Package: ggtime (via r-universe)

October 18, 2026

Title Grammar of Graphics and Plot Helpers for Time Series Visualization

Version 1.1.0.9000

Description Extends the capabilities of 'ggplot2' by providing grammatical elements and plot helpers designed for visualizing temporal patterns. The package implements a grammar of temporal graphics, which leverages calendar structures to highlight changes over time. The package also provides plot helper functions to quickly produce commonly used time series graphics, including time plots, season plots, and seasonal sub-series plots.

License GPL (>= 3)

URL https://pkg.mitchelloharawild.com/ggtime/, https://github.com/mitchelloharawild/ggtime

Imports ggplot2, grid, gtable, lifecycle, rlang, scales, tsibble, fabletools, dplyr, tidyr, vctrs, cli, vecvec, mixtime (>= 0.3.0.9000), S7, utils

Suggests testthat (>= 3.0.0), tibble, tsibbledata, lubridate, feasts, fable, ggrepel, roxygen2, ggdist, distributional, vdiffr, svglite, fontquiver, sysfonts, showtext

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

BugReports https://github.com/mitchelloharawild/ggtime/issues

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libicu-dev

Repository https://robjhyndman.r-universe.dev

Date/Publication 2026-10-18 01:49:01 UTC

RemoteUrl https://github.com/mitchelloharawild/ggtime

RemoteRef HEAD

RemoteSha d2faafec5cffd81918eecfaf1f9ce694ea0c992f

Contents

aes_nudge . . . 2
autoplot.dcmp_ts . . . 4
autoplot.fbl_ts . . . 5
autoplot.tbl_cf . . . 6
autoplot.tbl_ts . . . 6
coord_calendar . . . 7
coord_loop . . . 10
geom_candlestick . . . 13
geom_time_line . . . 19
gg_arma . . . 24
gg_irf . . . 25
gg_lag . . . 26
gg_season . . . 27
gg_subseries . . . 29
gg_tsdisplay . . . 30
gg_tsresiduals . . . 31
scale_candlestick . . . 32
scale_mixtime . . . 33
scale_mixtime_continuous . . . 37
transform_warp . . . 40

Index **43**

`aes_nudge` *Aesthetic specific alignment of discrete time*

Description

Positioning discrete time points (e.g. months) on a continuous time scale (e.g. days) is indeterminate - which day should represent a month? This is resolved by aligning each time point within its granularity, where 0 is start alignment, 1 is end alignment, and 0.5 is center alignment.

Usage

```
aes_nudge(
  center = 0.5,
  left = 0,
  right = 1,
  x = center,
  xmin = left,
  xmax = right,
  xend = center,
  xintercept = center,
  xmin_final = left,
  xmax_final = right,
  xlower = left,
```

```
  xmiddle = center,
  xupper = right,
  x0 = center,
  y = center,
  ymin = left,
  ymax = right,
  yend = center,
  yintercept = center,
  ymin_final = left,
  ymax_final = right,
  ylower = left,
  ymiddle = center,
  yupper = right,
  y0 = center
)
```

Arguments

`center`, `left`, `right`
: Alignment applied to centered (e.g. `x`, `xend`), lower (e.g. `xmin`, `xlower`), and upper (e.g. `xmax`, `xupper`) positional aesthetics respectively. Setting these changes the default for all semantically equivalent aesthetics below.

`x`, `xmin`, `xmax`, `xend`, `xintercept`, `xmin_final`, `xmax_final`, `xlower`, `xmiddle`, `xupper`, `x0`
: Alignment for individual `x` aesthetics.

`y`, `ymin`, `ymax`, `yend`, `yintercept`, `ymin_final`, `ymax_final`, `ylower`, `ymiddle`, `yupper`, `y0`
: Alignment for individual `y` aesthetics.

Details

Different positional aesthetics often require different alignments. A ribbon spanning a month should start at the beginning of the month and end at the end of it, while a line should pass through its center. `aes_nudge()` specifies these alignments per aesthetic, and is passed to the `align_discrete` argument of `scale_x_mixtime()`.

Value

A function that takes an aesthetic name and returns its alignment, suitable for the `align_discrete` argument of `scale_x_mixtime()`.

Examples

```
# Center aligned points, with intervals spanning the full granularity
aes_nudge(center = 0.5, left = 0, right = 1)

# Align all time points to the start of their granularity
aes_nudge(center = 0, left = 0, right = 0)
```

autoplot.dcmp_ts *Decomposition plots*

Description

Produces a faceted plot of the components used to build the response variable of the dable. Useful for visualising how the components contribute in a decomposition or model.

Usage

```
## S3 method for class 'dcmp_ts'
autoplot(object, .vars = NULL, scale_bars = TRUE, level = c(80, 95), ...)
```

Arguments

object	A dable.
.vars	The column of the dable used to plot. By default, this will be the response variable of the decomposition.
scale_bars	If TRUE, each facet will include a scale bar which represents the same units across each facet.
level	If the decomposition contains distributions, which levels should be used to display intervals?
...	Further arguments passed to ggplot2::geom_line(), which can be used to specify fixed aesthetics such as colour = "red" or size = 3.

Value

A ggplot object showing a set of time plots of the decomposition.

Examples

```
library(fabletools)
library(feasts)
tsibbledata::aus_production %>%
  model(STL(Beer)) %>%
  components() %>%
  autoplot()
```

autoplot.fbl_ts *Plot a set of forecasts*

Description

Produces a forecast plot from a fable. As the original data is not included in the fable object, it will need to be specified via the `data` argument. The `data` argument can be used to specify a shorter period of data, which is useful to focus on the more recent observations.

Usage

```
## S3 method for class 'fbl_ts'
autoplot(object, data = NULL, level = c(80, 95), show_gap = TRUE, ...)

## S3 method for class 'fbl_ts'
autolayer(
  object,
  data = NULL,
  level = c(80, 95),
  point_forecast = list(mean = mean),
  show_gap = TRUE,
  ...
)
```

Arguments

`object`	A fable.
`data`	A tsibble with the same key structure as the fable.
`level`	The confidence level(s) for the plotted intervals.
`show_gap`	Setting this to `FALSE` will connect the most recent value in `data` with the forecasts.
`...`	Further arguments passed used to specify fixed aesthetics for the forecasts such as `colour = "red"` or `linewidth = 3`.
`point_forecast`	The point forecast measure to be displayed in the plot.

Examples

```
library(fable)
library(tsibbledata)

fc <- aus_production %>%
  model(ets = ETS(log(Beer) ~ error("M") + trend("Ad") + season("A"))) %>%
  forecast(h = "3 years")

fc %>%
  autoplot(aus_production)
```

```
aus_production %>%
  autoplot(Beer) +
  autolayer(fc)
```

`autoplot.tbl_cf` *Auto- and Cross- Covariance and -Correlation plots*

Description

Produces an appropriate plot for the result of `feasts::ACF()`, `feasts::PACF()`, or `feasts::CCF()`.

Usage

```
## S3 method for class 'tbl_cf'
autoplot(object, level = 95, ...)
```

Arguments

`object`	A tbl_cf object (the result `feasts::ACF()`, `feasts::PACF()`, or `feasts::CCF()`).
`level`	The level of confidence for the blue dashed lines.
`...`	Unused.

Value

A ggplot object showing the correlations.

`autoplot.tbl_ts` *Plot time series from a tsibble*

Description

Produces a time series plot of one or more variables from a tsibble. If the tsibble contains a multiple keys, separate time series will be identified by colour.

Usage

```
## S3 method for class 'tbl_ts'
autoplot(object, .vars = NULL, ...)

## S3 method for class 'tbl_ts'
autolayer(object, .vars = NULL, ...)
```

Arguments

`object`	A tsibble.
`.vars`	A bare expression containing data you wish to plot. Multiple variables can be plotted using `ggplot2::vars()`.
`...`	Further arguments passed to `ggplot2::geom_line()`, which can be used to specify fixed aesthetics such as `colour = "red"` or `size = 3`.

Value

A ggplot object showing a time plot of a time series.

Examples

```
library(dplyr)
tsibbledata::gafa_stock %>%
 autoplot(vars(Close, log(Close)))
```

---

coord_calendar *Calendar coordinates*

Description

Arranges time series data into a calendar-like layout of rows and columns. Data is cut into loops as in `coord_loop()`, with each loop becoming its own row and column of a grid rather than being overlaid.

Usage

```
coord_calendar(
  cells = day(1L),
  rows = week(1L),
  blocks = NULL,
  panes = month(1L),
  cols = quarter(1L),
  pane_spacing = 0.25,
  col_spacing = 0.1,
  cell_ratio = NULL,
  label_cells = "{cyc(day, month)}",
  label_rows = NULL,
  label_blocks = NULL,
  label_panes = "{cyc(month, year, label = TRUE, abbreviate = TRUE)}",
  label_cols = NULL,
  time = "x",
  xlim = NULL,
  ylim = NULL,
  expand = FALSE,
```

```
  default = FALSE,
  clip = "on",
  coord = coord_cartesian()
)
```

Arguments

`cells`
: Size of a calendar cell (see the Granule hierarchy section); governs the cell labels and the only gridline drawn along the time axis.
  - `NULL`: no cells, no gridlines, no cell labels
  - a granule or duration, e.g. `mixtime::days(1L)`

  Defaults to `day(1L)`.

`rows`
: Size of a calendar row: `coord_loop()`'s `time_loops` under a calendar-specific name.
  - `NULL`: a single row spanning the whole column
  - a granule or duration, e.g. `mixtime::days(7L)`

  Defaults to `week(1L)`, or `day(7L)` if the calendar has no `week`.

`blocks`
: Size of a calendar block: rows are grouped and marked with a thicker gridline where each block starts.
  - `NULL` (default): no blocks
  - a granule or duration, e.g. `mixtime::months(1L)`

`panes`
: Size of a calendar pane: rows are grouped and set apart by a gap rather than a rule. Must be coarser than `rows`, no coarser than `cols`.
  - `NULL`: no panes
  - a granule or duration, e.g. `mixtime::months(1L)`

  Defaults to `month(1L)`, silently dropped if it doesn't fit.

`cols`
: Size of a calendar column, arranged left to right with no wrapping.
  - `NULL`: a single column spanning the whole time range
  - a granule or duration, e.g. `mixtime::quarters(1L)`

  Defaults to `quarter(1L)`, or `month(3L)` if the calendar has no `quarter`.

`pane_spacing`, `col_spacing`
: Gap between panes of rows / between columns, as a fraction of one row's height / one column's width.

`cell_ratio`
: Aspect ratio of a calendar cell, as its height divided by its width, in the manner of `ggplot2::coord_fixed()` but applied to a cell rather than to the data units of the axes.
  - `NULL` (default): the panel takes whatever shape it is given
  - a number, e.g. `1` for the square cells of a printed calendar

  Fixing a cell fixes the panel's own aspect ratio, so this cannot be combined with a `coord` that sets its own `ratio`, and `theme(aspect.ratio = )` overrides it. Applies to a whole row's tile when there are no cells to divide it.

`label_cells`, `label_rows`, `label_blocks`, `label_panes`, `label_cols`
: How to label each instance of a granule:

- a mixtime format string, as time_labels of scale_x_mixtime(), e.g. "{cyc(day, month)}" (a bare "{day}" is not valid)
- a function of the granule's times, returning a character vector
- NULL (default except for label_cells/label_panes): no labels

Named by the time each instance starts, except a block or pane (which spans several rows), named by the time in the middle of the group.

time — A string specifying which aesthetic contains the time variable that should be looped over. Default is "x".

xlim, ylim — Limits for the x and y axes. NULL means use the default limits.

expand — Logical indicating whether to expand the coordinate limits. Default is FALSE.

default — Logical indicating whether this is the default coordinate system. Default is FALSE.

clip — Should drawing be clipped to the extent of the plot panel? A setting of "on" (the default) means yes, and a setting of "off" means no.

coord — The underlying coordinate system to use. Default is coord_cartesian().

Details

Useful for visualizing long time spans with events over short intervals, such as holidays. Cuts the time axis at every calendar boundary at once, folds each piece into its row's window, and offsets it into its cell of the grid. As with coord_loop(), geometries crossing a boundary are cut, justified per align_discrete (see scale_x_mixtime()).

Granule arguments

cells/rows/blocks/panes/cols each accept:

- NULL
- a duration, e.g. mixtime::days(1L)
- a time granule, e.g. mixtime::cal_gregorian$month(1L)
- a bare expression, e.g. day(1L) or month(1L), naming a granule of whichever calendar the time axis resolves to (from the scale's common chronon, or the Gregorian calendar for a plain Date/POSIXct axis)

Granule hierarchy

Granules sit in a strict hierarchy: col, pane, block, row, cell. A granule never straddles a boundary of anything above it; a row cut short by a coarser boundary is left blank for the rest of its width, as on a printed calendar.

Breaks and labels

The time axis is broken at every cells boundary and labelled as a position within the rows cycle (e.g. "Mon", "Tue", ...), unless breaks/labels/time_breaks/time_labels are set explicitly (see scale_x_mixtime()). Falls back to the scale's own breaks when cells or rows is NULL, cells can't cut the axis, or a row holds too many cells to name individually.

Theming

Each granule has its own theme elements, `ggtime.calendar.<granule>.line`, `.background` and `.text` (`<granule>` = `cell`, `row`, `block`, `pane` or `col`), inheriting from `panel.grid`, `panel.background` and `text`. The panel's own grid is not drawn; the `cell` granule rules the time axis instead.

```
+ theme(
  ggtime.calendar.block.line = element_line(linewidth = 1, linetype = "22"),
  ggtime.calendar.cell.line = element_blank()
)
```

A granule's labels are justified within the granule they belong to, each in a different default corner so several can be labelled at once without colliding.

Examples

```
library(ggplot2)
library(mixtime)

# Hourly pedestrian counts in Melbourne, as mixtime time points.
pedestrian <- dplyr::mutate(tsibble::pedestrian, Time = datetime(Date_Time))

# A monthly calendar arrangement of pedestrian counts, showing the high
# activity at Birrarung Marr during the Australian Open in late January.
pedestrian_2015 <- dplyr::filter(
  pedestrian,
  mixtime::year(Time) == mixtime::year(2015L),
  Sensor == "Birrarung Marr"
)
ggplot(pedestrian_2015, aes(x = Time, y = Count, color = Sensor)) +
  geom_line() +
  coord_calendar(rows = month(1L), cols = NULL) +
  scale_x_mixtime(
    time_breaks = mixtime::days(1L),
    time_labels = "{cyc(day, cal_isoweek$week, label = TRUE, abbreviate = TRUE)}"
  ) +
  theme(
    legend.position = "bottom",
    axis.text.y = element_blank(), axis.ticks.y = element_blank()
  )
```

---

coord_loop *Looped coordinates*

Description

The looped coordinate system loops the cartesian coordinate system around specific loop points. This is particularly useful for visualising seasonal patterns that repeat over calendar periods, since the shape of seasonal patterns can be more easily seen when superimposed on top of each other.

Usage

```
coord_loop(
  loops = waiver(),
  time_loops = waiver(),
  time = "x",
  xlim = NULL,
  ylim = NULL,
  expand = FALSE,
  default = FALSE,
  clip = "on",
  coord = coord_cartesian()
)
```

Arguments

`loops`	Loop the time scale around a calendrical granularity, one of: • `NULL` or `waiver()` for no looping (the default) • A `mixtime` vector giving time points at which the `time` axis should loop • A function that takes the limits as input and returns loop points as output
`time_loops`	A duration giving the distance between temporal loops, such as `mixtime::weeks(2L)` or `mixtime::years(10L)`. If both `loops` and `time_loops` are specified, `time_loops` wins.
`time`	A string specifying which aesthetic contains the time variable that should be looped over. Default is `"x"`.
`xlim`, `ylim`	Limits for the x and y axes. `NULL` means use the default limits.
`expand`	Logical indicating whether to expand the coordinate limits. Default is `FALSE`.
`default`	Logical indicating whether this is the default coordinate system. Default is `FALSE`.
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no.
`coord`	The underlying coordinate system to use. Default is `coord_cartesian()`.

Details

This coordinate system is particularly useful for visualizing seasonal or cyclic patterns in time series data. It works by:

1. Dividing the time axis into loops based on the specified loop period
2. Folding the time values of every loop into the first loop's window
3. Cutting geometries that cross a loop boundary into one piece per loop

Since the looping is applied to the data rather than to the drawing, the panel is drawn only once regardless of how many loops are shown. The cost of the plot is therefore independent of the number of loops.

Value

A `Coord` ggproto object that can be added to a ggplot.

Practical usage

The looped coordinate system reveals patterns that repeat over regular time periods, such as annual seasonality in monthly data, or weekly patterns in daily data. It allows the `[x/y]` time aesthetic to be specified continuously, and loops the time axis around specified time intervals. This allows time within seasonal periods to be compared directly, and highlights the shape of seasonal patterns. This is commonly used in time series analysis to identify the peaks and troughs of seasonal patterns.

A key advantage of time being specified continuously is that the connection between the end of one seasonal period and the start of the next is preserved. This is otherwise lost when time is discretised into ordered factors (e.g. months of the year, or days of week). This allows lines and other geometries to be drawn across seasonal boundaries, such as a line that connects December to January when plotting annual seasonality.

Looping arranges time cyclically, so the time axis describes a position within the loop rather than the passage of time. The axis is labelled to match: monthly data looped over years is labelled with months of the year ("Jan", "Feb", ...), and daily data looped over weeks with days of the week ("Mon", "Tue", ...). Which labels are appropriate depends on both the chronon of the data and the loop's cycle, and is determined by the calendar being used. Labels given with the `labels` or `time_labels` options of `scale_x_mixtime()` are used unchanged, since they say how the user wants time written.

The justification of looping can be controlled using the `align_discrete` option of `scale_x_mixtime()`, where values from 0 to 1 specify the alignment. Left alignment (`align_discrete = 0`) places inter-seasonal connections on the left of the panel, right alignment (`align_discrete = 1`) uses the right side, and center alignment (`align_discrete = 0.5`, the default) uses equal spacing on both ends of the season.

Why not use seasonal factors?

Using factors to represent seasonal periods is common, but prone to errors and is very limiting. Suppose you want to visualize weekly seasonality in daily data. You could convert the date into a day of week factor (e.g. with `lubridate::wday(date, label = TRUE)`), but this loses information about the year and week of the observation. In order to correctly draw lines connecting each day of the week (avoiding sawtooth patterns), you would additionally need to group by year and week to separately identify each line segment. The aesthetic mapping for plotting this pattern would look something like:

```
aes(
  x = lubridate::wday(date, label = TRUE),
  group = interaction(lubridate::year(date), lubridate::week(date)),
  y = value
)
```

These operations are error-prone, cumbersome, and are complicated to update to show different seasonal patterns. For example, if you wanted to instead show the annual seasonal pattern, both the `x` and `group` aesthetics would need to be changed (to day of year and year respectively). Any errors in this process would produce sawtooth patterns or other artifacts in the plot.

Another common error in discretizing time into seasonal factors is incorrect ordering of the factor levels. For example, if you instead used `strftime(date, "%a")` to get the day of week, the levels would be sorted alphabetically rather than in time order ("Fri", "Mon", "Sat", ...). No-one wants to Monday to follow Friday!

Discretizing time into seasonal factors also prevents plotting the seasonal pattern across multiple granularities. For example when visualizing weekly seasonality across data at daily and hourly frequencies, both day of week and hour of week are needed. Since these factors have different levels, they cannot be plotted on the same axis. In contrast, it is possible to plot both daily and hourly data on the same axis using `scale_x_mixtime()`, which can then be looped over weekly periods with `coord_loop(time_loops = mixtime::weeks(1L))`.

Another subtle issue of using factors instead of continuous time is that spacing between time points is regularized. For example, when plotting the annual seasonal pattern with months as a factor, each month is given equal width on the x-axis despite the fact that months have different lengths.

Known limitations

Geometries are cut into loops by splitting the paths and rings that make them up, which requires those shapes to be monotone along the time axis. This works works for lines, paths, ribbons, areas, rects, tiles, bars, columns and segments. A non-monotone concave polygon that crosses a loop boundary is not cut correctly.

Examples

```
library(ggplot2)
library(ggtime)
library(mixtime)

# Basic usage with US accidental deaths data
uad <- tsibble::as_tsibble(USAccDeaths)
# Requires mixtime, POSIXct, or Date time types
uad$index <- mixtime::yearmonth(uad$index)

p <- ggplot(uad, aes(x = index, y = value)) +
  geom_line()

# Original plot
p

# With yearly looping to show seasonal patterns
p + coord_loop(time_loops = mixtime::years(1L))
```

`geom_candlestick` *Candlestick or OHLC chart layers*

Description

A candlestick summarises the values observed within one instance of a time granule with four statistics known as 'OHLC': the first value (`open`), the largest (`high`), the smallest (`low`), and the last (`close`). These summary statistics for each granule are drawn as a vertical line spanning from `low` to `high`, overlaid with a rectangle spanning `open` to `close`. The colour and fill of the geometry indicate the change's direction, by default rising values use green while falling values use red.

Candlesticks are cut on the `granule`'s boundaries, and the width of the candlestick reflects the length of time in that granule. Granules of uneven length (e.g. month granules on a day chronon scale via `geom_candlestick(granule = month(1L)) + scale_x_mixtime(time_chronon = day(1L))`) are therefore drawn at uneven widths, each filling its own instance.

Usage

```
geom_candlestick(
  mapping = NULL,
  data = NULL,
  stat = "ohlc",
  position = "identity",
  ...,
  granule,
  width = 0.9,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_ohlc(
  mapping = NULL,
  data = NULL,
  geom = "candlestick",
  position = "identity",
  ...,
  granule,
  width = 0.9,
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data` The data to be displayed in this layer. There are three options:

If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.

A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat` The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer.

An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.

- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`granule` The time granule to summarise within, given as:

- a duration, e.g. `mixtime::days(1L)`
- a time granule, e.g. `mixtime::cal_gregorian$month(1L)`
- a bare expression, e.g. `month(1L)`, naming a granule of whichever calendar the time axis uses (see the "Time granules" section below).

There is no default: a candlestick is defined by the granule it summarises, and the granule that suits a series is a property of the analysis rather than of the data.

`width` The width of each candlestick's body, as a proportion of the granule instance it summarises. Defaults to `0.9`, leaving a small gap between neighbouring candlesticks. A `width` of `1` makes each body span its whole granule instance, so that neighbours touch.

`lineend` Line end style for the high-low line (round, butt, square).

`linejoin` Line join style for the open-close body (round, mitre, bevel).

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`orientation` The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.

`show.legend` logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`inherit.aes` If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `annotation_borders()`.

`geom` The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following:

- A `Geom` ggproto subclass, for example `GeomPoint`.
- A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`.
- For more information and other ways to specify the geom, see the layer geom documentation.

Positional aesthetics

Each of the four summary statistics is carried by an axis-specific pair of the positional aesthetics ggplot2::geom_boxplot() uses: [x/y]lower, [x/y]upper, [x/y]min and [x/y]max. Being ggplot2's own position aesthetics, the position scales transform and train them, and ggplot2::flip_data() pairs the two members up, so only the member on the axis holding the values is ever mapped:

Statistic	Values on y (time on x)	Values on x (time on y)
open	lower	xlower
high	ymax	xmax
low	ymin	xmin
close	upper	xupper

ggplot2 spells the y member of the lower/upper pair without its y prefix (lower, not ylower), as ggplot2::geom_boxplot() does.

stat_ohlc() computes all four from the y aesthetic (the values to summarise), so they are only mapped directly to draw values that are already summarised. If these OHLC values are provided as plot aesthetics and stat_ohlc() is used with a coarser granule, they will be incorporated into the new summary at the coarser granularity (e.g. daily open-high-low-close prices can be drawn as weekly or monthly candlesticks)

Computed variables

These are calculated by stat_ohlc(), and available to ggplot2::after_stat():

open, high, low, close The four summary statistics, also placed on the positional aesthetics above.

direction Whether the candlestick closed above ("rising"), below ("falling"), or level with ("unchanged") where it opened. Mapped to fill and colour by default.

change The close minus the open, i.e. direction as a signed magnitude rather than a three-level factor. Useful for a continuous colour or fill scale, e.g. aes(fill = after_stat(change)), in place of the default discrete direction mapping.

n The number of observations summarised.

width The width of the candlestick's body, in data units.

Aesthetics

geom_candlestick() understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- x ***or*** y
- lower ***or*** xlower
- upper ***or*** xupper
- ymin ***or*** xmin
- ymax ***or*** xmax
- alpha → NA
- colour → via theme()
- fill → via theme()

- group → inferred
- linetype → via theme()
- linewidth → via theme()

Learn more about setting these aesthetics in vignette("ggplot2-specs").

Examples

```
library(ggplot2)

prices <- data.frame(
  time = mixtime::datetime(
    as.POSIXct("2024-01-01", tz = "UTC") + (0:719) * 3600
  ),
  price = 100 + cumsum(rnorm(720, 0, 0.5))
)

# Summarise hourly prices into daily candlesticks, coloured red/green by
# `direction` by default
ggplot(prices, aes(time, price)) +
  geom_candlestick(granule = mixtime::days(1L))

# `granule` accepts a bare granule of the axis's own calendar
ggplot(prices, aes(time, price)) +
  geom_candlestick(granule = hour(6L))

# Any other discrete diverging scale overrides the red/green default;
# `colour` needs its own scale too, or it stays red/green
ggplot(prices, aes(time, price)) +
  geom_candlestick(granule = mixtime::days(1L)) +
  scale_fill_brewer(type = "div", palette = "RdBu", drop = FALSE) +
  scale_colour_brewer(type = "div", palette = "RdBu", drop = FALSE)

# Use after_stat(change) with a diverging colour scale for extent of change
ggplot(prices, aes(time, price)) +
  geom_candlestick(
    aes(fill = after_stat(change), colour = after_stat(change)),
    granule = mixtime::days(1L)
  ) +
  scale_fill_gradient2(low = "red", mid = "grey90", high = "green") +
  scale_colour_gradient2(low = "red", mid = "grey90", high = "green")

# Already summarised prices can be plotted with stat = "identity"
set.seed(0)
close <- 100 + cumsum(rnorm(90))
ohlc <- data.frame(
  time = mixtime::date(as.Date("2024-01-01") + 0:89),
  open = c(100, close[-length(close)]),
  close = close
)
ohlc$high <- pmax(ohlc$open, ohlc$close) + runif(90, 0, 2)
```

```
ohlc$low <- pmin(ohlc$open, ohlc$close) - runif(90, 0, 2)
ohlc$direction <- ifelse(sign(ohlc$close - ohlc$open) > 0, "rising", "falling")
ggplot(
  ohlc,
  aes(
    x = time, fill = direction, colour = direction,
    lower = open, upper = close,
    ymin = low, ymax = high
  )
) +
  geom_candlestick(stat = "identity")

# Using stat = "ohlc" (the default) with complete OHLC data will
# compute the OHLC prices at coarser granules
ggplot(
  ohlc,
  aes(
    x = time, lower = open, upper = close,
    ymin = low, ymax = high
  )
) +
  geom_candlestick(granule = mixtime::cal_isoweek$week(1L))
```

---

geom_time_line *Line geometry with temporal semantics*

Description

geom_time_line() connects observations in order of the time variable, similar to ggplot2::geom_line(), but with special handling for time zones, gaps and duplicated values.

The geometry helps to visualise time with changing time offsets provided by the [x/y]timeoffset aesthetics. Changes in time offsets are drawn using dashed lines, which are most commonly used for timezone changes and daylight savings time transitions. Timezone offsets are automatically used when times from the mixtime package are plotted in local time, which is the scale's default behaviour (see the time_chronon argument of scale_x_mixtime()).

This geometry also respects implicit missing values in regular time series, and will not connect temporal observations separated by gaps.

The ggplot2::group aesthetic determines which cases are connected together.

Usage

```
geom_time_line(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  na.rm = FALSE,
```

```
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE,
  transitions = waiver(),
  transition_aesthetics = list(linetype = 2),
  ...
)
```

Arguments

`mapping` Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data` The data to be displayed in this layer. There are three options:

If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.

A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat` The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`orientation` Which positional axis (`"x"` or `"y"`) carries the time variable that observations are connected in order of. The default (`NA`) determines this automatically: whichever

of x/y is time-valued (a mixtime or POSIXct), preferring x if both (or neither) are.

show.legend
: logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.

inherit.aes
: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. annotation_borders().

transitions
: A data.frame of known time offset transitions, shaped like mixtime::tz_transitions()'s own output (time a time point, and offset_before/offset_after mixtime::duration()s), with an optional id column to scope rows to a specific series (matched against the xtimeid aesthetic; applied to every series when omitted). Defaults to waiver(), which automatically calls mixtime::tz_transitions() for every timezone present in the data. See the "Time transitions" section below.

transition_aesthetics
: A named list of aesthetics for the segment drawn for time transitions, specifying how it is styled differently to the rest of the line. Defaults to list(linetype = 2), drawing transitions with dashed lines. Valid aesthetics are colour/color, linewidth, linetype and alpha.

...
: Other arguments passed on to layer()'s params argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the position argument, or aesthetics that are required can *not* be passed through .... Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, colour = "red" or linewidth = 3. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a stat_*() function, the ... argument can be used to pass on parameters to the geom part of the layer. An example of this is stat_density(geom = "area", outline.type = "both"). The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a geom_*() function, the ... argument can be used to pass on parameters to the stat part of the layer. An example of this is geom_area(stat = "density", adjust = 0.5). The stat's documentation lists which parameters it can accept.
- The key_glyph argument of layer() may also be passed on through .... This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

Practical usage

The `geom_time_line()` geometry extends `ggplot2::geom_line()` with time semantics that ensure the line's slope accurately reflects rates of change in the measurements over time.

Most notably, `geom_time_line()` works closely with the time scale (`scale_x_mixtime()`) to correctly display time in local and absolute time formats. Local time (the scale's default whenever all time points share a timezone) shows time as experienced in that timezone, it is the time on clocks in that timezone. Absolute time shows time as a continuous timeline without timezone adjustments. Which of these is shown is controlled by the scale's `time_chronon`: a chronon with `tz = NA` combines time points by their local wall-clock reading (local time), while a chronon with a fixed timezone (such as UTC, the default when a common chronon must be identified across timezones) aligns them by the instant they occurred (absolute time).

When time series are visualised in local time, timezone offset changes (e.g. due to daylight saving time) cause 'jumps' in time which are indicated with dashed lines. This preserves the integrity of the line's slope across these transitions. Another benefit of visualising time series in local time is to compare time series across different timezones, as the time axis is better aligned with human behaviour in their local timezone (e.g. working hours, sleep patterns, etc). Plotting time series in *absolute time* shows the exact contemporaneous timing of events across multiple timezones, which is useful when resources or patterns are shared across timezones (e.g. international markets, server load balancing, etc).

This geometry also maintains semantically valid slopes when time values are missing (either implicitly or explicitly), or duplicated. Implicit missing values in regular time series are semantically equivalent to explicit missing values, and `geom_time_line()` since the slope between unkown values is also unknown, `geom_time_line()` will not draw lines connecting missing values of either type. Since duplicated time values are not semantically valid in regular time series, `geom_time_line()` will issue a warning (or an error if systematic duplicates are detected). When drawing a line between duplicated time points, the correct slopes are drawn by connecting all lines that lead to and from the duplicated time points (rather than drawing sawtooth lines).

Further details about each specific capability are described in the following sections.

Time transitions

When time is displayed locally, daylight savings transitions introduce discontinuities in the local timeline when the clock jumps forwards or backwards. `geom_time_line()` draws these jumps as dashed segments, to preserve the integrity of the line's slope across the transition. When the time scale is set to use local time (see the `time_chronon` argument of `scale_x_mixtime()`), the default behaviour (`transitions = waiver()`) sources daylight savings transitions automatically with `mixtime::tz_transitions()`.

The appearance of transition segments is controlled with `transition_aesthetics`, a named list of aesthetic overrides (`colour`, `linewidth`, `linetype` and/or `alpha`). The default is a dashed line.

Offset changes aren't always timezone related. A sensor may be periodically synchronized to adjust for clock drift, or the `transitions` may reflect an individual's personal travel through time zones. The local time should be mapped to the `[x/y]` positional aesthetics, with the offset from absolute time mapped to `[x/y]timeoffset` (a `mixtime::duration()`). The `transitions` argument then specifies the instants at which the offset changes, and the offset before and after each transition. This is specified as a `data.frame` shaped like `mixtime::tz_transitions()`, with an optional `id` column to scope rows to a specific series (matched against the `[x/y]timeid` aesthetic).

Missing time values

Explicit missing values are where an NA value is included in the data, but for regular time series it is also possible to identify implicit missing time values. Unlike ggplot2::geom_line(), geom_time_line() will also not connect points separated by implicit missing values, creating gaps in the line (just like when an explicit missing value is present in ggplot2::geom_line()).

Duplicated time values

If there are duplicated time values within a group, geom_time_line() will issue a warning. An error will be raised if these duplications are systematic across the geometry, specifically if more than 50% of time points contain the same number of duplicates. Systematic duplicates typically indicate a need to use grouping aesthetics (ggplot2::group, or ggplot2::colour) to draw separate lines for each time series. Rather than plotting an erroneous 'sawtooth' line which misrepresents the rate of change, the geometry will draw all lines that connect to and from each of the duplicated time values.

Aesthetics

geom_time_line() understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- x
- y
- alpha → NA
- colour → via theme()
- group → inferred
- linetype → via theme()
- linewidth → via theme()
- xtimeid
- xtimeoffset
- ytimeid
- ytimeoffset

Learn more about setting these aesthetics in vignette("ggplot2-specs").

See Also

scale_mixtime for defining local and absolute time using time_chronon.

ggplot2::geom_line()/ggplot2::geom_path() for standard line/path geoms in ggplot2.

Examples

```
library(ggplot2)


# Basic time line plot of a random walk (no timezone changes)
df_ts <- data.frame(
  time = as.POSIXct("2023-03-11", tz = "Australia/Melbourne") + 0:11 * 3600,
```

```
  value = cumsum(rnorm(12, 2))
)
ggplot(df_ts, aes(time, value)) +
  geom_time_line()

# Random walk with a backward timezone change (DST ends)
df_tz_back <- data.frame(
  time = as.POSIXct("2023-04-02", tz = "Australia/Melbourne") + 0:11 * 3600,
  value = cumsum(rnorm(12, 2))
)
# Naive/local time (`tz = NA`) shows the DST transition as a dashed jump
ggplot(df_tz_back, aes(time, value)) +
  geom_time_line() +
  scale_x_mixtime(time_chronon = mixtime::cal_gregorian$hour(1L, tz = NA))
# Absolute time aligns to a single fixed timezone, removing the jump
ggplot(df_tz_back, aes(time, value)) +
  geom_time_line() +
  scale_x_mixtime(time_chronon = mixtime::cal_gregorian$hour(1L, tz = "UTC"))

# Random walk with a forward timezone change (DST starts)
df_tz_forward <- data.frame(
  time = as.POSIXct("2023-10-01", tz = "Australia/Melbourne") + 0:11 * 3600,
  value = cumsum(rnorm(12, 2))
)
ggplot(df_tz_forward, aes(time, value)) +
  geom_time_line() +
  scale_x_mixtime(time_chronon = mixtime::cal_gregorian$hour(1L, tz = NA))
ggplot(df_tz_forward, aes(time, value)) +
  geom_time_line() +
  scale_x_mixtime(time_chronon = mixtime::cal_gregorian$hour(1L, tz = "UTC"))
```

---

gg_arma *Plot characteristic ARMA roots*

Description

Produces a plot of the inverse AR and MA roots of an ARIMA model. Inverse roots outside the unit circle are shown in red.

Usage

```
gg_arma(data)
```

Arguments

data	A mable containing models with AR and/or MA roots.

Details

Only models which compute ARMA roots can be visualised with this function. That is to say, the `glance()` of the model contains `ar_roots` and `ma_roots`.

Value

A ggplot object the characteristic roots from ARMA components.

Examples

```
library(fable)
library(tsibble)
library(dplyr)

tsibbledata::aus_retail %>%
  filter(
    State == "Victoria",
    Industry == "Cafes, restaurants and catering services"
  ) %>%
  model(ARIMA(Turnover ~ pdq(0,1,1) + PDQ(0,1,1))) %>%
  gg_arma()
```

---

gg_irf *Plot impulse response functions*

Description

Produces a plot of impulse responses from an impulse response function.

Usage

```
gg_irf(data, y = all_of(measured_vars(data)))
```

Arguments

`data`	A tsibble with impulse responses
`y`	The impulse response variables to plot (defaults to all measured variables).

Value

A ggplot object of the impulse responses.

gg_lag *Lag plots*

Description

A lag plot shows the time series against lags of itself. It is often coloured the seasonal period to identify how each season correlates with others.

Usage

```
gg_lag(
  data,
  y = NULL,
  period = NULL,
  lags = 1:9,
  geom = c("path", "point"),
  arrow = FALSE,
  ...
)
```

Arguments

data	A tidy time series object (tsibble)
y	The variable to plot (a bare expression). If NULL, it will automatically selected from the data.
period	The seasonal period to display. If NULL (default), the largest frequency in the data is used. If numeric, it represents the frequency times the interval between observations. If a string, it is a whole number of a single unit (e.g., "year", "2 weeks", "1y" for 1 year, "3m" for 3 months, "1d" for 1 day, "1h" for 1 hour, "1min" for 1 minute, "1s" for 1 second). A mixtime duration (e.g. `mixtime::weeks(2L)`) or granule can also be used. Note that the data must have at least one observation per seasonal period, and the period cannot be smaller than the observation interval.
lags	A vector of lags to display as facets.
geom	The geometry used to display the data.
arrow	Arrow specification to show the direction in the lag path. If TRUE, an appropriate default arrow will be used. Alternatively, a user controllable arrow created with `grid::arrow()` can be used.
...	Additional arguments passed to the geom.

Value

A ggplot object showing a lag plot of a time series.

Examples

```
library(tsibble)
library(dplyr)
tsibbledata::aus_retail %>%
  filter(
    State == "Victoria",
    Industry == "Cafes, restaurants and catering services"
  ) %>%
  gg_lag(Turnover)
```

---

gg_season *Seasonal plot*

Description

Produces a time series seasonal plot. A seasonal plot is similar to a regular time series plot, except the x-axis shows data from within each season. This plot type allows the underlying seasonal pattern to be seen more clearly, and is especially useful in identifying years in which the pattern changes.

Usage

```
gg_season(
  data,
  y = NULL,
  period = NULL,
  facet_period = NULL,
  max_col = Inf,
  max_col_discrete = 7,
  pal = (scales::hue_pal())(9),
  polar = FALSE,
  labels = c("none", "left", "right", "both"),
  labels_repel = FALSE,
  labels_left_nudge = 0,
  labels_right_nudge = 0,
  ...
)
```

Arguments

data	A tidy time series object (tsibble)
y	The variable to plot (a bare expression). If NULL, it will automatically selected from the data.
period	The seasonal period to display. If NULL (default), the largest frequency in the data is used. If numeric, it represents the frequency times the interval between observations. If a string, it is a whole number of a single unit (e.g., "year", "2 weeks", "1y" for 1 year, "3m" for 3 months, "1d" for 1 day, "1h"

for 1 hour, "1min" for 1 minute, "1s" for 1 second). A mixtime duration (e.g. `mixtime::weeks(2L)`) or granule can also be used. Note that the data must have at least one observation per seasonal period, and the period cannot be smaller than the observation interval.

`facet_period` A secondary seasonal period to facet by (typically smaller than period).

`max_col` The maximum number of colours to display on the plot. If the number of seasonal periods in the data is larger than `max_col`, the plot will not include a colour. Use `max_col = 0` to never colour the lines, or Inf to always colour the lines. If labels are used, then max_col will be ignored.

`max_col_discrete` The maximum number of colours to show using a discrete colour scale.

`pal` A colour palette to be used.

`polar` If TRUE, the season plot will be shown on polar coordinates (looping `ggplot2::coord_radial()`).

`labels` Position of the labels for seasonal period identifier.

`labels_repel` If TRUE, the seasonal period identifying labels will be repelled with the ggrepel package.

`labels_left_nudge`, `labels_right_nudge` Allows seasonal period identifying labels to be nudged to the left or right from their default position.

`...` Additional arguments passed to geom_line()

Details

The seasonal plot is drawn with `coord_loop()`, which loops the continuous time axis around each seasonal period. Since time remains continuous, the line connecting one season to the next (e.g. December to January) is drawn too, split between the right and left margins of the plot.

Value

A ggplot object showing a seasonal plot of a time series.

References

Hyndman and Athanasopoulos (2019) Forecasting: principles and practice, 3rd edition, OTexts: Melbourne, Australia. https://OTexts.com/fpp3/

Examples

```
library(tsibble)
library(dplyr)
tsibbledata::aus_retail %>%
  filter(
    State == "Victoria",
    Industry == "Cafes, restaurants and catering services"
  ) %>%
  gg_season(Turnover)
```

gg_subseries *Seasonal subseries plots*

Description

A seasonal subseries plot facets the time series by each season in the seasonal period. These facets form smaller time series plots consisting of data only from that season. If you had several years of monthly data, the resulting plot would show a separate time series plot for each month. The first subseries plot would consist of only data from January. This case is given as an example below.

Usage

```
gg_subseries(data, y = NULL, period = NULL, ...)
```

Arguments

data	A tidy time series object (tsibble)
y	The variable to plot (a bare expression). If NULL, it will automatically selected from the data.
period	The seasonal period to facet by. If NULL (default), the smallest common seasonal period of the data is used (e.g. a week for daily data, a day for hourly data). Otherwise, a string or duration as described in gg_season().
...	Additional arguments passed to geom_line()

Details

The horizontal lines are used to represent the mean of each facet, allowing easy identification of seasonal differences between seasons. This plot is particularly useful in identifying changes in the seasonal pattern over time.

similar to a seasonal plot (gg_season()), and

Value

A ggplot object showing a seasonal subseries plot of a time series.

References

Hyndman and Athanasopoulos (2019) Forecasting: principles and practice, 3rd edition, OTexts: Melbourne, Australia. https://OTexts.com/fpp3/

Examples

```
library(tsibble)
library(dplyr)
tsibbledata::aus_retail %>%
  filter(
    State == "Victoria",
```

```
  Industry == "Cafes, restaurants and catering services"
) %>%
gg_subseries(Turnover)
```

---

gg_tsdisplay *Ensemble of time series displays*

Description

Plots a time series along with its ACF along with an customisable third graphic of either a PACF, histogram, lagged scatterplot or spectral density.

Usage

```
gg_tsdisplay(
  data,
  y = NULL,
  plot_type = c("auto", "partial", "season", "histogram", "scatter", "spectrum"),
  lag_max = NULL
)
```

Arguments

data	A tidy time series object (tsibble)
y	The variable to plot (a bare expression). If NULL, it will automatically selected from the data.
plot_type	type of plot to include in lower right corner. By default ("auto") a season plot will be shown for seasonal data, a spectrum plot will be shown for non-seasonal data without missing values, and a PACF will be shown otherwise.
lag_max	maximum lag at which to calculate the acf. Default is 10*log10(N/m) where N is the number of observations and m the number of series. Will be automatically limited to one less than the number of observations in the series.

Value

A list of ggplot objects showing useful plots of a time series.

Author(s)

Rob J Hyndman & Mitchell O'Hara-Wild

References

Hyndman and Athanasopoulos (2019) *Forecasting: principles and practice*, 3rd edition, OTexts: Melbourne, Australia. https://OTexts.com/fpp3/

See Also

plot.ts, feasts::ACF(), spec.ar

Examples

```
library(tsibble)
library(dplyr)
tsibbledata::aus_retail %>%
  filter(
    State == "Victoria",
    Industry == "Cafes, restaurants and catering services"
  ) %>%
  gg_tsdisplay(Turnover)
```

---

gg_tsresiduals *Ensemble of time series residual diagnostic plots*

Description

Plots the residuals using a time series plot, ACF and histogram.

Usage

```
gg_tsresiduals(data, type = "innovation", plot_type = "histogram", ...)
```

Arguments

data	A mable containing one model with residuals.
type	The type of residuals to compute. If type="response", residuals on the back-transformed data will be computed.
plot_type	type of plot to include in lower right corner. By default ("auto") a season plot will be shown for seasonal data, a spectrum plot will be shown for non-seasonal data without missing values, and a PACF will be shown otherwise.
...	Additional arguments passed to gg_tsdisplay().

Value

A list of ggplot objects showing a useful plots of a time series model's residuals.

References

Hyndman and Athanasopoulos (2019) *Forecasting: principles and practice*, 3rd edition, OTexts: Melbourne, Australia. https://OTexts.com/fpp3/

See Also

gg_tsdisplay()

Examples

```
library(fable)

tsibbledata::aus_production %>%
  model(ETS(Beer)) %>%
  gg_tsresiduals()
```

---

scale_candlestick *Default red/green scales for candlestick direction*

Description

geom_candlestick()'s default fill and colour scales for direction computed in stat_ohlc(). This default works only when the defaults are used alongside the direction statistic computed by stat_ohlc(), so candlesticks are red (falling) or green (rising) and can be overridden with an explicit colour/fill scale palette.

Usage

```
scale_fill_candlestick(
  ...,
  falling = "#D32F2F",
  unchanged = "grey70",
  rising = "#388E3C"
)

scale_colour_candlestick(
  ...,
  falling = "#D32F2F",
  unchanged = "grey70",
  rising = "#388E3C"
)
```

Arguments

... Passed on to ggplot2::scale_fill_manual() / ggplot2::scale_colour_manual().

falling, unchanged, rising
Colours for each level of direction.

Value

A ggproto scale object.

See Also

The *Rising and falling candlesticks* section of geom_candlestick().

scale_mixtime *Position scales for mixtime data*

Description

These are the default scales for mixtime vectors, responsible for mapping time points to aesthetics along with identifying break points and labels for the axes and guides. To override the scales behaviour manually, use `scale_*_mixtime`. The primary purpose of these scales is to scale time points across multiple granularities onto a common time scale. This is achieved by identifying and coercing all time points to the finest chronon that all time points can be represented in. This common time chronon is automatically identified, but can be manually specified using the `time_chronon` argument.

Usage

```
scale_x_mixtime(
  name = waiver(),
  breaks = waiver(),
  time_breaks = waiver(),
  minor_breaks = waiver(),
  time_minor_breaks = waiver(),
  labels = waiver(),
  time_labels = waiver(),
  time_chronon = waiver(),
  align_discrete = aes_nudge(),
  transform = "identity",
  limits = NULL,
  expand = waiver(),
  oob = scales::censor,
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver()
)

scale_y_mixtime(
  name = waiver(),
  breaks = waiver(),
  time_breaks = waiver(),
  minor_breaks = waiver(),
  time_minor_breaks = waiver(),
  labels = waiver(),
  time_labels = waiver(),
  time_chronon = waiver(),
  align_discrete = aes_nudge(),
  transform = "identity",
  limits = NULL,
  expand = waiver(),
```

```
  oob = scales::censor,
  guide = waiver(),
  position = "left",
  sec.axis = waiver()
)
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`breaks` One of:

- `NULL` for no breaks
- `waiver()` for the breaks specified by `date_breaks`
- A `Date`/`POSIXct` vector giving positions of breaks
- A function that takes the limits as input and returns breaks as output

`time_breaks` A duration giving the distance between breaks, such as `mixtime::weeks(2L)` or `mixtime::years(10L)`. If both `breaks` and `time_breaks` are specified, `time_breaks` wins. Each break is positioned within its granule by `align_discrete`, as data of that granule would be (e.g. monthly breaks fall mid-month, on monthly observations).

`minor_breaks` One of:

- `NULL` for no breaks
- `waiver()` for the breaks specified by `date_minor_breaks`
- A `Date`/`POSIXct` vector giving positions of minor breaks
- A function that takes the limits as input and returns minor breaks as output

`time_minor_breaks` A duration giving the distance between minor breaks, such as `mixtime::weeks(2L)` or `mixtime::years(10L)`. If both `minor_breaks` and `time_minor_breaks` are specified, `time_minor_breaks` wins.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `NULL` for no labels
- `waiver()` for the default labels computed by the transformation object
- A character vector giving labels (must be same length as `breaks`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`time_labels` A mixtime format string to format the labels, as described in `vignette("time-format-strings", package = "mixtime")`.

`time_chronon` A time granule that defines the common chronon to use for mixed granularity (e.g. `mixtime::tu_day(1L)`). The default automatically selects it as the finest chronon that all time points can be represented in.

align_discrete — Either a single number between 0 and 1, or a `aes_nudge()` object, defining how to align coarser granularities onto the common time scale.

If a single number is supplied, it is used for all positional aesthetics: 0 means start alignment, 1 means end alignment, and 0.5 means center alignment (the default).

To specify different offsets for different positional aesthetics (e.g. `x`, `xmin`, `xend`, `y`, `ymin`, ...), pass a `aes_nudge()` call, for example:

'align_discrete = aes_nudge(center = 0.5, left = 0.25, right = 0.75)"

The `center`, `left`, and `right` arguments apply to the semantically equivalent positional aesthetics (e.g. `left` applies to `xstart`, `xmin`, and `xlower`).

transform — A transformation applied to the time scale, after time points have been mapped onto the common time scale. Given as either a `<transform>` object or the name of one. Defaults to `"identity"`, applying no further transformation.

limits — One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum
- A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

expand — For position scales, a vector of range expansion constants used to add some padding around the data to ensure that they are placed some distance away from the axes. Use the convenience function `expansion()` to generate the values for the `expand` argument. The defaults are to expand the scale by 5% on each side for continuous variables, and by 0.6 units on each side for discrete variables.

oob — One of:

- Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation.
- The default (`scales::censor()`) replaces out of bounds values with `NA`.
- `scales::squish()` for squishing out of bounds values into range.
- `scales::squish_infinite()` for squishing infinite values into range.

guide — A function used to create a guide or its name. See `guides()` for more information.

position — For position scales, The position of the axis. `left` or `right` for y axes, `top` or `bottom` for x axes.

sec.axis — `sec_axis()` is used to specify a secondary axis.

Practical usage

When using `mixtime` vectors to represent time variables in ggplot2, these scales are automatically applied. In most cases, the default behaviour will be sufficient for scaling time points into plot aesthetics. These scales can be used to manually adjust the scaling behaviour, such as adjusting the breaks and labels or using a different common time scale.

Similarly to the temporal scales in ggplot2 (ggplot2::scale_x_date() and ggplot2::scale_x_datetime()), these scales can adjust the breaks and labels using duration-based intervals and time formatting. These time aware options are prefixed with time_ (e.g. time_breaks and time_labels), and take precedence over the non-time aware options (e.g. breaks and labels). The scale's breaks are specified with mixtime::duration() objects (e.g. time_breaks = mixtime::months(1L)) or a time granule.

Labels are specified with mixtime format strings, which describe a time point as glue-style {} placeholders holding the granules to show. Since the granules come from a calendar, this works across calendars rather than only Gregorian ones: time_labels = "{cyc(month, year, label = TRUE, abbreviate = TRUE)} {lin(year)}" gives "Jan 2020". See vignette("time-format-strings", package = "mixtime") for the full syntax.

A core feature of these scales is the ability to handle time from multiple timezones, granularities, and calendars. This is achieved by mapping all time points to a common time scale, which is automatically identifying the finest compatible chronon that can represent the input data. This allows time points across different granularities (e.g. base::POSIXt, base::Date, and mixtime::yearmonth) to be plotted together on a common time scale. In this case the finest chronon is 1 second (from base::POSIXt), so all time points are mapped to a 1 second chronon for plotting. Mapping day and month chronons to seconds introduces indeterminancy - which second should be used to represent a day or month? This is resolved using the align_discrete argument, which defaults to center alignment. This means that a day is mapped to noon, and a month is mapped to the middle of the month.

Further details about time specific scale options are described in the following sections.

Granularity alignment

Visualising mixed granularity time data introduces indeterminacy in the mapping of less precise time points onto a common time scale. For example, plotting monthly and daily data together raises the question of where to place the monthly points relative to the daily points. By default, mixtime uses center alignment, mapping the monthly points to the middle of the month. This is controlled using the align_discrete argument, which accepts a value between 0 (start alignment) and 1 (end alignment) and defaults to 0.5.

The common time scale that defines how all granularities are mapped is automatically identified based on the input data. This is achieved by finding the finest chronon that all time points can be represented in. For example, if the data contains both monthly and daily time points, the common time scale will be daily, with the monthly points aligned according to the align_discrete argument. If multiple time zones are present, the common time zone will default to UTC. The common time scale can be manually specified using the time_chronon argument, which accepts a mixtime::time_unit.

Examples

```
library(ggplot2)
library(dplyr)
uad_month <- tibble(
  time = mixtime::yearmonth("1973 Jan") + 0:71,
  value = USAccDeaths
)
uad_year <- uad_month %>%
```

```
    group_by(time = mixtime::year(time)) %>%
    summarise(value = mean(value), .groups = "drop")

  bind_rows(
    month = uad_month,
    year = uad_year,
    .id = "grain"
  ) %>%
    ggplot(aes(time, value, color = grain)) +
    geom_line() +
    scale_x_mixtime()
```

---

scale_mixtime_continuous

Non-positional scales for mixtime data

Description

Beyond `x` and `y`, mixtime vectors can also be mapped to `colour`, `fill`, `alpha`, `size`, and `linewidth` (e.g. to shade points or lines by when they occurred). These are the default scales used for mixtime vectors mapped to those aesthetics, and are automatically applied. To override the scale's behaviour manually, use `scale_*_mixtime`.

Usage

```
scale_colour_mixtime(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour"
)

scale_color_mixtime(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour"
)
```

```
scale_fill_mixtime(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "fill"
)

scale_alpha_mixtime(name = waiver(), ..., range = NULL, aesthetics = "alpha")

scale_size_mixtime(name = waiver(), ..., range = NULL, aesthetics = "size")

scale_linewidth_mixtime(
  name = waiver(),
  ...,
  range = NULL,
  aesthetics = "linewidth"
)
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`...` Arguments passed on to `continuous_scale`

`scale_name` **[Deprecated]** The name of the scale that should be used for error messages associated with this scale.

`breaks` One of:

- `NULL` for no breaks
- `waiver()` for the default breaks computed by the transformation object
- A numeric vector of positions
- A function that takes the limits as input and returns breaks as output (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.

`minor_breaks` One of:

- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`n.breaks` An integer guiding the number of major breaks. The algorithm may choose a slightly different number to ensure nice break labels. Will only have an effect if `breaks = waiver()`. Use `NULL` to use the default number of breaks given by the transformation.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `NULL` for no labels
- `waiver()` for the default labels computed by the transformation object
- A character vector giving labels (must be same length as `breaks`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum
- A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`rescaler` A function used to scale the input values to the range [0, 1]. This is always `scales::rescale()`, except for diverging and n colour gradients (i.e., `scale_colour_gradient2()`, `scale_colour_gradientn()`). The `rescaler` is ignored by position scales, which always use `scales::rescale()`. Also accepts rlang lambda function notation.

`oob` One of:

- Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation.
- The default (`scales::censor()`) replaces out of bounds values with `NA`.
- `scales::squish()` for squishing out of bounds values into range.
- `scales::squish_infinite()` for squishing infinite values into range.

`trans` **[Deprecated]** Deprecated in favour of `transform`.

`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`low, high` Colours for low and high ends of the gradient.

`space` colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.

`na.value` Colour to use for missing values

`guide` A function used to create a guide or its name. See `guides()` for more information.

aesthetics — The names of the aesthetics that this scale should be applied to, allowing this to be used with a different aesthetic (e.g. a custom aesthetic tied to `colour` or `fill` by `ggplot2::aes()`/`ggplot2::register_theme_elements()`).

range — Output range for `size`, `alpha`, and `linewidth`, given as a numeric vector of length 2. If `NULL`, the aesthetic's default range is used.

Details

Unlike `scale_x_mixtime()` and `scale_y_mixtime()`, these scales don't draw a common time axis shared by multiple layers, so there is only a single aesthetic column to map. The common chronon (and, for `colour`/`fill`, the colour gradient) is still resolved the same way as the position scales, with mixed granularities coerced onto the finest chronon they can all be represented in and `time_breaks`/`time_labels` accepted for time-aware breaks and labelling.

Examples

```
library(ggplot2)
df <- data.frame(
  time = mixtime::yearmonth(600:611),
  value = as.numeric(USAccDeaths[1:12])
)

ggplot(df, aes(value, 1, colour = time)) +
  geom_point(size = 5) +
  scale_colour_mixtime()
```

---

transform_warp — *Warp a scale so that intervals between fixed points are equally spaced*

Description

Warping gives each interval between successive warp points the same width, however much of the scale it actually covers. Warp point *i* is placed at position *i*, and values in between are placed by linear interpolation with `stats::approx()`: a value one third of the way between two warp points is drawn one third of the way between their positions. Every interval is therefore exactly one unit wide, so wide intervals are compressed and narrow ones stretched.

Usage

```
transform_warp(warps)
```

Arguments

warps — A sorted vector of at least two points, giving the fixed points between which the scale is stretched or compressed. Should match the data being warped: a `mixtime` (or `Date`/`POSIXt`) vector for a time scale, or a `numeric` vector otherwise.

Details

Values outside the range of `warps` cannot be placed, and become `NA`. Because panels are drawn with their range expanded beyond the data, `warps` should extend past the data on both sides rather than merely cover it.

`warps` should be of a type compatible with the data being warped: a time vector to warp time, or a `numeric` vector to warp a numeric scale.

Value

A `<transform>` object, suitable for the `transform` argument of `ggplot2::scale_x_continuous()` or `scale_x_mixtime()`.

Warping time series

Warping is particularly useful for time series, where the intervals of a granularity are often unequal: calendar months span 28 to 31 days, so a daily series drawn on a linear axis gives February less width than March. Warping at month boundaries removes that unevenness, making months comparable at a glance and putting each month's gridlines at a regular spacing.

Warp points need not share the data's granularity: they are converted to the data's chronon before being compared with it, so monthly warp points can place daily observations.

Warping does change the granularity of the scale to that of `warps`, with time points becoming continuous positions *within* that chronon rather than whole units of it. Breaks and labels follow suit, so a monthly warp labels its axis in months: a day in mid January is month `612.5`, which mixtime prints as `2021 Jan 50.0%`. Fractions track the real calendar, so `613.5` is the midpoint of 28 day February and `614.5` the midpoint of 31 day March.

Examples

```
library(ggplot2)

# Warp points need not be evenly spaced. A straight line makes the effect
# obvious: it kinks at x = 50, where the intervals change from 25 wide to 50
# wide, halving the slope from there on.
ggplot(data.frame(x = 10:90, y = 10:90), aes(x, y)) +
  geom_line() +
  scale_x_continuous(transform = transform_warp(c(0, 25, 50, 100)))

# Daily pedestrian counts for the first quarter of 2021: busy on weekdays,
# much quieter at the weekend, drifting upwards over the quarter.
pedestrians <- data.frame(
  date = mixtime::date("2021-01-01") + 0:89,
  count = round(
    ifelse(seq_along(date) %% 7 %in% c(2, 3), 4500, 12000) +
      cumsum(rnorm(length(date), 15, 150)) +
      rnorm(length(date), 0, 700)
  )
)

# Warp points extend a month either side of the data, because panels are drawn
```

```
# with their range expanded beyond it. They are monthly while the data is
# daily, which is fine: warp points are converted to the data's granularity.
month_starts <- mixtime::yearmonth("2020 Dec") + 0:5

# Without warping, the weekly cycle is evenly spaced but the month gridlines
# are not: 28 day February is drawn narrower than its 31 day neighbours.
ggplot(pedestrians, aes(date, count)) +
  geom_line() +
  scale_x_mixtime(breaks = month_starts + 0)

# Warping at the start of each month evens out the gridlines, but the length
# of each day is adjusted: the 28 days of February and 31 days of January and
# March are stretched or compressed to the same width over the month.
ggplot(pedestrians, aes(date, count)) +
  geom_line() +
  scale_x_mixtime(
    breaks = month_starts,
    # + 0 indicates the start of each month (continuous time model)
    transform = transform_warp(month_starts + 0)
  )
```

Index

aes(), *14*, *20*
aes_nudge, 2
alpha, *17*, *23*
annotation_borders(), *16*, *21*
autolayer.fbl_ts *(*autoplot.fbl_ts*)*, 5
autolayer.tbl_ts *(*autoplot.tbl_ts*)*, 6
autoplot.dcmp_ts, 4
autoplot.fbl_ts, 5
autoplot.tbl_cf, 6
autoplot.tbl_ts, 6

base::Date, *36*
base::POSIXt, *36*

colour, *17*, *23*
continuous_scale, *38*
coord_calendar, 7
coord_cartesian(), *35*, *39*
coord_loop, 10
coord_loop(), *7*, *9*, *28*

expansion(), *35*

feasts::ACF(), *6*, *31*
feasts::CCF(), *6*
feasts::PACF(), *6*
fill, *17*
fortify(), *15*, *20*

geom_candlestick, 13
geom_candlestick(), *32*
geom_time_line, 19
gg_arma, 24
gg_irf, 25
gg_lag, 26
gg_season, 27
gg_season(), *29*
gg_subseries, 29
gg_tsdisplay, 30
gg_tsdisplay(), *31*
gg_tsresiduals, 31
ggplot(), *15*, *20*
ggplot2::aes(), *40*
ggplot2::after_stat(), *17*
ggplot2::colour, *23*
ggplot2::coord_fixed(), *8*
ggplot2::coord_radial(), *28*
ggplot2::flip_data(), *17*
ggplot2::geom_boxplot(), *17*
ggplot2::geom_line(), *4*, *7*, *19*, *22*, *23*
ggplot2::geom_path(), *23*
ggplot2::group, *19*, *23*
ggplot2::register_theme_elements(), *40*
ggplot2::scale_colour_manual(), *32*
ggplot2::scale_fill_manual(), *32*
ggplot2::scale_x_continuous(), *41*
ggplot2::scale_x_date(), *36*
ggplot2::scale_x_datetime(), *36*
ggplot2::vars(), *7*
grid::arrow(), *26*
group, *18*, *23*
guides(), *35*, *39*

key glyphs, *16*, *21*

lambda, *34*, *35*, *38*, *39*
layer geom, *16*
layer position, *15*, *20*
layer stat, *15*, *20*
layer(), *15*, *16*, *21*
linetype, *18*, *23*
linewidth, *18*, *23*

mixtime::duration(), *21*, *22*, *36*
mixtime::tz_transitions(), *21*, *22*
mixtime::yearmonth, *36*

plot.ts, *31*

scale_alpha_mixtime
*(*scale_mixtime_continuous*)*, 37
scale_candlestick, 32

scale_color_mixtime
(scale_mixtime_continuous), 37
scale_colour_candlestick
(scale_candlestick), 32
scale_colour_gradient2(), *39*
scale_colour_gradientn(), *39*
scale_colour_mixtime
(scale_mixtime_continuous), 37
scale_fill_candlestick
(scale_candlestick), 32
scale_fill_mixtime
(scale_mixtime_continuous), 37
scale_linewidth_mixtime
(scale_mixtime_continuous), 37
scale_mixtime, *23*, 33
scale_mixtime_continuous, 37
scale_size_mixtime
(scale_mixtime_continuous), 37
scale_x_mixtime (scale_mixtime), 33
scale_x_mixtime(), *3*, *9*, *12*, *13*, *19*, *22*, *40*, *41*
scale_y_mixtime (scale_mixtime), 33
scale_y_mixtime(), *40*
scales::censor(), *35*, *39*
scales::extended_breaks(), *38*
scales::rescale(), *39*
scales::squish(), *35*, *39*
scales::squish_infinite(), *35*, *39*
sec_axis(), *35*
spec.ar, *31*
stat_ohlc (geom_candlestick), 13
stat_ohlc(), *32*
stats::approx(), *40*

transform_warp, 40
transformation object, *38*

x, *17*, *23*
xmax, *17*
xmin, *17*

y, *17*, *23*
ymax, *17*
ymin, *17*