Package: ggtime (via r-universe)

May 24, 2026

Title Grammar of Graphics and Plot Helpers for Time Series Visualization

Version 0.2.0.9000

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, lubridate (>= 1.7.1), tidyr, vctrs, cli, vecvec, mixtime, S7

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

Remotes github::mitchelloharawild/mixtime

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

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

Config/pak/sysreqs libicu-dev

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

Date/Publication 2026-05-17 12:05:29 UTC

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

RemoteRef HEAD

RemoteSha eeb884977b72c9e1233bd02399a9ff54fa4d92aa

Contents

`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

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

coord_calendar *Calendar coordinates*

Description

The calendar coordinate system arranges time series data into a calendar-like layout, making it easier to see fine-grained temporal patterns over a long time span. It has similar semantics as the looped coordinate system (`coord_loop()`), however instead of overlaying looped data the calendar coordinate space arranges each loop into rows and columns like a calendar.

Usage

```
coord_calendar(
  rows = waiver(),
  time_rows = waiver(),
  cols = waiver(),
  time_cols = waiver(),
  time = "x",
  xlim = NULL,
  ylim = NULL,
  expand = FALSE,
  default = FALSE,
  clip = "on",
  clip_rows = "on",
  coord = coord_cartesian()
)
```

Arguments

rows	Layout the time scale into calendar rows, one of: • NULL or waiver() for no rows (the default) • A mixtime vector giving time points at which the time axis should layout into rows • A function that takes the limits as input and returns row layout points as output
time_rows	A duration giving the distance between calendar rows like "1 weeks", or "1 month". If both rows and time_rows are specified, time_rows wins.
cols, time_cols	Not yet supported.
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.
clip_rows	Should the drawing of each loop of the timescale be clipped to the breaks defined by time_rows? 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 long time spans with events that occur over short intervals (such as holidays).

It works by:

1. Dividing the time axis into segments based on the specified row (and column) periods
2. Translating each panel into the rows and columns of a calendar layout

The coordinate system requires R version 4.2.0 or higher due to its use of usage of clipping paths.

Practical usage

The calendar coordinate system arranges a cartesian coordinate system into a dense calendar-like layout. Calendar layouts are particularly useful for identifying specific dates or events that occur over short intervals in long series. For example, the daily pedestrian counts at Melbourne's Birrarung Marr park is nearby to several major sporting venues, and the calendar layout makes obvious the spikes in pedestrian activity that occur during annual sporting events (such as the Australian Open tennis tournament). Calendar layouts are also useful to identify the effect of holidays, especially when their dates change each year (such as Easter).

Similarly to `coord_loop()`, the calendar coordinate system draws geometries that cross the boundaries of calendar rows or columns. The justification of these geometries can be controlled with the `align_discrete` parameter of `scale_x_mixtime()` as described in `coord_loop()`.

The calendar coordinate system works well in conjunction with facetting to give more space between months and/or years of the calendar. When facetting, using `scales = "free_x"` is recommended to make each facet only include time periods appropriate for that panel.

Examples

```
library(ggplot2)

# A weekly calendar arrangement of pedestrian counts in Melbourne
# Notice the periods of high activity days for the Birrarung Marr sensor
# during the Australian Open tennis tournament in late January.
tsibble::pedestrian |>
  dplyr::filter(Date < "2015-02-01") |>
  ggplot(aes(x = Date_Time, y = Count, color = Sensor)) +
  geom_line() +
  coord_calendar(time_rows = "1 week") +
  scale_x_datetime(date_breaks = "1 day", date_labels = "%a") +
  theme(legend.position = "bottom")

# Monthly facets can be used to create a complete calendar for 2015.
tsibble::pedestrian |>
  dplyr::filter(lubridate::year(Date) == 2015) |>
  ggplot(aes(x = Date_Time, y = Count, color = Sensor)) +
  geom_line() +
  coord_calendar(time_rows = "1 week") +
  facet_wrap(
    vars(lubridate::month(Date, label = TRUE)),
    ncol = 4, scales = "free_x"
  ) +
  scale_x_datetime(date_breaks = "1 day", date_labels = "%a") +
  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",
  clip_loops = "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 like "2 weeks", or "10 years". 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.
clip_loops	Should the drawing of each loop of the timescale be clipped to the breaks defined by time_loops? 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 segments based on the specified loop period
2. Translating each segment to overlay on the first segment
3. Creating a visualization where multiple time periods are superimposed

The coordinate system requires R version 4.2.0 or higher due to its use of usage of clipping paths.

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. 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 = "1 week")`.

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.

Examples

```
library(ggplot2)
library(ggtime)

# Basic usage with US accidental deaths data
uad <- tsibble::as_tsibble(USAccDeaths)
# Requires mixtime, POSIXct, or Date time types
uad$index <- as.Date(uad$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_loop = "1 year")
```

`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 used in conjunction with `position_time_civil()` positioning (the default).

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 = "time_civil",
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE,
  ...
)
```

Arguments

`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.
`...`	Other arguments passed on to `ggplot2::geom_line()`.

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 `position_time_civil()` and `position_time_absolute()` to correctly display time in civil and absolute time formats, respectively. Civil time positioning (the default) shows time as experienced in a specific timezone (also known as 'local time', it is the time on clocks in that timezone). Absolute time positioning shows time as a continuous timeline without timezone adjustments.

When time series are visualised in civil 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 civil 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 semanti-

cally 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.

Changing time offsets

The `xtimeoffset` and `ytimeoffset` aesthetics allow for visualization of time offset changes, such as timezone transitions or daylight saving time changes. When successive time offsets differ, a dashed line segment is drawn to show the offset transition. These aesthetics are automatically set when using `position = position_time_civil()` (the default), however the offsets can also be set manually to show other types of time offsets. One example of when it is useful to set the offsets manually is when showing measurements from a sensor with a known time drift (e.g. a clock that runs fast or slow) that is re-calibrated at known times.

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()`
- `xtimeoffset`
- `ytimeoffset`

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

See Also

`position_time_civil()`/`position_time_absolute()` for civil and absolute time positioning.

`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))
)
ggplot(df_tz_back, aes(time, value)) +
  geom_time_line()
ggplot(df_tz_back, aes(time, value)) +
  geom_time_line(position = position_time_absolute())

# 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()
ggplot(df_tz_forward, aes(time, value)) +
 geom_time_line(position = position_time_absolute())
```

---

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

```
if (requireNamespace("fable", quietly = TRUE)) {
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 (e.g., "1y" for 1 year, "3m" for 3 months, "1d" for 1 day, "1h" for 1 hour, "1min" for 1 minute, "1s" for 1 second), it's converted to a Period class object from the lubridate package. 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 (e.g., "1y" for 1 year, "3m" for 3 months, "1d" for 1 day, "1h" for 1 hour, "1min" for 1 minute, "1s" for 1 second), it's converted to a Period class object from the lubridate package. 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.

`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()

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 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 (e.g., "1y" for 1 year, "3m" for 3 months, "1d" for 1 day, "1h" for 1 hour, "1min" for 1 minute, "1s" for 1 second), it's converted to a Period class object from the lubridate package. Note that the data must have at least one observation per seasonal period, and the period cannot be smaller than the observation interval.
`...`	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

```
if (requireNamespace("fable", quietly = TRUE)) {
library(fable)

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

`position_time` *Position adjustments for timezones*

Description

Timezone-aware position adjustments preserve the vertical position of a geometry while adjusting its horizontal position display the time in either civil or absolute time. It requires the time variables to be mapped to either the x or y aesthetic to be represented in either a `base::POSIXt` or `mixtime::mixtime()` format.

Usage

```
position_time_civil()

position_time_absolute()
```

Details

These position adjustments handle time data with different timezone behaviors:

- `position_time_civil()` applies timezone offsets to position time to align times that would be experienced in each respective timezone.
- `position_time_absolute()` does not apply any timezone offsets, keeping time positioned in their exact relative timing across timezones.

Practical usage

Using timezone information to position time differently reveals different structures in the data. This is most evident when plotting multiple time series across different timezones.

Civil time (`position_time_civil()`) positions time as experienced by the observer in their timezone (also known as local time). It will align 9AM in Australia/Melbourne with 9AM in America/New_York, even though they occur at different absolute times. This is useful for comparing behavioural patterns that vary throughout times of the day, for example the amount of traffic during morning rush hour as people commute to work.

Absolute time (`position_time_absolute()`) positions time in a single reference timezone (usually UTC), reflecting the exact time when events happen. In absolute time, 9AM in Australia/Melbourne (AEST, UTC+10) will be aligned with 7PM in America/New_York (EST, UTC-5) of the previous day. This accurately reflects the equivalent timing of events across different timezones, which is useful for comparing events that happen simultaneously around the world, such as global financial market openings or international conference calls.

Examples

```
df_tz_mixed <- data.frame(
  time = mixtime::mixtime(
    as.POSIXct("2023-10-01", tz = "Australia/Melbourne") + 0:23 * 3600,
    as.POSIXct("2023-10-01", tz = "America/New_York") + 0:23 * 3600
  ),
  value = c(cumsum(rnorm(12, 2)), cumsum(rnorm(12, -2)))
)
# Civil time positioning aligns times in the same local timezone
#ggplot(df_tz_mixed, aes(time, value)) +
#  geom_time_line(position = position_time_civil())
# Absolute time positioning aligns times in a common timezone (e.g. UTC)
#ggplot(df_tz_mixed, aes(time, value)) +
#  geom_time_line(position = position_time_absolute())

# Positioning can also be used in other geoms
#ggplot(df_tz_mixed, aes(time, value)) +
#  geom_point(position = position_time_civil())
```

`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 `common_time`.

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(),
  warps = waiver(),
  time_warps = waiver(),
  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(),
  warps = waiver(),
  time_warps = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = scales::censor,
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver()
)
```

Arguments

time_breaks	A duration giving the distance between breaks like "2 weeks", or "10 years". If both breaks and time_breaks are specified, time_breaks wins.
time_minor_breaks	A duration giving the distance between minor breaks like "2 weeks", or "10 years". If both minor_breaks and time_minor_breaks are specified, time_minor_breaks

wins.

`time_labels` A mixtime format string to format the labels.

`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`).

`warps` Normalises the time scale to have a consistent length between specified time points, one of:

- `NULL` or `waiver()` for no warping (the default)
- A `mixtime` vector giving positions of warping points
- A function that takes the limits as input and returns warping points as output

`time_warps` A duration giving the distance between temporal warping like "2 weeks", or "10 years". If both `warps` and `time_warps` are specified, `time_warps` wins.

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 strftime-like 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 can be specified with `mixtime::duration()` objects (e.g. `time_breaks = mixtime::months(1L)`), or with strings that can be parsed into durations (e.g. `time_breaks = "1 month"`). Labels for time points in Gregorian calendars can be specified using `base::strftime()` formats (e.g. `time_labels = "%b %Y"` for "Jan 2020"). Concise strings for non-Gregorian calendars are not yet supported, but can be created using custom label functions (e.g. `labels = function(x) { ... }`).

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.

Another time specific feature of these scales is temporal warping. This adjusts the mapping of time points to plot aesthetics to have a consistent length between specified time points. This is most useful in comparing the shapes of cycles with irregular durations, including:

- astronomical cycles (e.g. the rising and setting of the sun)
- economic cycles (e.g. growth and recession phases of economies)
- predator-prey cycles (e.g. population dynamics of interacting species)
- biological cycles (e.g. hormonal cycles)
- calendar cycles (e.g. days in each 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 `common_time` argument, which accepts a `mixtime::time_unit`.

Temporal warping

Time scales can be warped to have a consistent length between specified time points. This is useful when visually exploring cyclical patterns where each cycle has varying length. By warping the time scale, the shape of each cycle can be more easily compared. Temporal warping is controlled using the `warps` argument, which accepts a `mixtime` vector defining the positions of the warping points. Calendar-based warping points can be conveniently specified using the `time_warps` argument, which accepts a duration like "1 month".

Examples

```
library(ggplot2)
library(dplyr)
uad_month <- tibble(
  time = mixtime::yearmonth(36L + 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()
```

Index