

Package ‘tinypplot’

September 28, 2026

Type Package

Title Lightweight Extension of the Base R Graphics System

Version 0.8.0

Date 2026-09-27

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/tinypplot/>

BugReports <https://github.com/grantmcdermott/tinypplot/issues>

Config/roxygen2/version 8.1.0

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],

Miura Meng [ctb] (ORCID: <<https://orcid.org/0009-0004-1522-1997>>)

Maintainer Grant McDermott <contact@grantmcdermott.com>

Repository CRAN

Date/Publication 2026-09-28 00:00:02 UTC

Contents

draw_legend	3
get_saved_par	6
recordedtinypplot	8
tinylabel	9
tinypplot	11
tinypplot.array	29
tinypplot.data.frame	32
tinypplot.matrix	34
tinypplot.ts	35
tinypplot_add	37
tinyptheme	39
tinyptheme_get	44
tinyptheme_register	45
tpar	46
type_abline	51
type_area	53
type_barplot	58
type_boxplot	64
type_chull	65
type_density	66
type_ellipse	69
type_errorbar	70
type_function	74
type_glm	75
type_hexbin	76
type_histogram	78
type_jitter	80
type_lines	81
type_lm	84
type_loess	85
type_points	86
type_polygon	88
type_polypath	89
type_qq	90
type_rect	90
type_ridge	91
type_rug	96
type_segments	97
type_spineplot	98
type_spline	101
type_summary	102
type_text	103
type_tile	106
type_violin	110

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,  
  type_hints = 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 <code>legend(legend = ...)</code> .

labeller	Function, character keyword, or dictionary (named vector or list) for formatting or relabelling the labels (lgnl_labs). See tinylabel for the accepted forms. 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 <code>c(inner, outer)</code> , 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 <code>tpar("lmar")</code> for which the default values are <code>c(1.0, 0.1)</code> .
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 <code>plot.new</code> 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 <code>dynmar_computed[3]</code> value for "top!" legends under dynmar themes. When set, the legend margin formula uses this directly to ensure correct title positioning.
type_hints	Optional named list of semantic behaviour properties that a plot type declares for itself (e.g. <code>has_rhs_axis</code> for a secondary right-hand axis, or <code>legend_fills_from_col</code> for a legend swatch fill taken from <code>col</code>). Passed down from tinyplot ; defaults to NULL. Deliberately last, so that adding it did not shift the position of any pre-existing argument for positional callers.

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 `tinypplot` 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 `tinypplot` call.

Details

A potential side-effect of `tinypplot` 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 `tinypplot(..., 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 every new `tinypplot` 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 `tinypplot` 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)

```

recordedtinypplot	<i>Recorded tinypplot objects</i>
-------------------	-----------------------------------

Description

Objects of class `recordedtinypplot` are returned by `tinypplot(..., record = TRUE)`. They are a thin wrapper around `recordPlot`—thus conferring the same replay functionality—but with the initializing plot call and state added, so that recorded (tiny)plots play nicely with `tinypplot_add()` and friends.

Usage

```

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

## S3 method for class 'recordedtinypplot'
str(object, ...)

```

Arguments

<code>x</code>	a <code>recordedtinypplot</code> object, returned by a <code>tinypplot(..., record = TRUE)</code> call.
<code>...</code>	further arguments passed to <code>replayPlot</code> .
<code>object</code>	a <code>recordedtinypplot</code> object.

Details

Recording requires a device whose display list is enabled (see `dev.control`); without one there is nothing to record and the returned plot will replay blank. Interactive devices—the plot pane of an IDE, say—typically enable it by default, whereas file-based devices (png, pdf, svg, ...) do not. `tinypplot()` enables it for any device that it opens itself via the `file` argument, and warns if the current device is not recording.

Printing a recorded plot—either explicitly with `print()`, or simply by evaluating it at the console—is what restores the initializing call, so that a subsequent `tinypplot_add()` layers onto it. Calling `replayPlot()` on the object instead redraws it just as it would any other recording, but leaves the current plot context untouched; layering after that route targets whichever plot was drawn last.

This class is experimental, as is the `record` argument that produces it.

Value

`print()` replays the plot and returns `x` invisibly.

See Also

[recordPlot](#) and [replayPlot](#), which this class wraps and defers to for the actual recording and replaying. [dev.control](#) for enabling a device's display list, without which there is nothing to record.

Examples

```
## Not run:
p = tinypLOT(Sepal.Length ~ Petal.Length | Species, data = iris,
             record = TRUE)
tinypLOT(1:10) # some other plot
p             # replay, restoring plot context
tinypLOT_add(type = "lm") # layers onto p, not onto the 1:10 plot

## End(Not run)
```

tinylabel	<i>Format labels</i>
-----------	----------------------

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

- | | |
|-----------------------|---|
| <code>x</code> | a numeric or character vector |
| <code>labeller</code> | how the labels should be relabelled or reformatted. One of: <ul style="list-style-type: none"> • a function to be applied to <code>x</code>, e.g. format, toupper, abs, or any other custom function (including from the popular scales package). • one of the following convenience strings (or their symbol equivalents), for which common formatting transformations are provided: "percent" ("%"), "comma" (","), "log" ("l"), "dollar" ("\$"), "euro" ("€"), or "sterling" (£). • a dictionary, i.e. a <i>named</i> character vector or list of form <code>c(old_lab = "new_lab", ...)</code>. Entries of <code>x</code> that match a name are replaced by the corresponding value and the rest are left alone, so a partial mapping is fine. The lookup is by value rather than by position, which means it is unaffected by any reordering of the underlying categories. |

Note that an *unnamed* character vector of length greater than one is an error, rather than a positional replacement of the labels. Such a vector cannot be distinguished from a formatting keyword when *x* is of length one, and would not follow the categories if they were reordered. Pass a function if you want to compute the labels positionally, e.g. `function(x) LETTERS[seq_along(x)]`.

`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

[tinypplot](#)

Examples

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

# a named vector acts as a dictionary: matched entries are replaced and the
# rest are left alone
tinylabel(c("setosa", "versicolor"), c(setosa = "SET"))

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

# log example (combined with axis scaling)
tinypplot(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
tinypplot(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,
  xax1 = 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",
  yax1 = 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,  
xpad = NULL,  
yaxs = NULL,  
ypad = NULL,  
xaxb = NULL,  
yaxb = NULL,  
xaxl = NULL,  
yaxl = NULL,  
xaxr = NULL,  
yaxr = NULL,  
las = 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,
record = 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'
tinypplot(x = NULL, type = c("l", "area"), ...)

plt(x, ...)
```

Arguments

<code>x, y</code>	the <code>x</code> and <code>y</code> arguments provide the <code>x</code> and <code>y</code> 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 <code>xy.coords</code> for details. If supplied separately, <code>x</code> and <code>y</code> must be of the same length.
<code>...</code>	other graphical parameters. If <code>type</code> is a character specification (such as "hist") then any argument names that match those from the corresponding <code>type_*</code> () function (such as <code>type_hist</code>) are passed on to that. All remaining arguments from <code>...</code> can be further graphical parameters, see <code>par</code>).
<code>xmin, xmax, ymin, ymax</code>	minimum and maximum coordinates of relevant area or interval plot types. Only used when the <code>type</code> argument is one of "rect" or "segments" (where all four min-max coordinates are required), or "pointrange", "errorbar", or "ribbon" (where only <code>ymin</code> and <code>ymax</code> required alongside <code>x</code>). Can be passed as unquoted (NSE) variable names when called from the <code>tinypplot.formula</code> method, e.g. <code>ymin = varname</code> where <code>varname</code> is a variable in data.
<code>weights</code>	an optional numeric vector of observation weights, of the same length as the number of <code>x,y</code> coordinates. Can be passed as an unquoted (NSE) variable name when called from the <code>tinypplot.formula</code> method, e.g. <code>weights = varname</code> where <code>varname</code> is a variable in data. Weights are only consumed by the plot types that support them; specifically, the model fit and distribution types like <code>type_lm()</code> and <code>type_spineplot()</code> . 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).
<code>labels</code>	an optional vector of text labels for use with <code>type_text()</code> , of length 1 or the same length as the number of <code>x,y</code> coordinates. Can be passed as an unquoted (NSE) variable name when called from the <code>tinypplot.formula</code> method, e.g. <code>labels = varname</code> where <code>varname</code> is a variable in data. Ignored by other plot types. Overrides the <code>labels</code> argument of <code>type_text()</code> if both are supplied.
<code>by</code>	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., <code>pch</code> , <code>lty</code> , or <code>cex</code>) 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 <code>by</code> . To group by multiple variables, wrap them with <code>interaction</code> .
<code>facet</code>	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-sided 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.
- `drop` a logical value indicating whether facet levels that no observation uses should be dropped, rather than drawing an empty panel. Such levels arise from an unused factor level, or from a multi-variable facet like `~fvar1 + fvar2` whose cross-product covers combinations that the data never observe. Default is `FALSE`, i.e. the empty panels are drawn. For a two-sided (`fvar1 ~ fvar2`) grid, `drop = TRUE` cannot remove the panel outright, since the layout is a rectangle of rows by columns and doing so would misalign the panels that remain; instead the slot is kept but left blank, so that it reads as a gap rather than an empty box.
- `drop.levels` a logical value indicating whether each facet should keep only the categories of a categorical axis that it actually uses. Default is `FALSE`, i.e. every facet shows the full set of categories, even those with no data in that panel. With `drop.levels = TRUE` a panel's axis is recomputed as if its data had been passed through `factor()` on its own, so the surviving categories are re-spaced evenly and no gaps are left behind. Requires free scales (`free = TRUE`), since fixed panels share a single axis and per-panel categories would misalign them. Note that this is distinct from `drop` above: `drop` removes empty *facets*, whereas `drop.levels` removes unused *categories within* a facet.
- `axes` a character string for controlling which facets draw their own axes.

One of "all" (every facet gets its own axes), "outer" (only the facets along the bottom and left edges of the grid, so that redundant interior axes are dropped), or "none" (no facet axes at all). If left unspecified (the default), the behaviour is inherited from `frame.plot`: framed plots draw axes in every facet, since each frame visually contains its own axes, while frameless plots (e.g. `tinyptheme("clean2")`) draw only the outer ones. Use `axes = "outer"` to drop the interior axes while *keeping* the facet frames. Note that `axes` is ignored for free-scaled facets (`free = TRUE`), which necessarily require their own axes; the sole exception being `axes = "none"`, which is always respected.

- `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.
- `labeller` a formatting function or convenience string passed to `tinyplabel` for adjusting facet titles. This is applied to the underlying facet values, i.e. before any prefix (below) is added, and component-wise if the facets are defined over several variables. For the latter case, a (named) list or character vector can be used to format each variable differently, e.g. `labeller = list(firm = toupper, yield = "%")`, with any variable left unnamed not formatted. While not recommended, unnamed values are matched positionally, according to the variable order in the facet formula specification. Note that this per-variable naming claims the same slot that a `tinyplabel` dictionary would, so a dictionary has to be nested inside it, e.g. `labeller = list(Species = c(setosa = "SET"))`; a bare named vector is read as a per-variable mapping instead. Defaults to the value of `tpar("facet.labeller")`, which is `NULL` (no formatting).
- `prefix` a logical or character value for prefixing the facet titles with a descriptive name. Pass `TRUE` to prefix with the (deparsed) facet variable name(s), e.g. `"am = 0"` instead of just `"0"`. Alternatively, pass a custom prefix name as a character value, e.g. `prefix = "Automatic"` yields `"Automatic = 0"`. Like the `labeller` argument (above), a (named) character vector or list can be used to prefix multiple facet variables, e.g. `prefix = c(am = "Automatic", vs = "V-shaped")`, with any omitted variable taking its own (deparsed) name. While not recommended, unnamed values are matched positionally, according to the variable order in the facet formula specification. Similarly, a single string is recycled across all facet variables. (Regardless of how they are specified, note that prefixed multi-variable facet titles are separated by commas, rather than the usual colon.) Defaults to the value of `tpar("facet.prefix")`, which is `NULL` (equivalent to `FALSE`, i.e. no prefix).
- `sep` a character string separating the individual variables of a multi-variable facet title (ignored otherwise). For example, pass `sep = "\n"` to stack each variable on a separate line. Defaults to the value of `tpar("facet.sep")`, which is `NULL` and implies conditional behaviour depending on whether

the combined variables are prefixed (then: ", ") or the not (then: ":") for readability.

- 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`.
 - * "jitter" / `type_jitter()`: Jittered points.
 - * "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`.
 - * "rug" / `type_rug()`: Adds a rug to an existing plot.
 - * "segments" / `type_segments()`: Draws line segments between pairs of points.
 - * "text" / `type_text()`: Add text annotations.
 - * "tile" / `type_tile()`: Draws a grid of tiles, with the fill given by `by`.
 - 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.
- * "ellipse" / `type_ellipse()`: Draws confidence ellipse(s) around grouped points.
- * "heatmap" / `type_heatmap()`: Draws a grid of tiles, optionally rescaling the fill along one axis.
- * "hexbin" / `type_hexbin()`: Creates a hexagonal bin plot, a 2D analogue of a histogram.
- * "histogram" / `type_histogram()`: Creates a histogram of a single variable.
- * "qq" / `type_qq()`: Creates a quantile-quantile plot.
- * "ridge" / `type_ridge()`: Creates a ridgeline (aka joy) 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)`.

<code>main</code>	a main title for the plot, see also <code>title</code> .
<code>sub</code>	a subtitle for the plot.
<code>cap</code>	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 <code>sub</code> , since these two will otherwise overlap. Appearance can be customized via tpar parameters <code>adj.cap</code> , <code>cex.cap</code> , <code>col.cap</code> , <code>font.cap</code> , and <code>line.cap</code> .
<code>xlab</code>	a label for the x axis, defaults to a description of x.
<code>ylab</code>	a label for the y axis, defaults to a description of y.
<code>ann</code>	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
<code>xlim</code>	the x limits (<code>x1</code> , <code>x2</code>) of the plot. Note that <code>x1 > x2</code> 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., <code>xlim = c(0, 1)</code>), <code>tinyplot</code> supports three further convenience forms: • a single scalar (e.g. <code>xlim = 0</code>) 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. <code>xlim = c(0, NA)</code>) 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.
<code>ylim</code>	the y limits of the plot. Accepts the same input forms as <code>xlim</code> .

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.
xpad, ypad	numeric specifying how much padding, as a fraction of the data range, should be added to each end of the axes. If NULL (the default), follows an automatic padding heuristic that tries to optimize for plot aesthetic depending on the axis type: • numeric axis: inherited from x/yaxs. Under the default x/yaxs = "r" style, this means 0.04, i.e. 4% padding on each end (see par). • categorical axis: behaviour depends on the glyph-type and number of unique categories, and therefore tick marks. For example, point-like glyphs with numerous (8+) categories inherit the same padding logic as a numeric axis (i.e., usually 4%). For cases with fewer unique categories, the padding will instead correspond to 25% of the gap between interior tick marks, so that the end categories aren't drawn flush against the plot frame. Note that providing an explicit x/ylim or setting x/yaxs = "i" skips this heuristic in favour of the standard numeric behaviour.
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, character keyword, or dictionary (named vector or list) for formatting or (re)labelling the x- or y-axis tick labels. Passed to tinylabell ; see the latter's help file for more detailed documentation and examples. Note that this is a post-processing step that affects the <i>appearance</i> of the tick labels only; use in conjunction with x/yaxb if you would like to adjust the position of the tick marks too.
xaxr, yaxr	numeric giving the rotation of the x- or y-axis tick labels, in degrees counter-clockwise; NULL (the default) leaves them unrotated. Setting one overrides las for that axis alone, leaving the other axis under las as usual, and 0 (or any multiple of 360) counts as no rotation at all. Best paired with a dynamic theme (see tinytheme), since that is what resizes the margins to fit the tilted labels; under the default theme a long rotated label will be clipped unless you widen mar yourself. Defaults to the value of tpar("xaxr") / tpar("yaxr"), whose documentation covers this and the label-spacing caveat in more detail.
las	numeric in 0:3 giving the orientation of the axis tick labels, following the base par convention: 0 (parallel to the axis), 1 (always horizontal), 2 (perpendicular to the axis), or 3 (always vertical). NULL (the default) defers to the active theme, then to par("las"). Passing it here overrides both, but only for this plot; unlike

	<code>tpar(las=)</code> or <code>tinytheme(las=)</code> it does not persist. For rotations other than the four right angles, see <code>xaxr/yaxr</code> below.
<code>log</code>	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.
<code>flip</code>	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.
<code>frame.plot</code>	a logical indicating whether a box should be drawn around the plot. Can also use <code>frame</code> as an acceptable argument alias. The default is to draw a frame if both axis types (set via <code>axes</code> , <code>xaxt</code> , or <code>yaxt</code>) include axis lines.
<code>grid</code>	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 <code>grid()</code>. Note that this argument replaces the <code>panel.first</code> and <code>panel.last</code> arguments from <code>base plot()</code> 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 <code>tpar("grid")</code>.
<code>palette</code>	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 <code>palette</code> (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 <code>palette.pals()</code> or <code>hcl.pals()</code>. 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., <code>palette.colors</code>) or "closed" with a set of named arguments (e.g., <code>palette.colors(palette = "Okabe-Ito", alpha = 0.5)</code>). Note that any unnamed arguments will be ignored and the key <code>n</code> argument, denoting the number of colours, will automatically be spliced in as the number of groups. • A vector or list of colours, e.g. <code>c("darkorange", "purple", "cyan4")</code>. 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.
<code>pch</code>	plotting "character", i.e., symbol to use. Character, integer, or vector of length equal to the number of categories in the <code>by</code> variable. See <code>pch</code> . 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.

<code>lty</code>	line type. Character, integer, or vector of length equal to the number of categories in the <code>by</code> variable. See <code>lty</code> . In addition, users can supply a special <code>lty = "by"</code> 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., <code>par("lty")</code>) and recycle as necessary.
<code>lwd</code>	line width. Numeric scalar or vector of length equal to the number of categories in the <code>by</code> variable. See <code>lwd</code> . In addition, users can supply a special <code>lwd = "by"</code> 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., <code>par("lwd")</code>) and pad on either side of that.
<code>col</code>	plotting color. Character, integer, or vector of length equal to the number of categories in the <code>by</code> variable. See <code>col</code> . Note that the default behaviour in <code>tinypplot</code> is to vary group colors along any variables declared in the <code>by</code> 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 <code>"by"</code> keyword to one of <code>pch</code> , <code>lty</code> , <code>lwd</code> , or <code>bg</code> ; see below.)
<code>bg</code>	background fill color for the open plot symbols 21:25 (see <code>points.default</code>), 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 <code>col</code> : • <code>bg = "by"</code> will insert a background fill that inherits the main color mappings from <code>col</code>. • <code>bg = <numeric[0,1]></code> (i.e., a numeric in the range $[0, 1]$) will insert a background fill that inherits the main color mapping(s) from <code>col</code>, but with added alpha-transparency. For both of these convenience arguments, note that the (grouped) <code>bg</code> mappings will persist even if the (grouped) <code>col</code> 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.
<code>fill</code>	alias for <code>bg</code> . If non-NULL values for both <code>bg</code> and <code>fill</code> are provided, then the latter will be ignored in favour of the former.
<code>alpha</code>	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.
<code>cex</code>	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. <code>"p"</code>): • users can also supply a special <code>cex = "by"</code> convenience argument, in which case the character expansion will automatically adjust by group too. The

	range of this character expansion is controlled by the <code>clim</code> argument in the respective types; see <code>type_points()</code> for example. • passing a <code>cex</code> vector of equal length to the main <code>x</code> and <code>y</code> 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.
<code>add</code>	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 <code>tinypplot_add</code> , which provides a convenient wrapper around this functionality for layering on top of an existing plot without having to repeat arguments.
<code>draw</code>	a function that draws directly on the plot canvas (before <code>x</code> and <code>y</code> are plotted). The <code>draw</code> argument is primarily useful for adding common elements to each facet of a faceted plot, e.g. <code>abline</code> or <code>text</code> . Note that <i>no</i> internal checking is done for correctness; the provided argument is simply captured and evaluated as-is within <code>tinypplot()</code> and thus has access to the local definition of all variables such as <code>x</code> , <code>y</code> , etc. See Examples.
<code>empty</code>	logical indicating whether the interior plot region should be left empty. The default is FALSE. Setting to TRUE has a similar effect to invoking <code>type = "n"</code> above, except that any legend artifacts owing to a particular plot type (e.g., lines for <code>type = "l"</code> or squares for <code>type = "area"</code>) will still be drawn correctly alongside the empty plot. In contrast, <code>type = "n"</code> implicitly assumes a scatterplot and so any legend will only depict points.
<code>restore.par</code>	a logical value indicating whether the <code>par</code> settings prior to calling <code>tinypplot</code> 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 <code>tinypplot</code> legend. Users can opt out of this persistent behaviour by setting to TRUE instead. See also <code>get_saved_par</code> for another option to recover the original <code>par</code> settings, as well as longer discussion about the trade-offs involved.
<code>file</code>	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., <code>png</code> , <code>jpeg</code> , <code>pdf</code> , or <code>svg</code>). As a point of convenience, note that any global parameters held in <code>(t)par</code> 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 <code>".png"</code> , <code>".jpg"</code> (<code>".jpeg"</code>), <code>".pdf"</code> , or <code>".svg"</code> . (Other file types may be supported in the future.) The file dimensions can be controlled by the corresponding <code>width</code> and <code>height</code> arguments below, otherwise will fall back to the <code>"file.width"</code> and <code>"file.height"</code> values held in <code>tpar</code> (i.e., both defaulting to 7 inches, and where the default resolution for bitmap files is also specified as 300 DPI).
<code>width</code>	numeric giving the plot width in inches. Together with <code>height</code> , typically used in conjunction with the <code>file</code> argument above, overriding the default values held in <code>tpar("file.width", "file.height")</code> . If either <code>width</code> or <code>height</code> is specified, but a corresponding <code>file</code> 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 <code>tpar("file.height")</code> if not specified.
record	(experimental) a logical value indicating whether the plot should be recorded and returned as a replayable <code>recordedtinypplot</code> object. Defaults to FALSE. Setting to TRUE allows for assignment and later recall, e.g. <code>myplot = tinypplot(...); myplot</code> . This behaviour can also be set globally via <code>tpar(record = TRUE)</code> ; an explicit argument here takes precedence. Note that recording requires a device with an enabled display list (see <code>dev.control</code>). Most interactive devices enable this behaviour by default, whereas file-based devices do not. However <code>tinypplot</code> automatically enables it for any device that it opens itself via file, and further emits a warning if the current device is not recording.
asp	the y/xy/x aspect ratio, see <code>plot.window</code> .
theme	keyword string (e.g. "clean") or list defining a theme. Passed on to <code>tinytheme</code> , but reset upon exit so that the theme effect is only temporary. Useful for invoking ephemeral themes.
formula	a <code>formula</code> that optionally includes grouping variable(s) after a vertical bar, e.g. <code>y ~ x z</code> . One-sided formulae are also permitted, e.g. <code>~ y z</code> . Only a single y and x variable (if any) must be specified but multiple grouping variables can be included in different ways, e.g. <code>y ~ x z1 : z2</code> or <code>y ~ x z1 + z2</code> . (These two representations are treated as equivalent; both are parsed as <code>interaction(z1, z2)</code> internally.) If arithmetic operators are used for transforming variables, they should be wrapped in <code>I()</code> , e.g., <code>I(y1/y2) ~ x</code> . Note that the formula and x arguments should not be specified in the same call.
subset, na.action, drop.unused.levels	arguments passed to <code>model.frame</code> when extracting the data from formula and data.

Details

Disregarding the enhancements that it supports, `tinypplot` 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 `tinypplot` (or its shorthand alias `plt`), without causing unexpected changes to the output.

Value

By default, no return value; called for the side effect of producing a plot. If `record = TRUE` (or globally via `tpar(record = TRUE)`), the plot is instead returned invisibly as a "recordedtinypplot" object, which can be replayed later; see `recordedtinypplot`.

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.
tinypplot(
  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).
tinypplot(
  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.

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

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

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

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

tinypplot(

```

```

    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"
)

# Customize facets by passing a list of options to the companion `facet.args`
# argument. For example, here we arrange the facets in a single row and also
# turn off the inner axes labels to reduce redundancy.

tinyplot(
  Temp ~ Day | Summer,
  facet = ~Month,
  facet.args = list(nrow = 1, axes = "outer"),
  data = aq,
  type = "area",
  palette = "dark2",
  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$hot = aq$Temp >= 75
aq$windy = aq$Wind >= 15
tinyplot(
  Temp ~ Day,
  facet = windy ~ hot,
  facet.args = list(prefix = TRUE), # optional
  data = aq
)

# (Note: The optional `prefix = TRUE` argument prepends the facet variable
# names to the strip titles, making for a more informative display here.)

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

tinyplot(
  Temp ~ Day,
  facet = windy ~ hot,

```

```

    facet.args = list(prefix = TRUE),
    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 `tinypplot` to place the legend _outside_ the plot
# area.

tinypplot(
  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.

tinypplot(
  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:
tinypplot(
  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

tinypplot(
  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.array

tinypLOT Method for Plotting Arrays

Description

Convenience interface for visualizing `array` objects with tinypLOT. Extends the `tinypLOT.matrix` conventions to arrays with up to four dimensions, by mapping the higher dimensions to facets.

Usage

```

## S3 method for class 'array'
tinypLOT(
  x,
  type = NULL,
  legend = NULL,
  facet = NULL,
  xlab = NULL,
  ylab = NULL,
  xy = NULL,
  ...
)

```

Arguments

`x` an object of class "array".

`type` plot type passed on to tinypLOT. Defaults to "p" (points).

legend	specification passed on to <code>tinypplot</code> . The default is to draw a legend when the matrix has named columns, and to suppress it otherwise. For "tile" and "heatmap" types it is suppressed by default.
facet	must be NULL (the default) or FALSE for arrays with three or more dimensions, since the facets are then determined by the array dimensions. FALSE draws everything in a single panel instead, by folding the facet dimensions into the by groups. For x/y pairs, these groups then replace the first dimension as by, so that each path is drawn separately. Not supported for tile and heatmap types.
xlab, ylab	axis labels passed on to <code>tinypplot</code> . 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. For "tile" and "heatmap" types both titles default to NA, since the dimnames label both axes.
xy	which dimension (if any) holds x/y pairs; see Details. The default NULL auto-detects it, FALSE opts out, and a dimension index or name selects it explicitly. The selected dimension must have length 2, and its dimnames (if any) are used as the axis titles.
...	further arguments passed to <code>tinypplot</code> .

Details

The first two dimensions of the array are treated exactly like the rows and columns of a matrix; see [tinypplot.matrix](#). By default, that means the values are plotted against the (first dimension) row index, with a separate by group for each element of the second dimension. Higher dimensions are then mapped to facets:

- 1D arrays are treated as a single-column matrix, i.e. a simple index plot. If a y variable is also supplied, e.g. `tinypplot(tapply(...), y)`, then the array is instead treated as a plain x vector.
- 2D arrays are matrices, and hence dispatch to [tinypplot.matrix](#).
- 3D arrays are faceted by the third dimension (i.e., a "facet wrap").
- 4D arrays are faceted by the third and fourth dimensions, as the rows and columns of a "facet grid", respectively.
- Arrays with more than four dimensions are not supported. Subset or reshape them first.

Dimension names are used to label the groups, facets, and axis ticks, while the names of the dimnames (if any) are used as the axis, legend, and facet titles. To assign the dimensions to different roles, rearrange them first with [aperm](#).

x/y pairs. Arrays often hold a stack of matrices, one per variable, e.g. the hip and knee angles of the gait data (Time x Subject x Variable). If an array of three or more dimensions has exactly one length-2 dimension besides the first, then its two slices are plotted against each other, as x and y. Each of the remaining dimensions then shifts down a role:

- The first dimension becomes by, and orders the points along each path. It is kept numeric where possible (its dimnames, e.g. times, or else its index), so that each path is drawn as a line with a colour gradient. If the dimnames are not numeric, then the groups are discrete and drawn as points instead.

- The second dimension becomes a facet wrap (e.g. one panel per subject).
- The third dimension (if any) makes this a facet grid, as its columns.

Control this via the `xy` argument. Tile and heatmap types are exempt, since they need the array values as their fill.

Note that contingency tables created by `table` (e.g., `HairEyeColor` or `Titanic`) are of class `"table"` and hence do not dispatch to this method.

Value

By default, no return value; called for the side effect of producing a plot. If `record = TRUE` (or globally via `tpar(record = TRUE)`), the plot is instead returned invisibly as a `"recordedtinyplot"` object, which can be replayed later; see `recordedtinyplot`.

See Also

`tinyplot.matrix`, `matplot`

Examples

```
# 3D array: facet wrap by the third dimension
sims = array(
  cumsum(rnorm(20 * 3 * 4)), dim = c(20, 3, 4),
  dimnames = list(NULL, paste("Series", 1:3), paste("Run", 1:4))
)
tinyplot(sims, type = "l")

# 4D array: facet grid by the third and fourth dimensions
sims4 = array(
  rnorm(20 * 3 * 2 * 2), dim = c(20, 3, 2, 2),
  dimnames = list(
    time = NULL, series = paste0("s", 1:3),
    model = c("A", "B"), scenario = c("low", "high")
  )
)
tinyplot(sims4, type = "b", facet.args = list(prefix = TRUE))

# tile/heatmap types lay out each 2D slice as a grid
tinyplot(sims4[1:5, , , ], type = "heatmap", theme = "heatmap")

# x/y pairs: a length-2 dimension is plotted as x vs y
# (e.g., hip vs knee angle through the gait cycle, coloured by time and
# faceted by boy)
gait9 = gait[, 1:9, ] # take a subset for demonstration
tinyplot(gait9)

# all boys in a single panel instead
tinyplot(gait9, facet = FALSE)

# ... or in the background of each boy's own panel
tinyplot(
```

```

    gait9,
    draw = tinyplot(gait9, col = "lightgray", facet = FALSE, add = TRUE)
)

# opt out of x/y pairs, to plot the angles against time instead
tinyplot(gait9, type = "l", xy = FALSE, legend = FALSE)

```

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

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

Details

This is a convenience function for plotting data frames with or without a formula. The case with the formula mainly facilitates using `tinypplot()` 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 `tinypplot` variants (deliberately) produce a histogram for the univariate cases along the grid diagonal.

Value

By default, no return value; called for the side effect of producing a plot. If `record = TRUE` (or globally via `tpar(record = TRUE)`), the plot is instead returned invisibly as a "recordedtinypplot" object; see [recordedtinypplot](#). The exception is the `pairs`-style case (more than two columns), which draws a grid of sub-plots and always returns `NULL`.

Examples

```
## using tinypplot() with data frames
tinypplot(cars)
tinypplot(iris, Sepal.Length ~ Petal.Width | Species)
tinypplot(
  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 |> tinypplot()
# iris |> tinypplot(Sepal.Length ~ Petal.Width | Species)
# UCBAdmissions |>
#   as.data.frame() |>
#   tinypplot(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")
tinypplot(iris)

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

## another option (but assumes objects of equal length)
tinypplot(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

<code>x</code>	an object of class "matrix".
<code>type</code>	plot type passed on to tinypLOT. Defaults to "p" (points).
<code>legend</code>	specification passed on to tinypLOT. The default is to draw a legend when the matrix has named columns, and to suppress it otherwise. For "tile" and "heatmap" types it is suppressed by default.
<code>facet</code>	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.
<code>xlab, ylab</code>	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. For "tile" and "heatmap" types both titles default to NA, since the dimnames label both axes.
<code>...</code>	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.

The "tile" and "heatmap" types are an exception, since the matplot convention makes little sense for them. Instead the matrix is laid out as a grid—columns along the x-axis, rows along the y-axis—with the matrix *values* supplied as the fill. The y-axis is reversed so that the first row sits at the top, matching how one reads a matrix (cf. [heatmap](#) and [image](#)); pass an explicit ylim to override. Both axis titles are suppressed, since the dimnames already label the ticks, and so is the legend, since the fill merely re-encodes the matrix's own values. Pass an explicit legend (or xlab/ylab) to override either. See Examples.

Value

By default, no return value; called for the side effect of producing a plot. If record = TRUE (or globally via tpar(record = TRUE)), the plot is instead returned invisibly as a "recordedtinyplot" object, which can be replayed later; see [recordedtinyplot](#).

See Also

[tinyplot.array](#), [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")

# digression: equivalent "o" 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)))

# back to VADeaths running example, we can pass down other types too...

# heatmap
tinyplot(VADeaths, type = "heatmap", theme = "heatmap", col = "white")

# barplot(s)
tinyplot(VADeaths, type = "barplot", beside = TRUE)
tinyplot(t(VADeaths), type = "barplot", beside = TRUE)
tinyplot(VADeaths, type = "barplot", facet = "by", legend = FALSE)
```

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'
tinypplot(
  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 tinypplot.formula. The default in the tinypplot 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 tinypplot.

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 tinypplot(Nile, type = "histogram").

Value

By default, no return value; called for the side effect of producing a plot. If record = TRUE (or globally via tpar(record = TRUE)), the plot is instead returned invisibly as a "recordedtinypplot" object, which can be replayed later; see [recordedtinypplot](#).

Examples

```
## univariate series
tinypplot(Nile)

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

## multivariate series (generally, these also look better with a theme)
```

```

tinytheme("clean2")
tinypplot(EuStockMarkets)          ## faceted, free scales, same color
tinypplot(EuStockMarkets, facet.args = NULL) ## faceted, same scale, same color
tinypplot(EuStockMarkets, facet = "by")    ## faceted, free scales, diff colors
tinypplot(EuStockMarkets, facet = NULL)    ## single frame, diff colors

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

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

tinytheme() ## reset

```

tinypplot_add

Add new elements to the current tinypplot

Description

This convenience function grabs the preceding tinypplot 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 `tinypplot_add()`.

Usage

```
tinypplot_add(...)
```

```
plt_add(...)
```

Arguments

```
...
```

All named arguments override arguments from the previous calls. Arguments not supplied to [tinypplot_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 [tinypplot](#) call.

Value

By default, no return value; called for the side effect of producing a plot. If `record = TRUE` (or globally via `tpar(record = TRUE)`), the plot is instead returned invisibly as a "recordedtinypplot" object, which can be replayed later; see [recordedtinypplot](#).

Adding versus drawing

`tinypplot_add()` layers new elements *on top* of the existing plot. The complementary, top-level `tinypplot(..., draw = <>)` argument works in the opposite direction: it layers *underneath* the main plot elements.

However, note that `draw` is fully generic—it accepts any drawing or annotation expression—and this includes `tinypplot_add()` itself. Passing `draw = tinypplot_add(...)` thus yields a regular `tinypplot` layer, inheriting from the enclosing call in the usual way, but drawn below the main elements rather than above them. See Examples.

Limitations

- `tinypplot_add()` works reliably only when adding to a plot originally created using the `tinypplot.formula` method with a valid data argument. We cannot guarantee correct behavior if the original plot was created with the atomic `tinypplot.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

```
#
## Basic use

tinypplot(Sepal.Width ~ Sepal.Length | Species,
  facet = ~Species,
  data = iris)
tinypplot_add(type = "lm") ## or : plt_add(type = "lm")

## the previous line is equivalent to (but much more convenient than)
## re-writing the full call with the new type and `add=TRUE`:
# tinypplot(Sepal.Width ~ Sepal.Length | Species,
#   facet = ~Species,
#   data = iris,
#   type = "lm",
#   add = TRUE)

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

#
## add(ing) vs draw(ing)

dat = data.frame(x = 1:3, y = 0)

## the `draw` argument layers *underneath* the main plot elements...
tinypplot(
  y ~ x, data = dat, pch = 19, cex = 10,
  draw = abline(h = 0, lwd = 4, col = "hotpink")
)
```

```
## ... whereas `tinyplot_add()` layers *on top*.
tinyplot(y ~ x, data = dat, pch = 19, cex = 10)
tinyplot_add(type = type_hline(0), lwd = 4, col = "hotpink")

## combine = best of both worlds? Since `draw` is generic, we can hand it
## `tinyplot_add()` directly. The line is drawn as a regular tinyplot layer,
## but underneath once more. This is especially useful for, say. flipped
## plots since correct axes inheritance is preserved. (Note that a plain
## `abline(h = 0)` is not flip-aware)
tinyplot(
  y ~ x, data = dat, pch = 19, cex = 10, flip = TRUE,
  draw = tinyplot_add(type = type_hline(0), lwd = 4, col = "hotpink")
)
```

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", "heatmap", "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. – "heatmap" (*): a specialized theme for tile plots and heatmaps (see type_tile()). Builds off of "clean2", but removes the axis padding so that the tiles meet the panel edge, drops the (redundant) grid lines, rotates the tick labels and removes their tick marks, and defaults to the "tealgrn" sequential palette. Not recommended for non-tile plots. – "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 <code>tpar()</code> 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 <code>tinytheme()</code> 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)
tinypplot(..., 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 `tc1` 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\_get\(\)`](#) for retrieving the name of the active theme; [`tinytheme\_list\(\)`](#) for listing the names of all available themes; [`tinytheme\_register\(\)`](#) for registering custom named themes.

Examples

```

# Reusable plot function
p = function() tinypplot(
  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_get*Get the currently active theme*

Description

Returns the name of the [tinytheme](#) that is currently in effect. Handy for saving a theme and restoring it later, or for querying the active theme programmatically.

Usage

```
tinytheme_get()
```

Details

A thin convenience wrapper around the "tinytheme" entry of [tpar](#). The difference is that it always returns a plain character string, including in a session where no theme has been set yet (where `tpar("tinytheme")` returns NULL rather than "default"). Use [tpar](#) if you want the full set of underlying theme settings, rather than just the name.

Note that a theme passed to the `tinypplot(..., theme =)` argument is ephemeral: it is reset on exit, so it is only visible to `tinytheme_get()` from inside that same call (e.g. via `draw`).

Value

A character string naming the active theme.

See Also

[tinytheme](#), [tinytheme_register](#), [tpar](#)

Examples

```
# no theme set yet
tinytheme_get()

# save the current theme, switch, then restore it afterwards
otheme = tinytheme_get()
tinytheme("classic")
tinytheme_get()

tinypplot(mpg ~ wt, data = mtcars)

tinytheme(otheme) # back to where we started
```

```
tinytheme_get()
```

tinytheme_register	<i>Register, List, and Unregister Custom Themes</i>
--------------------	---

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

<code>name</code>	Character string. The name for your custom theme. Cannot clash with or overwrite a built-in theme name ("default", "clean", etc.)
<code>theme</code>	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".
<code>...</code>	Named arguments to override specific theme settings. These are the same parameters accepted by <code>tpar()</code> .

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\(\)](#), [tinytheme_get\(\)](#)

Examples

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

# Use it
tinypplot(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 difference is that, beyond supporting the standard group of R graphical parameters in [par](#), [tpar](#) also supports additional graphical parameters that are provided by [tinypplot](#). 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 tinypplot -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 <code>?setHook</code>).

Details

The [tinypplot](#)-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 `tinypplot_*` prefix, e.g. `options(tinypplot_grid = TRUE, tinypplot_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.
- `cex.xaxs`, `cex.yaxs`: Numeric expansion factors for the x- and y-axis tick labels, respectively. Both default to NULL, whereby the shared `cex.axis` value is used instead (which in turn falls back to `par("cex.axis")`). Set one of them to size a single axis' tick labels independently of the other, e.g. `cex.yaxs` to shrink a long list of category names on the y-axis without also shrinking the x-axis. Compare `cex.xlab` and `cex.ylab`, which do the same for the axis *titles*.
- `col.cap`: Character specifying the colour of the plot caption. Defaults to "black".
- `col.xaxs`, `col.yaxs`: Characters (or integers) specifying the colour of the x- and y-axis tick labels, respectively. Both default to NULL, whereby the shared `col.axis` value is used instead.
- `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 `tinypplot` 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.axes`: Character string controlling which facets draw their own axes: "all" (each facet), "outer" (only facets on the outer edge of the facet grid, dropping redundant interior axes), or "none". Defaults to NULL, whereby the choice is inferred from whether the plot is framed (see the `axes` argument of `facet.args`). Equivalent to setting `tinypplot(..., facet.args = list(axes = X))`, but globally, which also makes it available to themes.
- `facet.drop`: Logical indicating whether facet levels that no observation uses should be dropped, rather than drawing an empty panel. Defaults to FALSE. Equivalent to setting `tinypplot(..., facet.args = list(drop = X))`, but globally, which also makes it available to themes.
- `facet.drop.levels`: Logical indicating whether each free facet (`facet.args = list(free = TRUE)`) should keep only the categories of a categorical axis that it actually uses, re-levelled as if that panel's data had been passed through `factor()` on its own. Defaults to FALSE, i.e. every panel shows the full set of categories. Equivalent to setting `tinypplot(..., facet.args = list(drop.levels = X))`, but globally, which also makes it available to themes.
- `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).
- `facet.labeller`: A formatting function (or `tinylabeller` convenience string, e.g. "percent") applied to the facet titles, or a list of them (or a character vector of convenience strings), optionally named for the facet variables they apply to, for formatting each facet variable differently. Note that this per-variable naming claims the same slot that a `tinylabeller` dictionary would, so a dictionary has to be nested inside it, e.g. `facet.labeller = list(Species = c(setosa = "SET"))`. Defaults to NULL (no formatting). Applied to the underlying facet values, i.e. before any `facet.prefix` name is added. Equivalent to setting `tinypplot(..., facet.args = list(labeller = X))`, but globally.
- `facet.prefix`: Logical or character controlling whether facet titles are prefixed with their variable name, e.g. "vs = 0" rather than just "0". TRUE uses the variable name(s), while

a character string—or a vector or list of them, one element per facet variable, optionally named for the variables they apply to—supplies custom name(s) instead. Defaults to NULL, which is equivalent to FALSE (no prefix). Equivalent to setting `tinypplot(..., facet.args = list(prefix = X))`, but globally.

- `facet.sep`: Character string separating the individual variables of a multi-variable facet title, e.g. `"\n"` to stack them on separate lines. Ignored for single-variable facets. Defaults to NULL, i.e. the ":" that the variables were combined with, or ", " if they are prefixed via `facet.prefix` (above). Equivalent to setting `tinypplot(..., facet.args = list(sep = X))`, but globally.
- `file.height`: Numeric specifying the height (in inches) of any plot that is written to disk using the `tinypplot(..., 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 `tinypplot(..., file = X)` argument. Defaults to 300.
- `file.width`: Numeric specifying the width (in inches) of any plot that is written to disk using the `tinypplot(..., 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 `tinypplot` 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.
- `lty.xaxs`, `lty.yaxs`: Line types for the x- and y-axis lines, respectively. Both default to NULL, whereby the shared `lty.axis` value is used instead.
- `lwd.xaxs`, `lwd.yaxs`: Line widths for the x- and y-axis lines, respectively. Both default to NULL, whereby the shared `lwd.axis` value is used instead.

- `palette.qualitative`: Palette for qualitative colors. See the `palette` argument in `?tinypplot`.
- `palette.sequential`: Palette for sequential colors. See the `palette` argument in `?tinypplot`.
- `record`: (experimental) Logical indicating whether `tinypplot()` should record plots and return them as replayable `recordedtinypplot` objects. Defaults to `NULL`, which is equivalent to `FALSE`. Setting to `TRUE` allows for assignment and later recall, e.g. `myplot = tinypplot(...); myplot`. Sets the default for the `record` argument of `tinypplot()`, which takes precedence. Note that recording requires a device with an enabled display list (see `dev.control`). Most interactive devices enable this behaviour by default, whereas file-based devices do not. However `tinypplot()` automatically enables it for any device that it opens itself via file, and further emits a warning if the current device is not recording.
- `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`.
- `xpad`, `ypad`: Numeric specifying how much padding, as a fraction of the data range, should be added to each end of the x- and y-axis, respectively. Both default to `NULL`, in which case an automatic heuristic applies: a numeric axis inherits from `x/yaxis` (usually 4% padding on each end), whereas a categorical axis with point-like glyphs and few unique categories instead gets 25% of the gap between interior tick marks. Sets the default for the `xpad` and `ypad` arguments of `tinypplot()`, which take precedence and where the heuristic is described in full.
- `xaxr`, `yaxr`: Numeric giving the rotation of the x- and y-axis tick labels, in degrees counter-clockwise; `NULL` (the default) leaves them unrotated. Unlike `las`, which is limited to the four right angles, any angle is permitted. Setting one overrides `las` for that axis alone, leaving the other axis under `las` as usual, and `0` (or any multiple of 360) counts as no rotation at all. Sets the default for the `xaxr` and `yaxr` arguments of `tinypplot()`, which take precedence. Two caveats follow from `tinypplot` drawing rotated labels itself rather than deferring to `base axis()`. First, margins are only resized to fit them under a theme with `dynmar = TRUE` (see `tinytheme`); under the default theme the margins are left alone, so a long rotated label will be clipped unless you widen `mar` yourself. Second, rotated labels do not inherit the thinning that `axis()` applies via `gap.axis`, so they start to overlap once the spacing between ticks falls below `line height / sin(srt)`.

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 `tinypplot` 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. `tinytheme_get` returns the name of the currently active theme.

Examples

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

# Simple facet plot with these default values
tinypplot(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 `tinypplot_add`.

Line parameters always refer to the original (pre-flip) variables. So, if `flip = TRUE`, then `type_hline()` draws vertical line(s) at the given y value(s), `type_vline()` draws horizontal line(s) at the given x value(s), and `type_abline()` draws the line $y = a + bx$ with the axes swapped.

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, `tinypplot(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: `tinypplot(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: `tinypplot(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

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

# customize by passing bespoke intercept and slope values
tinypplot_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
tinypplot(x = -10:10, y = -10:10, grid = TRUE, type = "abline")
```

```

#
## hline and vline

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

# Add horizontal lines at the (default) 0 y-intercept
tinypplot_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)
tinypplot_add(type = type_hline(with(mtcars, tapply(mpg, cyl, mean))), lty = 2)

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

#
## Flipped plots

# Line parameters refer to the original x and y variables, so "hline" is
# drawn vertically and "vline" horizontally when flip = TRUE
tinypplot(mpg ~ wt, data = mtcars, flip = TRUE)
tinypplot_add(type = type_hline(20), col = "hotpink")
tinypplot_add(type = type_vline(3), col = "dodgerblue")
tinypplot_add(type = type_abline(a = 37, b = -5), lty = 2)

#
## Recycling logic

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

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

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

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

```

Description

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

Usage

```
type_area(alpha = NULL, stack = FALSE, byord = NULL, fun = 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 <code>tpar("ribbon.alpha")</code> (i.e., probably 0.2 unless this has been overridden by the user in their global settings.) |
| stack | logical. Should the by groups be stacked on top of one another, rather than overplotted from a common zero baseline? Only relevant for grouped area plots. Default is FALSE. See the "Stacked area plots" section below. |
| byord | <p>keyword string or function. Permits on-the-fly (re)ordering of the by group layers, thus controlling the order in which they stack. Options are:</p> <ul style="list-style-type: none"> • "desc" and "asc" rank the groups by their summed y across the x axis, largest or smallest first. With "desc" the biggest group is stacked first, as the bottom layer. (Long forms like "descending" and "increasing" are also accepted.) • "start" and "end" instead rank on the y values at the smallest and largest x respectively, again stacking the largest group first. • "minvar" ranks by variance, putting the lowest variance group on the baseline. • "asis" or "rev" permute the existing levels without consulting the data at all. The former takes the groups in the order that they appear in the data, while the latter reverses the current level order. • a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so <code>function(y) -median(y)</code> layers by median y value, from the biggest to the smallest. Note that if a function requires access to a group's x values, then one of its arguments <i>must</i> be named x, e.g. <code>function(y, x) coef(lm(y ~ x))[2]</code> would layer by trend. <p>Defaults to NULL, i.e. keep the existing factor levels. To set that order explicitly, call <code>factor(levels = ...)</code> on the grouping variable beforehand. See Examples, as well as the "Stacked area plots" section below.</p> |
| fun | a function for collapsing repeated y values within a group and x position, used only when <code>stack = TRUE</code> . Defaults to mean, matching <code>type_barplot()</code> , so that the same data stacks to the same heights whether it is drawn as bars or as an area. |

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. <code>dodge = 0.1</code> places the outermost groups one-tenth of the way to adjacent breaks, <code>dodge = 0.5</code> 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 <i>strongly</i> 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.

Stacked area plots

Passing `type_area(stack = TRUE)` stacks the by groups cumulatively, rather than drawing each one from a zero baseline. Groups are accumulated in the order of their (factor) levels, so the first level forms the bottom band and the top of the final band traces the group total. Stacking is computed separately within each facet.

The `byord` argument is a helpful companion to stacked area plots, since it enables on-the-fly adjustment of the stacking order. For example, the size keywords—`"desc"`, `"start"`, and `"end"`—rank the stacked by groups according to their y values, either summed across the x axis or taken at one end of it. Following convention, the ranking runs in descending order, so that the biggest group is drawn on the bottom layer (use `"asc"` for the reverse). However, size is not the only route to a stable baseline. Because each band is drawn on top of the ones below it, they all inherit whatever movement the bottom layer has. A large but volatile group can therefore be a worse choice of foundation than a small, steady one. In this latter case, the `"minvar"` keyword would be a more appropriate choice since it places the lowest variance group at the bottom. Your choice of stacking ordering should therefore respond to the patterns in your data and which layers you want to emphasize.

Stacking needs exactly one y value per group per x value. Repeated cells—typically caused by a variable that is present in the data but absent from the plot—are collapsed with `fun` (default mean) rather than being stacked against each other. Conversely, groups that are *missing* an x value (or have an NA there) count as contributing zero at that point, so that a gap in one group does not shift the groups stacked above it. Note that stacking negative values is not meaningful and will produce overlapping bands. Note that the legend key for stacked area plots is deliberately inverted compared to other plot types (including non-stacked area plots) to ensure a consistent ordering

with the "bottoms-up" layering of the stacked regions. Similarly, reordering of the by group levels will reassign the palette, since group colours are allocated by level position. This matches what releveling a factor does elsewhere, but it does mean that reordering the bands repaints them.

Finally, note that unlike non-stacked area plots, the stacked bands are drawn with opaque fill by default, since they do not overlap. Pass an explicit alpha or fill value to override.

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")

#
## Stacked area plots

# Grouped area plots can be stacked cumulatively, rather than being drawn
# from a common zero baseline.

# Group B is small and steady; A and C are larger and wobblier.
dat = expand.grid(year = 2000:2020, grp = factor(c("A", "B", "C")))
dat$val = as.integer(dat$grp) +
  c(1.2, 0.1, 1.8)[dat$grp] * sin(dat$year / 3) +
  c(0.06, 0.02, 0.10)[dat$grp] * (dat$year - 2000)

tinyplot(val ~ year | grp, data = dat, type = type_area(stack = TRUE))

# Use `byord` to control which group stacks where. Here we stack by their
# largest end value.

tinyplot(
```

```

    val ~ year | grp, data = dat,
    type = type_area(stack = TRUE, byord = "end")
  )

  # `"minvar"` instead puts the *least variable* group on the baseline. Every
  # band inherits the movement of the ones below it, so a steady bottom layer
  # keeps the whole chart legible. Here that picks group B, which the default
  # level order leaves in the middle and `"end"/`"desc"` push to the top.

  tinyplot(
    val ~ year | grp, data = dat,
    type = type_area(stack = TRUE, byord = "minvar")
  )

  # `"rev"` simply flips the existing level order, which is the one thing a
  # ranking function cannot do (it never sees which group it was handed).

  tinyplot(
    val ~ year | grp, data = dat,
    type = type_area(stack = TRUE, byord = "rev")
  )

  # Custom ranking functions are also accepted. Name an argument `x` and it
  # receives the group's x values too, which is what a slope needs.

  tinyplot(
    val ~ year | grp, data = dat,
    type = type_area(stack = TRUE, byord = function(y, x) coef(lm(y ~ x))[2])
  )

  # Stacking expects a single `y` value per group per `x` value. Any repeats
  # are collapsed for us first, using `fun` (`mean` by default). Here, for
  # instance, ChickWeight records many chicks per diet at each timepoint.

  tinyplot(
    weight ~ Time | Diet, data = ChickWeight,
    type = type_area(stack = TRUE, fun = median)
  )

  # (Illustrative purposes aside, we leave it to the reader to decide whether
  # stacking separate diets on top of one another makes any sense...)

  #
  ## 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 = "Overlapping 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 $\sim x$ (without left-hand side) the barplot visualizes the counts (absolute frequencies) of the levels of x . For formulas of type $y \sim x$ 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,
  xord = NULL,
  drop.zeros = FALSE,
  na.as.zero = NULL,
  lighten = TRUE,
  xaxlabels = NULL,
  FUN = NULL
)
```

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. • <i>Positions</i> via an unnamed numeric scalar or vector. Bars start at the offset value(s) rather than zero, matched per x-level after any xlevels or xord reordering (a scalar is applied to all bars). Useful for waterfall charts. The positional form cannot be combined with center. • <i>Category</i> 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, FUN	a function to compute the summary statistic for y within each group of x in case of using a two-sided formula $y \sim x$ (default: mean). FUN is a (backwards-compatible) alias for fun. Supply only one or the other; fun takes precedence if both are provided.
xlevels, xord	arguments controlling the order of the x variable, and hence of the x-axis. Supply one or the other; if both arguments are provided, xlevels takes precedence and xord is silently ignored. • xlevels specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., c("C", "B", "A")), or a numeric vector of the corresponding level indexes (e.g. 3:1). • xord instead accepts a keyword or custom function, which then <i>derives</i> the order from the data. Options are: – "desc(ending)" and "asc(ending)" rank (sort) the categories by bar height, tallest or shortest first. Both the abbreviated and long form strings are permitted, as are the "decreasing" and "increasing" aliases. Note that the ranking is applied to the <i>aggregated</i> bars, i.e. whatever fun produced, rather than the underlying rows. With by groups or facets, a single ordering is computed and shared across all of them, by summing each category's bars over every group and facet. For stacked bars that sum is the height of the full stack; with beside = TRUE it is the group total rather than any individual bar. (A factor carries one level order, so a per-facet ranking is not expressible.)

- "asis" or "rev" permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order.
- a custom function that determines both the ranking statistic and its direction. The statistic is always sorted in ascending order, so `function(y) -median(y)` ranks by median, largest first.

Note that a numeric `x` is coerced to a factor before the bars are drawn, so it is reordered like any other categorical variable. Each argument defaults to `NULL`, i.e. keep the existing factor levels.

<code>drop.zeros</code>	logical. Should bars with zero height be dropped? If set to <code>FALSE</code> (default) a zero height bar is still drawn for which the border lines will still be visible.
<code>na.as.zero</code>	logical. Should a category that no observation reaches be treated as a zero? Defaults to <code>NULL</code> , i.e. let fun decide; see the "Implicit zeros and empty cells" section below. Set to <code>TRUE</code> to always mark such categories with a zero-height bar, or <code>FALSE</code> to never draw them.
<code>lighten</code>	logical. Should the fills use a lighter, opaque tint of the series colour(s)? Default is <code>TRUE</code> , which keeps single- and multi-group displays consistent and lets the fill read cleanly over grid lines. Set to <code>FALSE</code> to use the fully-saturated palette colour(s) instead.
<code>xaxlabels</code>	[Deprecated] a character vector with the axis labels for the <code>x</code> variable. Use the top-level <code>xax1</code> argument instead (see tinylabel). This argument will be removed in a future release.

Implicit zeros and empty cells

Grouping (by) or faceting a barplot asks for a bar per unique combination (i.e., category + group + facet). However, some combinations may not be supported by any observations in the data. In these cases, the bars are still computed off the completed combination set; particularly since stacked and centered barplots have to line up regardless. Yet this in turn invites the question of whether an "empty" cell should be read as an implicit zero, or treated as missing (NA).

In most cases, the question is answered by the fun aggregation. This is because it computes the actual value that is being plotted. For example, a *count* of no observations is 0, whereas a *mean* of no observations is undefined and thus better treated as NA (similarly for other summary statistics like the variance or median). Nonetheless, users can override this behaviour either way with the `na.as.zero` argument.

Three related points are worth highlighting. First, a zero-height bar only reads as zero when it is measured *from* zero. So offset layouts (waterfall, diverging/Likert) never draw one unless `na.as.zero = TRUE` asks for it explicitly. Second, a "zero" combination is still subject to the `drop.zeros` argument like any other zero. So `na.as.zero = TRUE`, `drop.zeros = TRUE` cancel out. Finally, `na.as.zero = TRUE` marks only the categories that *could* have held data. Where `x`, `by` or `facet` are mapped to the same variable, for example, most combinations cannot occur at all—and are never drawn, regardless.

Examples

```
#
## Basic use (raw values)
```

```

# (named) atomic vector
tinypplot(c(A = 1, B = 2, C = 3), type = "barplot")

# formula + data.frame method
tinypplot(GNP ~ Year, data = longley, type = "barplot")

tinypplot(demand ~ Time, data = BOD, type = "bar") # "bar" is a shorthand
tinypplot_add(type = "text", pos = 3, xpd = NA) # add y values as text

# reordering (just to demonstrate; these aren't sensible for a time variable)
tinypplot(demand ~ Time, data = BOD, type = "bar", xord = "asc")
jumble = c("7", "1", "5", "2", "4", "3") # note: Time = 6 is also missing
tinypplot(demand ~ Time, data = BOD, type = "bar", xlevels = jumble)

# useful top-level args:
# 1) xaxl to format the x labels, e.g. with a dictionary, keyword, or (here:)
#    function
# 2) xaxr to rotate long category labels (best used with a dynamic theme)
tinypplot(
  demand ~ Time, data = BOD,
  type = "bar",
  xaxl = function(x) paste("Time =", x),
  xaxr = 45,
  theme = "broadsheet"
)

#
## Aggregated vs grouped values (multiple ys per x)

# each person receives two drugs
sleep2 = transform(sleep, drug = group) # less misleading name

# default aggregation fun is mean
tinypplot(
  extra ~ ID, data = sleep2,
  type = "barplot",
  main = "Mean extra sleep from 2 soporific drugs"
)
# switch to diff (answers a more relevant q: who benefits most from drug 2?)
tinypplot(
  extra ~ ID, data = sleep2,
  type = "barplot", fun = diff,
  main = "Sleep gain (drug 2 vs drug 1)"
)
# we can sort in descending (or ascending) order too
tinypplot(
  extra ~ ID, data = sleep2,
  type = "barplot", fun = diff, xord = "desc",
  main = "Sleep gain (drug 2 vs drug 1), ordered"
)

# of course, we don't have to aggregate if we specify groups (stacked or non)

```

```

tinyplot(extra ~ ID | drug, data = sleep2, type = "barplot", beside = TRUE)

# Aside: We used automatic argument passing for 'xord', `fun`, etc. above.
# 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(
  extra ~ ID | drug, data = sleep2,
  type = type_barplot(beside = TRUE, xord = "desc", width = 0.5)
)

#
## matrix method (no formula required)

tinyplot(VADeaths, type = "barplot")
tinyplot(VADeaths, type = "barplot", beside = TRUE)
# etc. see ?tinyplot.matrix

#
## Frequency tables

# No y variable (frequency calculated on the fly)
tinyplot(~ cyl, data = mtcars, type = "barplot")
tinyplot(~ cyl | vs, data = mtcars, type = "barplot")
tinyplot(~ cyl | vs, data = mtcars, type = "barplot", beside = TRUE)

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

#
## Centering

# Centered barplot for conditional proportions of "dark" (black/brown) vs.
# "fair" (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)
)
tinyplot_add(type = "hline", col = "white") # "hline" b/c flip = TRUE

#
## Offset examples

```

```

# 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$value[1:3]), 0)
tinypplot(
  value ~ item | I(value < 0), data = d,
  type = type_barplot(offset = d$offset, lighten = FALSE),
  legend = FALSE
)
tinypplot_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$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")
tinypplot(
  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"
)
tinypplot_add(type = "hline") # "hline" b/c flip = TRUE
tinypplot_add(type = type_hline(1), lty = 2) # ditto

#
## Implicit zeros and empty cells (see the section of the same name above)

# No (mt)car has 8 cylinders and a straight engine, so that bar is a count
# of zero and is marked as such (flat, along the baseline)
tinypplot(~ cyl | vs, data = mtcars, type = "barplot", facet = "by")

# But in this example, the aggregating statistic is a mean rather than a
# count. The mean of unobserved combinations (e.g., carb==1 & vs==0) is
# undefined, so nothing is drawn for the empty cells
tinypplot(
  mpg ~ factor(carb), data = mtcars,

```

```

    type = "barplot",
    facet = ~ vs, facet.args = list(ncol = 1)
  )

  # ... use na.as.zero to override and mark as (implicit) zeros
  tinyplot(
    mpg ~ factor(carb), data = mtcars,
    type = type_barplot(na.as.zero = TRUE),
    facet = ~ vs, facet.args = list(ncol = 1)
  )

```

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
tinypplot(weight ~ feed, data = chickwts, type = "boxplot")

# Note: Specifying the type here is redundant. Like base plot, tinypplot
# automatically produces a boxplot if x is a factor and y is numeric
tinypplot(weight ~ feed, data = chickwts)

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

# Grouped boxplot example using a different dataset
# (C.f., the final example in `?boxplot`)
tinypplot(
  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
tinypplot(
  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
tinypplot(Sepal.Length ~ Petal.Length | Species, data = iris, type = "chull")

# layer filled convex hull(s) on top of points
tinypplot(Sepal.Length ~ Petal.Length | Species, data = iris, theme = "basic")
tinypplot_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,
  echo.bw = FALSE,
  alpha = NULL,
  singletons = c("warn", "drop", "none")
)
```

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 <code>adjust*bw</code> . 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 <code>fft</code> is used) and the final result is interpolated by <code>approx</code> . So it almost always makes sense to specify <code>n</code> as a power of two.
echo.bw	where to report the smoothing bandwidth (and the number of observations behind it), since neither is visible from the curve itself. The default FALSE reports nothing. TRUE is shorthand for "sub", and any combination of "sub" (subtitle), "cap" (caption), and "cat" (printed to the console, with "print" accepted as a synonym) may be given. A destination the user has already labelled is left alone. Bandwidths shared across subgroups are reported once and named as joint; individual bandwidths are reported per group, truncated after three so the label stays legible.
alpha	numeric value between 0 and 1 specifying the opacity of ribbon shading. If no alpha value is provided, then will default to <code>tpar("ribbon.alpha")</code> (i.e., probably 0.2 unless this has been overridden by the user in their global settings.)
singletons	character string indicating what to do with singleton groups, i.e. combinations of <code>by</code> and <code>facet</code> that consist of only 1 row. The default "warn" option removes any singleton cases and emits a warning reporting how many there were. "drop" does the same thing, but quietly. In either case the dropped groups may still be represented as empty facets in your plot. Finally, "none" skips all singleton checks and retains the affected groups; possibly leading to an error. Note that singletons require a numeric <code>bw</code> , since the data-driven bandwidth rules need at least 2 observations.

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 $\pm\text{Inf}$ and the density estimate is of the sub-density on $(-\text{Inf}, +\text{Inf})$.

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. Use `echo.bw` to put those details back in the subtitle, a caption, or the console.

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
tinypplot(
  ~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.
tinypplot(~Sepal.Length | Species, data = iris,
  ylim = c(0, 1.25), type = "density")      # mean (default)
tinypplot_add(joint.bw = "full", lty = 2)    # full data
tinypplot_add(joint.bw = "none", lty = 3)    # none (individual)
legend("topright", c("Mean", "Full", "None"), lty = 1:3, bty = "n", title = "Joint BW")

# neither the bandwidth nor the number of observations is visible from the
# curve, so `echo.bw` can report them: in the subtitle, a caption, or the
# console
tinypplot(~Sepal.Length | Species, data = iris,
  type = type_density(echo.bw = "sub"))

# individual bandwidths are listed per group
tinypplot(~Sepal.Length | Species, data = iris,
  type = type_density(joint.bw = "none", echo.bw = "sub"))
```

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
tinypplot(Sepal.Length ~ Petal.Length | Species, data = iris, type = "ellipse")

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

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

# custom confidence level
tinypplot(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,
  xlevels = NULL,
  xord = "asis"
)

type_pointrange(dodge = 0, fixed.dodge = FALSE, xlevels = NULL, xord = "asis")
```

Arguments

- | | |
|---------------|--|
| length | length of the edges of the arrow head (in inches). |
| dodge | <p>Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either:</p> <ul style="list-style-type: none"> • numeric value in the range $[0, 1)$. Note that values are scaled relative to the spacing of x-axis breaks, e.g. <code>dodge = 0.1</code> places the outermost groups one-tenth of the way to adjacent breaks, <code>dodge = 0.5</code> 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. <p>Default value is 0 (no dodging). While we do not check, it is <i>strongly</i> recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks.</p> |
| fixed.dodge | <p>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).</p> |
| xlevels, xord | <p>arguments controlling the order of the x variable, and hence of the x-axis. Supply one or the other; if both arguments are provided, <code>xlevels</code> takes precedence and <code>xord</code> is silently ignored.</p> <ul style="list-style-type: none"> • <code>xlevels</code> specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., <code>c("C", "B", "A")</code>), or a numeric vector of the corresponding level indexes (e.g. <code>3:1</code>). • <code>xord</code> instead accepts a keyword or custom function, which then <i>derives</i> the order from the data. Options are: <ul style="list-style-type: none"> – <code>"desc"</code> and <code>"asc"</code> rank the categories by their mean y value, largest or smallest first. (Long forms like <code>"descending"</code> and <code>"increasing"</code> are also accepted.) – <code>"minvar"</code> ranks them by variance, lowest first. This needs more than one observation per category, so it does not apply to the usual one-row-per-term coefficient table. – <code>"asis"</code> or <code>"rev"</code> permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order. – a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so <code>function(y) -median(y)</code> ranks by median, largest first. |

Note that x is only reordered when it is categorical (i.e., factor or character). A numeric x is plotted at its own values and cannot be reordered, so supplying either argument there is ignored with a warning.

Unlike most other plot types, `xord` defaults to `"asis"` here rather than `NULL`: these types are typically used for coefficient plots, where the row order of the data (e.g., the terms of a model) is usually intentional. Set `xord = NULL` to follow the factor levels instead, matching the other types.

Examples

```

tinytheme("basic") # filled points & background grid (but not dynamic yet)

#
## 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 for horizontal axis tick labels + appropriate spacing

tinytheme("classic") # or "clean(2)", "bw", "socviz", "float", ...
tinyplot(est ~ term, ymin = lwr, ymax = upr, data = coefs, type = "errorbar",
         flip = TRUE)
tinyplot_add(type = "hline", lty = 2) # "hline" b/c flip = TRUE (not vline!)

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"))
    )
  })
)

```

```

tinypointplot(est ~ 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`.

tinypointplot(est ~ 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.

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

#
## Handling (long/overlapping) tick labels

# You may face the annoyance of long and/or overlapping tick labels, e.g.
mod2 = lm(mpg ~ 0 + factor(cyl) * factor(am), mtcars)
coefs2 = data.frame(names(coef(mod2)), coef(mod2), confint(mod2))
colnames(coefs2) = c("term", "est", "lwr", "upr")
# (re-usable plot function)
demo_plot = function(...) {
  tinypointplot(
    est ~ term, ymin = lwr, ymax = upr,
    data = coefs2,
    type = "errorbar",
    xlab = NA,
    ylab = "MPG (miles per gallon)",
    main = "Impact on fuel efficiency",
    ...
  )
}
demo_plot()

# Here are some useful arguments (strategies) to avoid this annoyance...

# 1) dynamic theme + rotated x-labels
demo_plot(theme = "clean", xaxr = 45)
# 2) dynamic theme + labeller function (here: dictionary w/ newline spacing)

```

```
dict = c(
  "factor(cyl)4"      = "Manual\n4 Cyclinders",
  "factor(cyl)6"      = "Manual\n6 Cyclinders",
  "factor(cyl)8"      = "Manual\n8 Cyclinders",
  "factor(am)1"       = "Automatic\n4 Cyclinders",
  "factor(cyl)6:factor(am)1" = "Automatic\n6 Cyclinders",
  "factor(cyl)8:factor(am)1" = "Automatic\n8 Cyclinders"
)
demo_plot(theme = "clean", xaxl = dict)
# 3) dynamic theme + flipped axes
demo_plot(theme = "clean", flip = TRUE)
# 4) any combination of the above, e.g., labeller dictionary + flipped axes
demo_plot(theme = "clean", flip = TRUE, xaxl = dict)

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 <code>lines()</code> function, ex: <code>type="p", col="pink"</code> .

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.
tinypplot(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)
tinypplot(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 <code>glm</code> this can be a character string naming a family function, a family function or the result of a call to a family function. For <code>glm.fit</code> 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 tinypplot (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_hexbin

Hexagonal binning plot type

Description

Type function for hexagonal bin plots (a 2D analogue of a histogram). Point density is aggregated into a regular hexagonal lattice and each occupied cell is drawn as a hexagon whose fill colour encodes the number of points it contains. This is a lightweight (vectorized), pure R implementation of the canonical hexagonal binning algorithm of Carr et al. (1987).

Usage

```
type_hexbin(
  xbins = 30,
  shape = 1,
  mincnt = 1,
  maxcnt = Inf,
  border = NA,
  fun = NULL
)
```

Arguments

xbins	Integer. The number of bins partitioning the range of the x-axis. Default is 30.
shape	Numeric. The aspect ratio (height/width) of the plotting region used to compute the hexagon geometry. Default is 1.
mincnt, maxcnt	Integer. Cells with counts outside the [mincnt, maxcnt] range are omitted. Defaults keep all occupied cells.
border	Colour for the hexagon borders, passed to polygon . The default NA omits the border entirely. Set a colour (e.g. "black") for outlined hexagons. Passing the sentinel string "fill" matches the border to each cell's fill colour, which produces perfectly seamless tiling.
fun	Function used to summarise a <i>numeric</i> by variable within each cell (see Details). Defaults to NULL, which is equivalent to mean. Has no effect when by is absent (count mode) or discrete (modal mode).

Details

Hexagonal binning partitions the plotting region into a regular lattice of hexagons. Following the original Carr et al. (1987) algorithm, rows are spaced $\sqrt{3}/2$ apart in scaled space, with odd rows offset by half a cell in the x-direction, while each point is assigned to its nearest lattice node.

What the fill colour encodes depends on whether a by variable is supplied (e.g. $y \sim x \mid z$):

- **No by (default).** The fill encodes the *cell count*, mapped to a continuous colour gradient with an automatic colourbar legend.
- **Discrete by.** The fill encodes the *modal* (most frequent) level of the by variable within each cell, drawn with a discrete qualitative palette and legend. Useful for mapping the dominant category across a dense scatter.
- **Numeric by.** The fill encodes a per-cell summary of the by variable—by default the mean, or any other function passed via *fun* (e.g. *fun* = *sum*). Like the original no-by case, the legend is drawn as a continuous gradient. This is the hexagonal analogue of a binned heatmap.

In all three modes, the *mincnt*/*maxcnt* filters still apply to the raw cell *count*, so sparsely populated cells can be excluded regardless of what the colour encodes.

Because the hexagon size is derived from the data ranges, hexagons will only appear perfectly regular when the plot region's aspect ratio matches shape. Adjust shape (or the device dimensions) if you need visually regular hexagons.

The **tinypplot** implementation presented here is a compact (vectorised) translation of the original Carr et al. algorithm into base R. Users requiring the fuller feature set of a dedicated hexbinning package should consult **hexbin** as a canonical reference, which is built on top of compiled Fortran code. Note that we have validated our cell counts against this package to confirm they match exactly.

References

Carr, D. B., Littlefield, R. J., Nicholson, W. L., and Littlefield, J. S. (1987). *Scatterplot Matrix Techniques for Large N*. Journal of the American Statistical Association, 82(398), 424-436. Available: <https://doi.org/10.2307/2289444>

Carr, D., Lewin-Koh, N., Maechler, M., and Sarkar, D. (2024). *hexbin: Hexagonal Binning Routines*. R package version 1.28.5. Available: <https://doi.org/10.32614/CRAN.package.hexbin>. The **hexbin** package provides the canonical implementation of this algorithm, and was a valuable reference for validating our cell counts.

Examples

```
set.seed(1234)
dat = data.frame(x = rnorm(20000), y = rnorm(20000))

# "hexbin"/"hex" type convenience string
tinypplot(y ~ x, data = dat, type = "hexbin")

# Use type_hexbin() to pass extra arguments
tinypplot(y ~ x, data = dat, type = type_hexbin(xbins = 40))

# tinypplot's default palette logic maps darker colours (the end of the
# spectrum) to higher densities. For hexbin plots it can sometimes be more
```

```

# visually pleasing to reverse this, which users can do manually by passing a
# reversed palette.
tinypplot(
  y ~ x, data = dat, type = "hexbin",
  palette = hcl.colors(100, palette = "viridis", rev = TRUE)
)

# Passing a `by` grouping variable will colour cells according to a summary
# of this variable (in each hex cell) instead of density count. The default
# summary function depends on whether `by` is discrete or continuous:

# 1) Discrete grouping variable: each cell is coloured by its mode.
dat$g = cut(dat$x, breaks = c(-Inf, -1, 1, Inf), labels = c("lo", "mid", "hi"))
tinypplot(y ~ x | g, data = dat, type = "hexbin")

# 2) Continuous grouping variable: each cell is coloured by its mean.
# Example: Create a long version of the `volcano` dataset, and plot its
# elevations onto a gridded terrain map.
volc = data.frame(
  x = as.vector(row(volcano)),
  y = as.vector(col(volcano)),
  elevation = as.vector(volcano)
)
tinypplot(
  y ~ x | elevation, data = volc,
  type = "hexbin", xbins = 50,
  palette = terrain.colors(100, rev = TRUE)
)

```

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 <code>hist</code>. One of: • a vector giving the breakpoints between histogram cells, • a function to compute the vector of breakpoints, • a single number giving the number of cells for the histogram, • a character string naming an algorithm to compute the number of cells (see ‘Details’ of <code>hist</code>), • 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 <i>if and only if</i> 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 <code>free</code> 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 <code>facet.args = list(free)</code> , 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)
tinypplot(Nile, type = "histogram")

# Use `type_histogram()` to pass extra arguments for customization
tinypplot(Nile, type = type_histogram(breaks = 30))
tinypplot(Nile, type = type_histogram(breaks = 30, freq = FALSE))
# etc.

# Grouped histogram example
tinypplot(

```

```

    ~ Petal.Width | Species,
    type = "histogram",
    data = iris
  )

# Special case: `by`==`x`.
# Here we colour each bar (bin) according to its position along the x-axis
tinyplot(
  ~ Petal.Width | Petal.Width,
  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 $r \leftarrow x + \text{amount} * \text{runif}(n, -1, 1)$ where $n \leftarrow \text{length}(x)$.

When amount is NULL (the default) $\text{amount} \leftarrow \text{factor} * d/5$ where d is about the smallest difference between adjacent unique (up to fuzz) x values.

If $\text{amount} == 0$, $\text{amount} \leftarrow \text{factor} * (\max(x) - \min(x))/50$, (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
tinypplot(Sepal.Length ~ Species, data = iris, type = "jitter")

# Use `type_jitter()` to pass extra arguments for customization
tinypplot(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,
  xlevels = NULL,
  xord = NULL
)
```

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 | <p>Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either:</p> <ul style="list-style-type: none"> • numeric value in the range $[0, 1)$. Note that values are scaled relative to the spacing of x-axis breaks, e.g. <code>dodge = 0.1</code> places the outermost groups one-tenth of the way to adjacent breaks, <code>dodge = 0.5</code> 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. <p>Default value is 0 (no dodging). While we do not check, it is <i>strongly</i> recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks.</p> |
| 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). |
| xlevels, xord | <p>arguments controlling the order of the (categorical) x variable, and hence of the x-axis. Supply one or the other; if both arguments are provided, xlevels takes precedence and xord is silently ignored.</p> <ul style="list-style-type: none"> • xlevels specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., <code>c("C", "B", "A")</code>), or a numeric vector of the corresponding level indexes (e.g. <code>3:1</code>). • xord instead accepts a keyword or custom function, which then <i>derives</i> the order from the data. Options are: <ul style="list-style-type: none"> – "desc" and "asc" rank the categories by their mean y value, largest or smallest first. (Long forms like "descending" and "increasing" are also accepted.) – "minvar" ranks them by the variance of those values, lowest first. – "asis" or "rev" permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order. – a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so <code>function(y) -median(y)</code> ranks by median, largest first. |

Note that x is only reordered when it is categorical (i.e., factor or character). A numeric x is plotted at its own values and cannot be reordered, so supplying either argument there is ignored with a warning. Each argument defaults to NULL, i.e. keep the existing factor levels.

Categorical axes

Like the other plot types, `type_lines()` places categorical (factor or character) data according to the factor levels. Character variables are coerced with `factor()` and so end up in alphabetical order. To order the categories by their appearance in the data instead, use `xord = "asis"`, or set the levels explicitly, e.g. `factor(x, levels = unique(x))`.

Note that the lines themselves are always drawn in the order that the rows arrive in, exactly as base `lines()` does. Categories whose level order differs from their row order will therefore produce a zig-zag, just as an unsorted numeric x-variable would.

Examples

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

# Use `type_lines()` to pass extra arguments for customization
tinypplot(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)
tinypplot(
  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))
tinypplot(
  circumference ~ age | Tree, data = Orange2, type = "l",
  legend = list("direct", repel = TRUE), # auto repel
  theme = "socviz"
)
tinypplot(
  circumference ~ age | Tree, data = Orange2, type = "l",
  legend = list("direct", nudge_y = c("Tree 1" = 3, "Tree 3" = -5)), # manual
  theme = "socviz"
)

# A continuous `by` variable is also supported (as of tinypplot v0.8.0).
# This is particularly useful for trajectories, where a third variable (often
# time) orders the path.
time = seq(0, 8*pi, length.out = 600)
spiral = data.frame(time = time, x = time * cos(time), y = time * sin(time))
tinypplot(
  y ~ x | time, data = spiral,
  type = "l",
  lwd = 2, asp = 1, # optional args
  theme = "clean"
)

# The canonical time-path example: the x-z projection of a Lorenz attractor.
step = function(p, i) {
```

```

    p + 0.005 * c(
      10 * (p[2] - p[1]),
      p[1] * (28 - p[3]) - p[2],
      p[1] * p[2] - 8/3 * p[3]
    )
  }
  lz = do.call(rbind, Reduce(step, 1:6000, c(1, 1, 1), accumulate = TRUE))
  lorenz = data.frame(time = seq_len(nrow(lz)) * 0.005, x = lz[, 1], z = lz[, 3])
  tinyplot(z ~ x | time, data = lorenz, type = "l", theme = "clean")

```

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	<i>Local polynomial regression plot type</i>
------------	--

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,
  n = 100,
  weights = NULL
)
```

Arguments

span	the parameter α which controls the degree of smoothing.
degree	the degree of the polynomials to be used, normally 1 or 2. (Degree 0 is also allowed, but see the ‘Note’.)
family	a description of the error distribution and link function to be used in the model. For <code>glm</code> this can be a character string naming a family function, a family function or the result of a call to a family function. For <code>glm.fit</code> 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 <code>se = TRUE</code> . Default is 0.95.
n	integer. Number of equally-spaced points at which the fitted model is evaluated for drawing, per group. Default is 100. Note that this affects only where the fitted curve is sampled, not the fit itself, which always uses every observation. Larger values trace a wiggly fit more faithfully, at proportionally greater cost; consider raising it if you combine a small span with a large dataset.
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 tinypoint (which is evaluated with non-standard evaluation in the formula method, and takes precedence if both are given).

Plotting large data

`loess` can be a time-consuming algorithm. From a plotting perspective, though, the expensive half is usually not the model fit but the *prediction* step (y values corresponding to each x). `type_loess()` mitigates this in two ways.

First, the fitted model is evaluated on an equally-spaced grid of *n* points spanning the data range of each group, rather than at every observation. On large data this skips a great deal of redundant work—dozens of observations can crowd into a single pixel, so evaluating the curve at each of them costs time without revealing anything—while *n* points remain ample to trace a smooth curve. The model is still fitted on the full dataset, so the estimates are unchanged. It's only the points along *x* at which they are read off that differ. (If the *x*-axis is logarithmic, the grid is spaced evenly in $\log(x)$, so that the curve is traced evenly across the plot.)

Second, once prediction is cheap the fit itself becomes the bottleneck, and here `lowess` is substituted for `loess()` wherever the two are equivalent: unweighted local linear fits, without standard errors, on a non-logarithmic *x*-axis. That path smooths millions of observations in seconds. Standard errors, quadratic fits and weighted fits all fall back to `loess()`, as does a logarithmic *x*-axis, where `lowess()` interpolates too coarsely through the compressed decades.

Note that `se = TRUE` carries a hard ceiling of its own: `predict.loess`'s workspace grows quadratically in the number of observations—regardless of how many points you predict at—and fails outright at around 38,000. Confidence intervals are thus unavailable at the sizes where the `lowess()` path matters most, and `se = FALSE` is required to draw a smooth over very large data.

Examples

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

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

# The fast `lowess` path is used automatically where it is equivalent, which
# makes short work of larger datasets
tinypplot(sunspots, type = "l", col = "grey")
tinypplot_add(type = type_loess(span = 0.05, degree = 1, se = FALSE), lwd = 2)

# Raise `n` if a small `span` needs more than 100 points to trace faithfully
tinypplot(sunspots, type = "l", col = "grey")
tinypplot_add(type = type_loess(span = 0.05, degree = 1, se = FALSE, n = 500), lwd = 2)
```

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,
  xlevels = NULL,
  xord = NULL
)
```

Arguments

- | | |
|---------------|--|
| clim | Numeric giving the lower and upper limits of the character expansion (cex) normalization for bubble charts. |
| dodge | <p>Adjustment parameter for dodging overlapping points or ranges in grouped plots along the x-axis (or y-axis for flipped plots). Either:</p> <ul style="list-style-type: none"> • numeric value in the range $[0, 1)$. Note that values are scaled relative to the spacing of x-axis breaks, e.g. <code>dodge = 0.1</code> places the outermost groups one-tenth of the way to adjacent breaks, <code>dodge = 0.5</code> 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. <p>Default value is 0 (no dodging). While we do not check, it is <i>strongly</i> recommended that dodging only be used in cases where the x-axis comprises a limited number of discrete breaks.</p> |
| 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). |
| xlevels, xord | <p>arguments controlling the order of the (categorical) x variable, and hence of the x-axis. Supply one or the other; if both arguments are provided, xlevels takes precedence and xord is silently ignored.</p> <ul style="list-style-type: none"> • xlevels specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., <code>c("C", "B", "A")</code>), or a numeric vector of the corresponding level indexes (e.g. <code>3:1</code>). • xord instead accepts a keyword or custom function, which then <i>derives</i> the order from the data. Options are: <ul style="list-style-type: none"> – "desc" and "asc" rank the categories by their mean y value, largest or smallest first. (Long forms like "descending" and "increasing" are also accepted.) – "minvar" ranks them by the variance of those values, lowest first. – "asis" or "rev" permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order. |

- a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so `function(y) -median(y)` ranks by median, largest first.

Note that `x` is only reordered when it is categorical (i.e., factor or character). A numeric `x` is plotted at its own values and cannot be reordered, so supplying either argument there is ignored with a warning. Each argument defaults to `NULL`, i.e. keep the existing factor levels.

Examples

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

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

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

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

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

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

# Fancier version with dual legends and extra customization
tinypplot(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
tinypplot(1:9, c(2,1,2,1,NA,2,1,2,1), type = "polygon")

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

type_polypath	<i>Polypath polygon type</i>
---------------	------------------------------

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
tinypplot(
  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
tinypplot(
  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	<i>Quantile-Quantile plot (QQ)</i>
---------	------------------------------------

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

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

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

type_rect	<i>Rectangle plot type</i>
-----------	----------------------------

Description

Type function for plotting rectangles.

Usage

```
type_rect()
```

Details

Contrary to base [rect](#), rectangles in [tinypplot](#) must be specified using the `xmin`, `ymin`, `xmax`, and `ymax` arguments.

Examples

```

i = 4*(0:10)

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

# Same result with type_rect()
tinypplot(
  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 tinypplot scaffolding, which enables added functionality such as grouping and faceting.

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

Usage

```

type_ridge(
  scale = 1.5,
  joint.max = c("all", "facet", "by"),
  breaks = NULL,
  probs = NULL,
  ylevels = NULL,
  yord = 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,
singletons = c("warn", "drop", "none")
)

```

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, yord	arguments controlling the order of the y variable, and hence of the y-axis. Supply one or the other; if both arguments are provided, ylevels takes precedence and yord is silently ignored. • ylevels specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., c("C", "B", "A")), or a numeric vector of the corresponding level indexes (e.g. 3:1). • yord instead accepts a keyword or custom function, which then <i>derives</i> the order from the data. Options are: – "desc" and "asc" rank the ridges by their mean x value, largest or smallest first. (Long forms like "descending" and "increasing" are also accepted.) (Ridge plots have no separate response, so the ranking runs on the continuous x variable.) – "minvar" ranks them by the spread of each distribution, narrowest first. – "asis" or "rev" permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order. – a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so function(y) -median(y) ranks by median, largest first.

Note that a numeric y is coerced to a factor before the ridges are drawn, so it is reordered like any other categorical variable. Each argument defaults to NULL, i.e. keep the existing factor levels.

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 <code>adjust*bw</code> . 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 <code>gradient = TRUE</code> corresponding to <code>gradient = "viridis"</code> . If a character vector of length greater than 1 is used, then it should specify the colors in the palette, e.g., <code>gradient = hcl.colors(512)</code> .
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 <code>tinypplot(..., col =)</code> call (the <code>type_ridge(col =)</code> argument takes precedence if both are given). When neither is supplied, the outline defaults to the active theme's <code>col.default</code> (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 <code>by==y</code> or <code>by==x</code>), will default to 0.6.
singletons	character string indicating what to do with singleton groups, i.e. combinations of y, by, and facet that consist of only 1 row. The default "warn" option removes any singleton cases and emits a warning reporting how many there were. "drop"

does the same thing, but quietly. In either case the dropped groups may still be represented as empty ridge lines or facets in your plot. Finally, "none" skips all singleton checks and retains the affected groups; possibly leading to an error. Note that singletons require a numeric bw, since the data-driven bandwidth rules need at least 2 observations.

Technical note on gradient fills

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

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

# the "ridge(2)" themes are especially helpful for long y labels, due to
# dynamic plot adjustment
```

```

tinypplot(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)

tinypplot(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)
tinypplot(Month ~ Temp | Month, data = aq, type = "ridge")

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

# aside: pass explicit `type_ridge(col = <col>)` arg to set a different
# border color
tinypplot(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 tinypplot calls are equivalent)

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

tinypplot(Month ~ Temp, data = aq,
  type = type_ridge(gradient = TRUE, alpha = 0.5),
  main = "polygon fill (default)")
tinypplot(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)
tinypplot(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)
tinypplot(Month ~ Temp, data = aq, type = type_ridge(
  gradient = hcl.colors(250, "Dark Mint")[c(250:1, 1:250)],
  probs = 0:500/500))

## faceting works too. for example:

tinypplot(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
tinypplot(Month ~ Temp, facet = ~ Late, data = aq,
  type = type_ridge(scale = 1))
tinypplot(Month ~ Temp, facet = ~ Late, data = aq,
  type = type_ridge(scale = 1, joint.max = "facet"))

# jointly across all densities (default) vs. per by row
tinypplot(Month ~ Temp | Late, data = aq,
  type = type_ridge(scale = 1))
tinypplot(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 `tinypplot_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

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

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

type_segments	<i>Line segments plot type</i>
---------------	--------------------------------

Description

Type function for plotting line segments.

Usage

```
type_segments()
```

Details

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

Examples

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

# Same result with type_segments()
tinypplot(
```

```

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,
  xord = NULL,
  ylevels = NULL,
  yord = NULL,
  col = NULL,
  weights = NULL,
  lighten = FALSE,
  xaxlabels = NULL,
  yaxlabels = NULL
)

```

Arguments

- | | |
|---------------|--|
| breaks | if the explanatory variable is numeric, this controls how it is discretized. breaks is passed to hist and can be a list of arguments. |
| tol.ylab | convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly. |
| off | vertical offset between the bars (in per cent). It is fixed to 0 for spinograms and defaults to 2 for spine plots. |
| xlevels, xord | arguments controlling the order of the x variable, and hence of the x-axis. Supply one or the other; if both arguments are provided, <code>xlevels</code> takes precedence and <code>xord</code> is silently ignored. <ul style="list-style-type: none"> • <code>xlevels</code> specifies the levels <i>literally</i>, either a character vector of level names in the desired order (e.g., <code>c("C", "B", "A")</code>), or a numeric vector of the corresponding level indexes (e.g. <code>3:1</code>). |

- `xord` instead accepts a keyword or custom function, which then *derives* the order from the data. Options are:
 - “`desc`” and “`asc`” rank the categories by (weighted) frequency, i.e. most or least common first. (Long forms like “`descending`” and “`increasing`” are also accepted.)
 - “`asis`” or “`rev`” permute the existing levels without consulting the data at all. The former takes the categories in the order that they appear in the data, while the latter reverses the current level order.
 - a custom function that determines both the ranking statistic and its direction. The statistic is always sorted ascending, so `function(y) -median(y)` ranks by median, largest first.

Note that `x` is only reordered when it is categorical (i.e., factor or character). Both arguments are thus ignored for spinograms, which have a (binned) numeric `x` axis. Each argument defaults to `NULL`, i.e. keep the existing factor levels.

<code>ylevels</code> , <code>yord</code>	as for <code>xlevels</code> / <code>xord</code> above, but for the <code>y</code> variable. Note that <code>y</code> is always coerced to a factor for spineplots and spinograms, so these arguments are always binding if provided. Be aware that a numeric <code>y</code> gives one level per distinct value.
<code>col</code>	a vector of fill colors of the same length as <code>levels(y)</code> . The default is to call <code>gray.colors</code> .
<code>weights</code>	numeric. A vector of frequency weights for each observation in the data. If <code>NULL</code> all weights are implicitly assumed to be 1. If <code>x</code> is already a 2-way table, the weights are ignored.
<code>lighten</code>	logical. For grouped spineplots where the <code>y</code> variable is itself the grouping variable (i.e. <code>y == by</code>), should the fills use a lighter, opaque tint of the series colour(s)? Default is <code>FALSE</code> , i.e. the fills use the fully-saturated palette colour(s). (Unlike the other area types such as <code>type_barplot</code> , 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 <code>TRUE</code> to opt in to the lighter tint. Note that <code>lighten</code> has no effect on other spineplot displays (single-group or <code>x == by</code>), which always use a sequential shading ramp of the base colour.
<code>xaxlabels</code> , <code>yaxlabels</code>	[Deprecated] character vectors for annotation of the <code>x</code> and <code>y</code> axis. Use the top-level <code>xaxl</code> / <code>yaxl</code> arguments instead, which apply consistently across <code>tinypplot</code> types. These two type-specific arguments will be removed in a future release.

Examples

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

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

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

```

    type = type_spineplot(breaks = 4)
  )

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

# More idiomatic tinypplot way of drawing the previous plot: use y == by
tinypplot(
  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).
tinypplot(
  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()`.)
tinypplot(
  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
tinypplot(
  Survived ~ Sex | Class, data = ttnc,
  facet = "by",
  type = "spineplot", weights = Freq
)

# Fancier version. Note the smart inheritance of spacing etc.
tinypplot(
  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
tinypplot(dist ~ speed, data = cars, type = "spline")

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

type_summary	<i>Plot summary values of y at unique values of x</i>
--------------	---

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 [aggregate](#) and then passes the result to [type_lines](#) for drawing.

Usage

```
type_summary(fun = mean, dodge = 0, fixed.dodge = FALSE, ...)
```

Arguments

fun	summarizing function. Should be compatible with aggregate . Defaults to mean .
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. <code>dodge = 0.1</code> places the outermost groups one-tenth of the way to adjacent breaks, <code>dodge = 0.5</code> 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 <i>strongly</i> 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).
...	Additional arguments are passed to the <code>lines()</code> function, e.g. <code>type="p"</code> or <code>col="pink"</code> .

See Also

[aggregate](#) which performs the summarizing (aggregating) behind the scenes.

Examples

```
# Plot the mean chick weight over time
tinypplot(weight ~ Time, data = ChickWeight, type = "summary")

# Note: "mean" is the default function, so these are also equivalent:
# tinypplot(weight ~ Time, data = ChickWeight, type = type_summary())
# tinypplot(weight ~ Time, data = ChickWeight, type = type_summary(mean))

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

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

# Custom/complex function example
tinypplot(
  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 tinypplot argument, which additionally supports non-standard evaluation against data and takes precedence if both are given. See Examples.
labeller	A formatting function, convenience string, or dictionary (named vector or list) passed to tinylabell for formatting or relabelling the labels. Useful for ensuring that the text labels match the formatting of an axis, e.g. <code>labeller = "%"</code> 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
tinypplot(1:12, type = "text")
```

```

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

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

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

# for advanced customization, it's safer to pass args through `type_text()`
tinypplot(
  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)

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

# tip: use `repel = TRUE` to automatically nudge overlapping labels apart
tinypplot(
  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
tinypplot(
  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
tinypplot(hp ~ wt, data = mt)
tinypplot_add(subset = carb==2, col = "red", pch = 16)
tinypplot_add(subset = carb==2, col = "red", type = "text", labels = model, pos = 4)

# fancier plot that combines a theme and some other customizations
tinypplot(

```

```

    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_tile

Tile and heatmap plot types

Description

Type functions for tile plots, i.e. a grid of rectangles whose fill colour encodes a third variable. `type_tile()` is the default building block for these gridded shapes, drawing the values exactly as supplied. It underpins heatmaps, correlation matrices, calendar plots, confusion matrices, and similar displays.

`type_heatmap()` is a specialised case that first rescales the fill values within each category of one axis. Reach for it when those values are not already on a common scale.

Usage

```

type_tile(width = 1, height = 1)

type_heatmap(
  width = 1,
  height = 1,
  scale = c("none", "x", "y"),
  method = c("zscore", "rescale")
)

```

Arguments

width, height Numeric tile dimensions in data units. Both default to 1, which produces contiguous tiles on categorical (or unit-spaced numeric) axes. Values below 1 inset the tiles, leaving gaps between them. Recycled across tiles, so a vector may be used for variable sizes.

scale	Character. Should the by (fill) values be scaled <i>within</i> each category of one axis? One of "none" (default, i.e. the raw values are used), "x", or "y". Scaling is what makes a raw matrix legible when its variables span very different magnitudes: left alone, the largest-magnitude column monopolises the entire colour ramp. See Examples. Note that "x" and "y" refer to the axes <i>as written in the formula</i>, i.e. before any <code>flip = TRUE</code> is applied. We deliberately avoid base R's "row"/"column" wording, since a tile's position depends on which variable the user placed where in the formula, so there is no fixed matrix orientation to refer to. Rescaling is computed independently per facet; pooling across facets would pin a panel on a different scale to one end of the ramp and lose its internal structure. Since rescaled values are no longer in the units of the by variable, the legend title is annotated accordingly.
method	Character. How should the values be rescaled, if scale is not "none"? Either "zscore" (default) to centre each group and divide by its standard deviation, or "rescale" to map each group onto the unit interval [0, 1]. Ignored when scale = "none". "zscore" matches base R's <code>heatmap</code> and keeps values comparable across groups, since -1 means "one standard deviation below this group's mean" everywhere. "rescale" instead pins every group's minimum and maximum to the ends of the colour ramp, which uses the palette more fully but makes the endpoints an artefact of the transform rather than a feature of the data. Groups with no spread—a constant column, or a single tile—would divide by zero, so they are set to the midpoint of the target range (0.5 and 0 respectively) and a warning is emitted.

Details

Tile plots are specified as $z \sim x$ with the fill variable passed as the by grouping, i.e. `tinypLOT(y ~ x | z, type = "tile")`. The x and y variables may be factors, characters, or numerics; the by variable supplies the fill and will typically be numeric, yielding a continuous colour gradient and colourbar legend. Omitting by leaves the tiles unfilled, since there is nothing for the fill to encode; pass an explicit fill (or bg) if you want a uniform colour in that case.

Unlike the closely-related `type_rect`, which requires explicit `xmin/xmax/ymin/ymax` bounds, `type_tile()` derives the tile bounds for you: each tile is centred on its x/y position and extends `width/2` and `height/2` in each direction. Categorical axes are converted to consecutive integer positions and the axis tick labels are taken from the factor levels automatically.

Explicit bounds still take precedence. Passing any of `xmin`, `xmax`, `ymin`, or `ymax` leaves that dimension untouched, which is useful for irregular or unequal-width tiles (e.g. binned continuous data). Bounds may be given for one axis while the other is derived.

Note that tiles are opaque and drawn edge-to-edge, so the default axis padding and grid lines of most themes are redundant (and the grid is hidden behind the tiles in any case). We therefore ship a dedicated "heatmap" theme that removes the padding and grid, rotates the tick labels, and switches to a sequential palette. See `tinytheme()` and the Examples.

`type_heatmap()`'s `scale` argument is the analogue of the `scale` argument in base R's `heatmap`, and like the latter it z-scores along the chosen margin by default. Pass `method = "rescale"` to map each group onto the unit [0, 1] interval instead.

type_heatmap() also reverses the y-axis by default, so that the first row sits at the top, matching how one reads a matrix (and again cf. base R's heatmap() and image). Pass an explicit ylim to override. type_tile() makes no such adjustment, since it draws the values exactly as supplied.

Either way, note that scaling along a margin necessarily discards the *relative* spread of each group: a narrow-range column will occupy as much of the colour ramp as a wide-range one, since both are divided by their own measure of spread. That is the price of making a matrix of incomparable units legible; use scale = "none" (or type_tile()) when preserving cross-group magnitudes matters more.

See Also

type_rect for the lower-level rectangle type that type_tile() builds on, and tinytheme() for the companion "heatmap" theme.

Examples

```
# It is recommended to use the dedicated "heatmap" theme for tile plots
tinytheme("heatmap")

#
## type_tile ----

# Correlation matrix of the base `attitude` dataset in "long" form.
catt = as.data.frame(as.table(cor(attitude)), responseName = "Correlation")

tinypplot(Var1 ~ Var2 | Correlation, data = catt, type = "tile")

# fancier version where we reverse the y-axis (to mimic the usual correlation
# matrix layout), add white borders around each tile, and suppress the legend
# but layer on the values as text
tinypplot(
  Var1 ~ Var2 | Correlation, data = catt,
  type = "tile",
  col = "white",
  legend = FALSE,
  main = "Correlation matrix of base attitude dataset",
  xlab = NA, ylab = NA,
  ylim = "rev"
)
tinypplot_add(type = "text", labels = round(catt$Correlation, 2))

# Pass scaled tile widths and heights through type_tile() for a gridded look
tinypplot(
  Var1 ~ Var2 | Correlation, data = catt,
  type = type_tile(width = 0.9, height = 0.9)
)

# It doesn't really work for this example, but you can easily switch to a
# diverging palettes if it makes sense for your data
tinypplot(
  Var1 ~ Var2 | Correlation, data = catt,
  type = type_tile(width = 0.9, height = 0.9),
```

```

    palette = "tropic"
  )

  # Numeric axes work too, e.g. a (reshaped long) data.frame of volcano heights
  volc = data.frame(
    x      = as.vector(row(volcano)),
    y      = as.vector(col(volcano)),
    elevation = as.vector(volcano)
  )
  tinypplot(
    y ~ x | elevation, data = volc,
    type = "tile",
    theme = "void", # void theme looks better with this numeric example
    xlab = NA, ylab = NA,
    main = "Maunga Whau volcano"
  )

  #
  ## type_heatmap ----

  # Raw data matrices are usually dominated by their largest-magnitude column.
  # `type_heatmap()` can rescale within each column to make the rest legible.
  mt = as.data.frame(as.table(as.matrix(mtcars)))

  # first, the unscaled version: only `disp` and `hp` are visible
  tinypplot(
    Var1 ~ Var2 | Freq, data = mt,
    type = "heatmap",
    xlab = NA, ylab = NA
  )

  # and now scaled within each x variable (i.e., column). The default is to
  # z-score, matching base R's `heatmap(scale = "column")`.
  tinypplot(
    Var1 ~ Var2 | Freq, data = mt,
    type = type_heatmap(scale = "x"),
    xlab = NA, ylab = NA
  )

  # `method = "rescale"` maps each column onto [0, 1] instead. This uses the
  # colour ramp more fully, at the cost of pinning every column's min and max to
  # the same two colours.
  tinypplot(
    Var1 ~ Var2 | Freq, data = mt,
    type = type_heatmap(scale = "x", method = "rescale"),
    xlab = NA, ylab = NA
  )

  #
  ## tips ----

  # tip 1: use tinypplot.matrix() directly to avoid reshaping
  tinypplot(as.matrix(mtcars), type = type_heatmap(scale = "x"), col = "white")

```

```
# tip 2: use per-axis tick label scaling (cex) for dense heatmaps
tinypplot(as.matrix(mtcars), type = type_heatmap(scale = "x"), col = "white",
          theme = list("heatmap", cex.yaxis = 0.75, cex.xaxis = 1.25))

## restore the default theme
tinytheme()
```

type_violin

Violin and sina plot types

Description

Type functions for violin plots, which are an alternative to box plots for visualizing continuous distributions (by group) in the form of mirrored densities. `type_violin()` draws the smooth outline of each density, while `type_sina()` scatters the underlying observations as points within the same outline (similar to a beeswarm plot).

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,
  singletons = c("warn", "drop", "none")
)

type_sina(
  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,
  method = c("quasirandom", "random"),
  singletons = c("keep", "warn", "drop")
)
```

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 <code>adjust*bw</code> . 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 densities should be trimmed to the range of the data. Default is FALSE. For <code>type_sina()</code> this only affects the envelope that bounds the displacement, so it matters mainly for exact alignment with a <code>type_violin()</code> layer drawn the same way.
width	numeric (ideally in the range $[0, 1]$, although this isn't enforced) giving the normalized width of the individual violins. For <code>type_sina()</code> this is the width of the (undrawn) violin that the points are scattered inside.
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. Only applies to <code>type_violin()</code> , since <code>type_sina()</code> has no fill of its own.
singletons	character string indicating what to do with singleton groups, i.e. combinations of x, by, and facet that consist of only 1 row. Both types accept "warn", which removes any singleton cases and emits a warning reporting how many there were, and "drop", which does the same thing quietly. In either case the dropped groups may still be represented as empty violins or facets in your plot. The remaining option differs by type, as does the default. <code>type_violin()</code> defaults to "warn" and also accepts "none", which skips all singleton checks and retains the affected groups; possibly leading to an error. Note that singletons then require a numeric bw, since the data-driven bandwidth rules need at least 2 observations. <code>type_sina()</code> instead defaults to "keep", which draws the lone observation on its group's tick. No density can be estimated from a single point, but the point

itself is still worth showing, and discarding an observation from what is fundamentally a scatter plot is worse than discarding an unrenderable violin. There is no "none" for this type, since "keep" already retains these cases without error.

method character string giving how `type_sina()` spreads points across the available width at each y value. "quasirandom" (the default) walks a low-discrepancy sequence, which fills the width more evenly than chance does and—unlike `type_jitter`—is deterministic, so repeated calls give the same plot without setting a seed. "random" draws the displacements uniformly at random instead.

Details

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

A sina plot (Sidiropoulos et al., 2018) is closely related to a beeswarm plot, but is arguably the more principled of the two. Both spread a group's observations sideways to expose its shape. A beeswarm does so by packing points until they no longer collide, which makes its width an artefact of the *rendering*: change the symbol size or the device and the swarm changes shape. A sina instead displaces each point by the kernel density at its own y value, so its width is a property of the *data*, stable across devices and comparable between groups. The trade-off is occlusion: a beeswarm guarantees that no point hides another, whereas a sina accepts the occasional overlap. If you need collision-free packing, use a dedicated package such as **beeswarm**.

References

Sidiropoulos, N., Sohi, S. H., Pedersen, T. L., Porse, B. T., Winther, O., Rapin, N., and Bagger, F. O. (2018). *SinaPlot: An Enhanced Chart for Simple and Truthful Representation of Single Observations Over Multiple Classes*. *Journal of Computational and Graphical Statistics*, 27(3), 673-676. Available: <https://doi.org/10.1080/10618600.2017.1366914>

See Also

`type_boxplot`, `type_density` and `type_ridge` for the other ways of displaying a distribution by group, and `type_jitter` for displacing points without reference to a density.

Examples

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

# to match the defaults of `ggplot2::geom_violin()`, use `trim = TRUE` and
# `joint.bw = FALSE`
tinypplot(
  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
tinypplot(
```

```

    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)
tinypplot(weight ~ feed | feed, data = chickwts, type = "violin", legend = FALSE)

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

# the "sina" type shows the observations themselves, rather than a smooth
# outline drawn around them
tinypplot(weight ~ feed, data = chickwts, type = "sina")

# layering a sina on top of a violin lines up exactly, since the points are
# displaced by the violin's own half-width
tinypplot(weight ~ feed, data = chickwts, type = "violin")
tinypplot_add(type = "sina", pch = 16, col = "black")

# unlike `type_violin()`, `type_sina()` supports a continuous `by` variable;
# it colours the points rather than splitting them into groups
tinypplot(
  Sepal.Length ~ Species | Petal.Width, data = iris,
  type = "sina", pch = 16
)

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

```

Index

abline, [6](#), [23](#)
abs, [9](#)
aggregate, [102](#), [103](#)
aperm, [30](#)
approx, [67](#), [93](#), [101](#), [111](#)
array, [29](#)

bandwidth, [66](#), [92](#), [111](#)
boxplot, [64](#)
boxplot.stats, [64](#)

cairo_pdf, [47](#)
chull, [65](#)
cov, [70](#)

data.frame, [32](#)
density, [66](#), [92](#), [111](#)
dev.control, [8](#), [9](#), [24](#), [50](#)
dev.off, [6](#)
dnorm, [74](#)
draw_legend, [3](#), [6](#)

eigen, [70](#)

facet.args, [48](#)
factor(), [83](#)
family, [75](#), [85](#)
fft, [67](#), [93](#), [111](#)
format, [9](#)
formula, [24](#), [32](#)

get_saved_par, [6](#), [6](#), [23](#), [50](#)
glm, [75](#)
graphics::par, [50](#)
gray.colors, [99](#)

heatmap, [35](#), [107](#)
Hershey, [104](#)
hist, [79](#), [98](#)

image, [35](#), [108](#)

interaction, [14](#)

jitter, [80](#), [96](#)
jpeg, [23](#)

legend, [3](#)
lines(), [83](#)
list, [101](#)
lm, [84](#)
loess, [85](#), [86](#)
loess.control, [85](#)
lowess, [86](#)

matplot, [31](#), [34](#), [35](#)
matrix, [34](#)
mean, [102](#)

options, [46](#)

pairs, [32](#), [33](#)
palette, [21](#)
par, [6](#), [7](#), [14](#), [20](#), [23](#), [46](#), [47](#), [104](#)
pdf, [23](#), [47](#)
plot, [11](#), [17](#), [24](#), [82](#)
plot.density, [68](#)
plt (tinypoint), [11](#)
plt_add (tinypoint_add), [37](#)
png, [23](#)
polygon, [76](#), [88](#), [93](#)
polypath, [89](#)
predict.loess, [86](#)
print.recordedtinypoint
 (recordedtinypoint), [8](#)

qchisq, [70](#)

rasterImage, [93](#), [94](#)
recordedtinypoint, [8](#), [24](#), [31](#), [33](#), [35–37](#), [50](#)
recordPlot, [8](#), [9](#)
rect, [90](#)
replayPlot, [8](#), [9](#)

segments, [97](#)
 spline, [101](#)
 str.recordedtinypplot
 (recordedtinypplot), [8](#)
 svg, [23](#)

 table, [31](#)
 text, [6](#), [23](#)
 tinylabel, [4](#), [9](#), [16](#), [20](#), [48](#), [60](#), [104](#)
 tinypairs (tinypplot.data.frame), [32](#)
 tinypplot, [3](#), [4](#), [6](#), [9](#), [10](#), [11](#), [37](#), [75](#), [84](#), [85](#), [90](#),
 [97–99](#), [104](#)
 tinypplot(), [50](#)
 tinypplot.array, [29](#), [35](#)
 tinypplot.data.frame, [32](#)
 tinypplot.default, [38](#)
 tinypplot.formula, [32](#), [38](#)
 tinypplot.matrix, [29–31](#), [34](#)
 tinypplot.ts, [35](#)
 tinypplot_add, [8](#), [23](#), [37](#), [37](#), [52](#)
 tinypplot_add(), [97](#)
 tinytheme, [19](#), [24](#), [39](#), [44](#), [50](#)
 tinytheme(), [45](#), [107](#), [108](#)
 tinytheme_get, [44](#), [50](#)
 tinytheme_get(), [42](#), [45](#)
 tinytheme_list (tinytheme_register), [45](#)
 tinytheme_list(), [42](#)
 tinytheme_register, [44](#), [45](#)
 tinytheme_register(), [41](#), [42](#)
 tinytheme_unregister
 (tinytheme_register), [45](#)
 toupper, [9](#)
 tpar, [16](#), [17](#), [19](#), [23](#), [42](#), [44](#), [46](#), [93](#)
 ts, [35](#)
 type_abline, [51](#)
 type_abline(), [18](#)
 type_area, [53](#)
 type_area(), [17](#)
 type_barplot, [58](#), [99](#)
 type_barplot(), [18](#), [54](#)
 type_boxplot, [64](#), [112](#)
 type_boxplot(), [18](#)
 type_chull, [65](#)
 type_chull(), [18](#)
 type_density, [66](#), [93](#), [112](#)
 type_density(), [18](#)
 type_ellipse, [69](#)
 type_ellipse(), [18](#)
 type_errorbar, [70](#)
 type_errorbar(), [17](#)
 type_function, [74](#)
 type_function(), [18](#)
 type_glm, [75](#)
 type_glm(), [18](#)
 type_heatmap (type_tile), [106](#)
 type_heatmap(), [18](#)
 type_hexbin, [76](#)
 type_hexbin(), [18](#)
 type_hist, [14](#)
 type_hist (type_histogram), [78](#)
 type_histogram, [78](#)
 type_histogram(), [18](#)
 type_hline (type_abline), [51](#)
 type_hline(), [18](#)
 type_jitter, [80](#), [112](#)
 type_jitter(), [17](#)
 type_lines, [81](#), [102](#)
 type_lm, [84](#)
 type_lm(), [14](#), [18](#)
 type_loess, [85](#)
 type_loess(), [18](#)
 type_pointrange (type_errorbar), [70](#)
 type_pointrange(), [17](#)
 type_points, [86](#)
 type_points(), [23](#)
 type_polygon, [88](#)
 type_polygon(), [17](#)
 type_polypath, [89](#)
 type_polypath(), [17](#)
 type_qq, [90](#)
 type_qq(), [18](#)
 type_rect, [90](#), [107](#), [108](#)
 type_rect(), [17](#)
 type_ribbon (type_area), [53](#)
 type_ribbon(), [17](#)
 type_ridge, [91](#), [112](#)
 type_ridge(), [18](#), [41](#)
 type_rug, [96](#)
 type_rug(), [17](#)
 type_segments, [97](#)
 type_segments(), [17](#)
 type_sina (type_violin), [110](#)
 type_spineplot, [98](#)
 type_spineplot(), [14](#), [18](#)
 type_spline, [101](#)
 type_spline(), [18](#)
 type_summary, [102](#)

`type_summary()`, [18](#)
`type_text`, [103](#)
`type_text()`, [14](#), [17](#)
`type_tile`, [106](#)
`type_tile()`, [17](#), [41](#)
`type_violin`, [110](#)
`type_violin()`, [18](#)
`type_vline (type_abline)`, [51](#)
`type_vline()`, [18](#)

`xy.coords`, [14](#)