are based on all groups, ensuring "fixed" spacing across x-axis breaks (i.e., even if some groups are missing for a particular `x` value).

## Examples

```
tinytheme("basic")

#
## Basic coefficient plot(s)

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

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

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

#
## Flipped plots

# For flipped errobar / pointrange plots, it is recommended to use a dynamic
# theme that applies horizontal axis tick labels

tinytheme("classic")
tinyplot(est ~ term, ymin = lwr, ymax = upr, data = coefs, type = "errorbar",
         flip = TRUE)
tinyplot_add(type = 'vline', lty = 2)

tinytheme("basic") # back to basic theme for the remaining examples

#
## Dodging groups

models = list(
```

```
    "Model A" = lm(mpg ~ wt, data = mtcars),
    "Model B" = lm(mpg ~ wt + cyl, data = mtcars),
    "Model C" = lm(mpg ~ wt + cyl + hp, data = mtcars)
)

models = do.call(
  rbind,
  lapply(names(models), function(m) {
    data.frame(
      model = m,
      term = names(coef(models[[m]])),
      estimate = coef(models[[m]]),
      setNames(data.frame(confint(models[[m]])), c("conf.low", "conf.high"))
    )
  })
)

tinyplot(estimate ~ term | model,
         ymin = conf.low, ymax = conf.high,
         data = models,
         type = type_pointrange(dodge = 0.1))

# Aside 1: relative vs fixed dodge
#  The default dodge position is based on the unique groups (here: models)
#  available to each x value (here: coefficient term). To "fix" the dodge
#  position across all x values, use `fixed.dodge = TRUE`.

tinyplot(estimate ~ term | model,
         ymin = conf.low, ymax = conf.high,
         data = models,
         type = type_pointrange(dodge = 0.1, fixed.dodge = TRUE))

# Aside 2: layering
#  For layering on top of dodged plots, rather pass the dodging arguments
#  through the top-level call if you'd like the dodging behaviour to be
#  inherited automatically by the added layers.

tinyplot(estimate ~ term | model,
         ymin = conf.low, ymax = conf.high,
         data = models,
         type = "pointrange",
         dodge = 0.1, fixed.dodge = TRUE)
tinyplot_add(type = "l", lty = 2)

tinytheme() # reset theme
```

---

## 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 (default function)
tinyplot(x = -4:4, type = "function")
# tinyplot(x = -4:4, type = type_function()) # same

# Customize by passing explicit arguments to your function
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 `function(x)`, but you could also use the shorter `\(x)`
## anonymous function syntax introduced in R 4.1.0)
tinyplot(x = -4:4, type = type_function(fun = function(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",
  weights = NULL
)
```

## 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. |
| weights | an optional numeric vector of observation weights for the model fit, of the same length as the number of data points. Weights can also be supplied via the top-level weights argument of tinyplot (which is evaluated with non-standard evaluation in the formula method, and takes precedence if both are given). |

## 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:<br>• a vector giving the breakpoints between histogram cells,<br>• a function to compute the vector of breakpoints,<br>• a single number giving the number of cells for the histogram,<br>• a character string naming an algorithm to compute the number of cells (see 'Details' of `hist`),<br>• 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_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; used in calculation of `amount` when the provided `amount` is 0 or `NULL`. |
| `amount` | number or `NULL`; when positive, specifies the amount of jitter; see Details, also for other cases. |

## Details

The result of `r <- x + amount * runif(n, -1, 1)` where `n <- length(x)`.

When `amount` is `NULL` (the default) `amount <- factor * d/5` where $d$ is about the smallest difference between adjacent unique (up to fuzz) `x` values.

If `amount == 0`, `amount <- factor * (max(x) - min(x))/50`, (using only finite `x` values), which is the same as in S and the references. In the above cases, the final `amount` is always ensured to be positive.

The short `jitter()` function is itself a succinct definition.

## 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", dodge = 0, fixed.dodge = FALSE)
```

## 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. |
| dodge | Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either:<br>• numeric value in the range [0,1). Note that values are scaled relative to the spacing of x-axis breaks, e.g. dodge = 0.1 places the outermost groups one-tenth of the way to adjacent breaks, dodge = 0.5 places them midway between breaks, etc. Values < 0.5 are recommended.<br>• logical. If TRUE, the dodge width is calculated automatically based on the number of groups (0.1 per group for 2-4 groups, 0.45 for 5+ groups). If FALSE or 0, no dodging is performed.<br>Default value is 0 (no dodging). While we do not check, it is *strongly* recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks. |
| fixed.dodge | Logical. If FALSE (default), dodge positions are calculated independently for each x value, based only on the groups present at that position. If TRUE, dodge positions are based on all groups, ensuring "fixed" spacing across x-axis breaks (i.e., even if some groups are missing for a particular x value). |

## 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"))

# Direct legend labels are a good option for grouped lined plots (assuming
# there aren't too many groups and the data are sorted along the x-axis)
tinyplot(
```

```
  circumference ~ age | Tree, data = Orange, type = "l",
  legend = "direct"
)

# Fancier version(s) that use a theme and repel overlapping labels
Orange2 = transform(Orange, Tree = paste("Tree", Tree))
tinyplot(
  circumference ~ age | Tree, data = Orange2, type = "l",
  legend = list("direct", repel = TRUE), # auto repel
  theme = "socviz"
)
tinyplot(
  circumference ~ age | Tree, data = Orange2, type = "l",
  legend = list("direct", nudge_y = c("Tree 1" = 3, "Tree 3" = -5)), # manual
  theme = "socviz"
)
```

---

`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, weights = NULL)
```

## Arguments

| | |
|---|---|
| `se` | logical. If TRUE, confidence intervals are drawn. |
| `level` | the confidence level required. |
| `weights` | an optional numeric vector of observation weights for the model fit, of the same length as the number of data points. Weights can also be supplied via the top-level `weights` argument of `tinyplot` (which is evaluated with non-standard evaluation in the formula method, and takes precedence if both are given). |

## 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))
```

```
# Weighted fit example, either as a top-level argument (which supports
# non-standard evaluation in the formula method)...
s77 = as.data.frame(state.x77)
tinyplot(`Life Exp` ~ Income, data = s77, type = "lm", weights = Population)
# ...or directly via the type constructor (requires an evaluated vector)
tinyplot(`Life Exp` ~ Income, data = s77, type = type_lm(
  weights = s77$Population))
```

---

`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,
  weights = NULL
)
```

## Arguments

| | |
|---|---|
| `span` | the parameter $\alpha$ 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` | 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.) |
| `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. |
| `weights` | an optional numeric vector of observation weights for the model fit, of the same length as the number of data points. Weights can also be supplied via the top-level `weights` argument of `tinyplot` (which is evaluated with non-standard evaluation in the formula method, and takes precedence if both are given). |

## 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(clim = c(0.5, 2.5), dodge = 0, fixed.dodge = FALSE)
```

## Arguments

| | |
|---|---|
| clim | Numeric giving the lower and upper limits of the character expansion (cex) normalization for bubble charts. |
| dodge | Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either:<br>• numeric value in the range [0,1). Note that values are scaled relative to the spacing of x-axis breaks, e.g. dodge = 0.1 places the outermost groups one-tenth of the way to adjacent breaks, dodge = 0.5 places them midway between breaks, etc. Values < 0.5 are recommended.<br>• logical. If TRUE, the dodge width is calculated automatically based on the number of groups (0.1 per group for 2-4 groups, 0.45 for 5+ groups). If FALSE or 0, no dodging is performed.<br>Default value is 0 (no dodging). While we do not check, it is *strongly* recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks. |
| fixed.dodge | Logical. If FALSE (default), dodge positions are calculated independently for each x value, based only on the groups present at that position. If TRUE, dodge positions are based on all groups, ensuring "fixed" spacing across x-axis breaks (i.e., even if some groups are missing for a particular x value). |

## 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)

# Bubble chart version
tinyplot(Sepal.Length ~ Petal.Length, data = iris, cex = iris$Sepal.Width)

# Fancier version with dual legends and extra customization
tinyplot(Sepal.Length ~ Petal.Length | Species,
  data = iris,
  cex = iris$Sepal.Width, clim = c(1, 5),
  pch = 21, fill = 0.3)
```

---

`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, see density for details and options.

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. If left unset here, the outline color can also be supplied via the top-level tinyplot(..., col =) call (the type_ridge(col =) argument takes precedence if both are given). When neither is supplied, the outline defaults to the active theme's col.default (see tpar), falling back to the first color of the theme's qualitative palette, or black if no theme is set.

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
# dynamic 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,
  xlevels = NULL,
  ylevels = NULL,
  col = NULL,
  xaxlabels = NULL,
  yaxlabels = NULL,
  weights = NULL,
  lighten = FALSE
)
```

## 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. |
| `xlevels, ylevels` | a character or numeric vector specifying the ordering of the levels of the `x` and `y` variables (if character) or the corresponding indexes (if numeric) for the plot. |
| `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.

lighten — logical. For grouped spineplots where the y variable is itself the grouping variable (i.e. y == by), should the fills use a lighter, opaque tint of the series colour(s)? Default is FALSE, i.e. the fills use the fully-saturated palette colour(s). (Unlike the other area types such as type_barplot, where lightening is the default, spineplot tiles abut one another with no gap, so the darker saturated fills read better against their matching border colours.) Set to TRUE to opt in to the lighter tint. Note that lighten has no effect on other spineplot displays (single-group or x == by), which always use a sequential shading ramp of the base colour.

## 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)
)

# Passing custom colors to the y-axis categories
tinyplot(
  Species ~ Sepal.Width, data = iris,
  type = type_spineplot(breaks = 4, col = palette.colors(3, "Pastel 1"))
)

# 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)

# Note: The Titanic (ttnc) dataset is pre-tabulated, so we pass its frequency
# counts via the top-level `weights` argument (accepted via non-standard
# evaluation in the formula method).
tinyplot(
  Survived ~ Sex, facet = ~ Class, data = ttnc,
  # type_spineplot(weights = ttnc$Freq), ## same thing but not NSE
  type = "spineplot", weights = Freq
```

```
)

# Reorder x and y variable categories either by their character levels or
# numeric indexes. (Here we combine a top-level `weights` with constructor-
# level arguments passed through `type_spineplot()`.)
tinyplot(
  Survived ~ Sex, facet = ~ Class, data = ttnc,
  type = type_spineplot(xlevels = c("Female", "Male"), ylevels = 2:1),
  weights = Freq
)

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

# Fancier version. Note the smart inheritance of spacing etc.
tinyplot(
  Survived ~ Sex | Class, data = ttnc,
  facet = "by", facet.args = list(nrow = 1),
  type = "spineplot", weights = Freq,
  theme = "void", axes = "t", lty = 0, legend = FALSE,
  main = "Who survived the Titanic disaster?",
  sub = "Frequencies by boarding class and sex"
)

# Aside: 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 = "spineplot", weights = 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

See spline for further details on the available interpolation methods.

## 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. For example, plot the mean y value for each x value. 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 = "summary")

# Note: "mean" is the default function, so these are also equivalent:
# tinyplot(weight ~ Time, data = ChickWeight, type = type_summary())
# 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 = "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 at the specified (x,y) coordinates.

### Usage

```
type_text(
  labels = NULL,
  labeller = NULL,
  adj = NULL,
  pos = NULL,
  offset = 0.5,
  family = NULL,
  font = NULL,
```

```
  vfont = NULL,
  xpd = NULL,
  srt = 0,
  repel = FALSE,
  clim = c(0.5, 2.5)
)
```

## Arguments

| | |
|---|---|
| labels | Character vector of length 1 or the same length as the number of x,y coordinates. If left as NULL, then the labels will automatically inherit the corresponding y values. Can also be supplied as a top-level tinyplot argument, which additionally supports non-standard evaluation against data and takes precedence if both are given. See Examples. |
| labeller | A formatting function (or convenience string) passed to tinylabel for formatting the labels. Useful for ensuring that the text labels match the formatting of an axis, e.g. labeller = "%" to display the labels as percentages. Default is NULL, i.e. no formatting. See Examples. |
| 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. |
| family | The name of a font family. Default of NULL means that the family will be the same as the main plot text, following par. Note that if a family argument is provided, then vfont (below) will automatically be ignored. |
| font | Integer giving the font face to be used, following par. On most devices, the mapping is: 1 = regular, 2 = bold, 3 = italic, 4 = bold italic, and 5 = symbol. |
| 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. |
| xpd | Logical value or NA denoting text clipping behaviour, following par. |
| srt | Numeric giving the desired string rotation in degrees. |
| repel | Logical or numeric controlling automatic repulsion of overlapping text labels. The default FALSE draws the labels at their exact (x,y) coordinates. TRUE nudges overlapping labels apart using a force-directed algorithm (labels are pushed off each other and then sprung back toward their original positions). A numeric value does the same but additionally enforces that much minimum padding (in user coordinates) between labels. Works best with the default centered text placement (i.e. without pos). **Caveat:** The repulsion logic currently operates within each group rather than across groups. So the text of different groups may still overlap. See Examples. |
| clim | Numeric giving the lower and upper limits of the character expansion (cex) normalization for bubble charts. |

## Examples

```
# simplest case (no labels), will auto revert to y labels
tinyplot(1:12, type = "text")

# pass explicit `labels` arg if you want specific text
tinyplot(1:12, type = "text", labels = month.abb)

# you can also use a labeller function (passed to `tinylabel`) to
# customize
tinyplot(1:12, type = "text", labels = month.abb, labeller = toupper)

# tip: use `xpd = NA` to avoid clipping text at the plot region
tinyplot(1:12, type = "text", labels = month.name, xpd = NA)

# for advanced customization, it's safer to pass args through `type_text()`
tinyplot(
  1:12,
  type = type_text(
  labels = month.abb,
  family = "HersheyScript",
  srt = -20)
)

# some formula + data.frame examples...

mt = transform(mtcars[1:20,], cyl = factor(cyl), model = rownames(mtcars[1:20,]))

# as a top-level argument, `labels` supports non-standard evaluation, so a
# bare column name is resolved against `data` (just like the plot variables)

# tinyplot(mpg ~ wt, data = mt, type = type_text(labels = mt$model))
tinyplot(mpg ~ wt, data = mt, type = "text", labels = model) # same

# tip: use `repel = TRUE` to automatically nudge overlapping labels apart
tinyplot(
  mpg ~ wt, data = mt,
  type = "text", labels = model,
  repel = TRUE, xpd = NA
)

# limitation: `repel` logic currently works per group, so grouped text data
# may still overlap
tinyplot(
  mpg ~ wt | cyl, data = mt,
  type = "text", labels = model,
  repel = TRUE, xpd = NA
)

# an advantage of the top-level `labels` approach is that data subsetting and
# sanitizing ops pass through automatically
tinyplot(hp ~ wt, data = mt)
tinyplot_add(subset = carb==2, col = "red", pch = 16)
```

```
tinyplot_add(subset = carb==2, col = "red", type = "text", labels = model, pos = 4)

# fancier plot that combines a theme and some other customizations
tinyplot(
  mpg ~ hp | cyl,
  data = mt,
  legend = list(title = "Cylinders"),
  theme = "web"
)
tinyplot_add(
  labels = model,
  type = type_text(
    family = "HersheySans",
    font = 2,
    pos = 4,
    repel = TRUE,
    xpd = NA
  )
)
```

---

`type_violin` *Violin plot type*

---

## Description

Type function for violin plots, which are an alternative to box plots for visualizing continuous distributions (by group) in the form of mirrored densities.

## Usage

```
type_violin(
  bw = "nrd0",
  joint.bw = c("mean", "full", "none"),
  adjust = 1,
  kernel = c("gaussian", "epanechnikov", "rectangular", "triangular", "biweight",
    "cosine", "optcosine"),
  n = 512,
  trim = FALSE,
  width = 0.9,
  lighten = TRUE
)
```

## Arguments

| | |
|---|---|
| `bw` | the smoothing `bandwidth` to be used, see `density` for details and options. |
| `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.

trim — logical indicating whether the violins should be trimmed to the range of the data. Default is FALSE.

width — numeric (ideally in the range [0, 1], although this isn't enforced) giving the normalized width of the individual violins.

lighten — logical. Should the fills use a lighter, opaque tint of the series colour(s)? Default is TRUE, which keeps single- and multi-group displays consistent and lets the fill read cleanly over grid lines. Set to FALSE to use the fully-saturated palette colour(s) instead.

## Details

See type_density for more details and considerations related to bandwidth selection and kernel types.

## Examples

```
# "violin" type convenience string
tinyplot(weight ~ feed, data = chickwts, type = "violin")

# to match the defaults of `ggplot2::geom_violin()`, use `trim = TRUE` and
# `joint.bw = FALSE`
tinyplot(
  weight ~ feed, data = chickwts,
  # type = type_violin(trim = TRUE, joint.bw = FALSE) # same but see final ex.
  type = "violin", trim = TRUE, joint.bw = FALSE
)

# For flipped violin plots, it's usually better to use a dynamic theme to
# accommodate (horizontal) y-axis labels
tinyplot(
  weight ~ feed, data = chickwts, type = "violin", flip = TRUE,
```

```
  theme = "dynamic" # or "clean(2)", "classic", "minimal", etc.
)

# you can group by the x var to add colour (here with the original orientation)
tinyplot(weight ~ feed | feed, data = chickwts, type = "violin", legend = FALSE)

# dodged grouped violin plot example (different dataset)
tinyplot(len ~ dose | supp, data = ToothGrowth, type = "violin")

# note: above we relied on `...` argument passing alongside the "violin"
# type convenience string. But this won't work for `width`, since it will
# clash with the top-level `tinyplot(..., width = <width>)` arg. To ensure
# correct arg passing, it's safer to use the functional `type_violin()` type.
tinyplot(
  len ~ dose | supp, data = ToothGrowth,
  type = type_violin(width = 0.75)
)
```

# Index


abline, *6*, *19*
abs, *8*, *16*
approx, *50*, *69*, *77*, *82*
ave, *77, 78*

bandwidth, *50*, *69*, *81*
boxplot, *47*
boxplot.stats, *48*

cairo_pdf, *37*
chull, *49*
cov, *53*

data.frame, *25*
density, *50*, *69*, *81*
dev.off, *6*
dnorm, *56*
draw_legend, 3, *6*

eigen, *53*

family, *57*, *63*
fft, *50*, *69*, *82*
format, *8*, *16*
formula, *20*, *25*

get_saved_par, 5, *6*, *19*, *39*
glm, *57*
graphics::par, *39*
gray.colors, *74*

Hershey, *79*
hist, *58*, *74*

interaction, *12*

jitter, *60*, *72*
jpeg, *19*

legend, *3*
list, *77*

lm, *62*
loess, *63*
loess.control, *63*

matplot, *27*
matrix, *26*
mean, *78*

options, *37*

pairs, *25*
palette, *17*
par, *6*, *12*, *16*, *19*, *36, 37*, *79*
pdf, *19*, *37*
plot, *9*, *13*, *20*, *61*
plot.density, *51*
plt *(*tinyplot*)*, 9
plt_add *(*tinyplot_add*)*, 29
png, *19*
polygon, *65*, *69*
polypath, *66*

qchisq, *53*

rasterImage, *69, 70*
rect, *67*

segments, *73*
spline, *76, 77*
svg, *19*

text, *6*, *19*
tinylabel, *3*, 7, *16*, *79*
tinypairs *(*tinyplot.data.frame*)*, 25
tinyplot, *3, 4*, *6*, *8*, 9, *30*, *57*, *62, 63*, *67*, *73, 74*, *79*
tinyplot.data.frame, 25
tinyplot.default, *30*
tinyplot.formula, *25*, *30*
tinyplot.matrix, 26
tinyplot.ts, 28

tinyplot_add, *19*, *29*, 29, *40*
tinyplot_add(), *73*
tinytheme, *16*, *20*, 30, *39*
tinytheme(), *36*
tinytheme_list *(*tinytheme_register*)*, 35
tinytheme_register, 35
tinytheme_register(), *32*, *34*
tinytheme_unregister *(*tinytheme_register*)*, 35
toupper, *8*, *16*
tpar, *13*, *16*, *19*, *34*, 36, *69*
ts, *28*
type_abline, 40
type_abline(), *15*
type_area, 42
type_area(), *14*
type_barplot, 44, *75*
type_barplot(), *14*
type_boxplot, 47
type_boxplot(), *14*
type_chull, 49
type_chull(), *14*
type_density, 50, *69*, *82*
type_density(), *14*
type_ellipse, 52
type_errorbar, 53
type_errorbar(), *14*
type_function, 55
type_function(), *15*
type_glm, 57
type_glm(), *15*
type_hist, *12*
type_hist *(*type_histogram*)*, 58
type_histogram, 58
type_histogram(), *14*
type_hline *(*type_abline*)*, 40
type_hline(), *15*
type_jitter, 60
type_jitter(), *14*
type_lines, 61, *77*
type_lm, 62
type_lm(), *12*, *14*
type_loess, 63
type_loess(), *14*
type_pointrange *(*type_errorbar*)*, 53
type_pointrange(), *14*
type_points, 64
type_points(), *18*
type_polygon, 65
type_polygon(), *14*
type_polypath, 66
type_polypath(), *14*
type_qq, 66
type_qq(), *14*
type_rect, 67
type_rect(), *14*
type_ribbon *(*type_area*)*, 42
type_ribbon(), *14*
type_ridge, 68
type_ridge(), *14*, *32*
type_rug, 72
type_rug(), *14*
type_segments, 73
type_segments(), *14*
type_spineplot, 74
type_spineplot(), *12*, *14*
type_spline, 76
type_spline(), *15*
type_summary, 77
type_summary(), *15*
type_text, 78
type_text(), *12*, *14*
type_violin, 81
type_violin(), *14*
type_vline *(*type_abline*)*, 40
type_vline(), *15*

xy.coords, *12*