Package 'openair'

April 2, 2026

Type Package

Title Tools for the Analysis of Air Pollution Data

Version 3.0.0

Description Tools to analyse, interpret and understand air pollution data. Data are typically regular time series and air quality measurement, meteorological data and dispersion model output can be analysed. The package is described in Carslaw and Ropkins (2012, <doi:10.1016/j.envsoft.2011.09.008>) and subsequent papers.

License MIT + file LICENSE

URL https://openair-project.github.io/openair/, https://github.com/openair-project/openair

BugReports https://github.com/openair-project/openair/issues

Depends R (>= 4.1.0)

Imports cli, cluster, dplyr (>= 1.2), ggplot2 (>= 4.0.0), graphics, grDevices, grid, lubridate, MASS, methods, mgcv, patchwork, purrr (>= 1.0.0), Rcpp, readr, rlang, scales, stats, tidyr, utils

Suggests geomtextpath, KernSmooth, knitr, legendry (>= 0.2.4), quantreg, rmarkdown, rnaturalearthdata, sf, spelling, testthat (>= 3.0.0)

LinkingTo Rcpp

ByteCompile true

Config/Needs/website openair-project/openairpkgdown

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

LazyData yes

LazyLoad yes

RoxygenNote 7.3.3

NeedsCompilation yes

Author David Carslaw [aut, cre] (ORCID:
<https://orcid.org/0000-0003-0991-950X>),
Jack Davison [aut] (ORCID: <https://orcid.org/0000-0003-2653-6615>),
Karl Ropkins [aut] (ORCID: <https://orcid.org/0000-0002-0294-6997>)

Maintainer David Carslaw <david.carslaw@york.ac.uk>

Repository CRAN

Date/Publication 2026-04-02 09:00:02 UTC

Contents

`aqStats` *Calculate summary statistics for air pollution data by year*

Description

This function calculates a range of common and air pollution-specific statistics from a data frame. The statistics are calculated on an annual basis and the input is assumed to be hourly data. The function can cope with several sites and years, e.g., using `type = "site"`. The user can control the output by setting `transpose` appropriately. Note that the input data is assumed to be in mass units, e.g., ug/m3 for all species except CO (mg/m3).

Usage

```
aqStats(
  mydata,
  pollutant = "no2",
  type = "default",
  data.thresh = 0,
  percentile = c(95, 99),
  transpose = FALSE,
  progress = TRUE,
  ...
)
```

Arguments

`mydata`	A data frame containing a `date` field of hourly data.
`pollutant`	The name of a pollutant e.g. `pollutant = c("o3", "pm10")`. Additional statistics will be calculated if `pollutant %in% c("no2", "pm10", "o3")`.

type	type allows timeAverage() to be applied to cases where there are groups of data that need to be split and the function applied to each group. The most common example is data with multiple sites identified with a column representing site name e.g. type = "site". More generally, type should be used where the date repeats for a particular grouping variable. However, if type is not supplied the data will still be averaged but the grouping variables (character or factor) will be dropped.
data.thresh	The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as NA. See also interval, start.date and end.date to see whether it is advisable to set these other options.
percentile	Percentile values to calculate for each pollutant.
transpose	The default is to return a data frame with columns representing the statistics. If transpose = TRUE then the results have columns for each pollutant-type combination.
progress	Show a progress bar when many groups make up type? Defaults to TRUE.
...	Passed to cutData() for use with type.

Details

The following statistics are calculated:

For all pollutants:

- **data.capture** — percentage data capture over a full year.
- **mean** — annual mean.
- **minimum** — minimum hourly value.
- **maximum** — maximum hourly value.
- **median** — median value.
- **max.daily** — maximum daily mean.
- **max.rolling.8** — maximum 8-hour rolling mean.
- **max.rolling.24** — maximum 24-hour rolling mean.
- **percentile.95** — 95th percentile. Note that several percentiles can be calculated.

When pollutant == "o3":

- **roll.8.O3.gt.100** — number of days when the daily maximum rolling 8-hour mean ozone concentration is >100 ug/m3. This is the target value.
- **roll.8.O3.gt.120** — number of days when the daily maximum rolling 8-hour mean ozone concentration is >120 ug/m3. This is the Limit Value not to be exceeded > 10 days a year.
- **AOT40** — is the accumulated amount of ozone over the threshold value of 40 ppb for daylight hours in the growing season (April to September). Note that latitude and longitude can also be passed to this calculation.

When `pollutant == "no2"`:

- **hours** — number of hours NO2 is more than 200 ug/m3.

When `pollutant == "pm10"`:

- **days** — number of days PM10 is more than 50 ug/m3.

For the rolling means, the user can supply the option `align`, which can be "centre" (default), "left" or "right". See `rollingMean()` for more details.

There can be small discrepancies with the AURN due to the treatment of rounding data. The `aqStats()` function does not round, whereas AURN data can be rounded at several stages during the calculations.

Author(s)

David Carslaw

Examples

```
# Statistics for 2004. NOTE! these data are in ppb/ppm so the
# example is for illustrative purposes only
aqStats(selectByDate(mydata, year = 2004), pollutant = "no2")
```

---

binData *Bin data, calculate mean and bootstrap confidence interval in the mean*

Description

`binData()` summarises data by intervals and calculates the mean and bootstrap confidence intervals (by default 95% CI) in the mean of a chosen variable in a data frame. Any other numeric variables are summarised by their mean intervals. This occurs via `bootMeanDF()`, which calculates the uncertainty intervals in the mean of a vector.

Usage

```
binData(
  mydata,
  bin = "nox",
  uncer = "no2",
  type = "default",
  n = 40,
  interval = NA,
  breaks = NA,
  conf.int = 0.95,
  B = 250,
  ...
)

bootMeanDF(x, conf.int = 0.95, B = 1000)
```

Arguments

mydata	Name of the data frame to process.
bin	The name of the column to divide into intervals.
uncer	The name of the column for which the mean, lower and upper uncertainties should be calculated for each interval of bin.
type	Used for splitting the data further. Passed to cutData(). Note that intervals are calculated on the whole dataset before the data is categorised, meaning intervals will be the same for the different groups.
n	The number of intervals to split bin into.
interval	The interval to be used for binning the data.
breaks	User specified breaks to use for binning.
conf.int	The confidence interval, defaulting to 0.95 (i.e., the 95% Confidence Interval).
B	The number of bootstrap simulations.
...	Passed to cutData() for use with type.
x	A vector from which the mean and bootstrap confidence intervals in the mean are to be calculated

Details

There are three options for binning. The default is to bin bin into 40 intervals. Second, the user can choose an binning interval, e.g., interval = 5. Third, the user can supply their own breaks to use as binning intervals. Note that intervals are calculated on the whole dataset before the data is cut into categories using type.

Value

Returns a summarised data frame with new columns for the mean and upper / lower confidence intervals in the mean.

Examples

```
# work with vectors
test <- rnorm(20, mean = 10)
bootMeanDF(test)

# how does nox vary by intervals of wind speed?
results <- binData(mydata, bin = "ws", uncer = "nox")
## Not run:
library(ggplot2)
ggplot(results, aes(x = ws, y = mean, ymin = min, ymax = max)) +
  geom_pointrange()

## End(Not run)

# what about weekend vs weekday?
results2 <- binData(mydata, bin = "ws", uncer = "nox", type = "weekend")
## Not run:
```

```
ggplot(results2, aes(x = ws, y = mean, ymin = min, ymax = max)) +
  geom_pointrange() +
  facet_wrap(vars(weekend))

## End(Not run)
```

`calcPercentile` *Calculate percentile values from a time series*

Description

Calculates multiple percentile values from a time series, with flexible time aggregation. This function is a wrapper for `timeAverage()`, making it easier to calculate several percentiles at once. Like `timeAverage()`, it requires a data frame with a `date` field and one other numeric variable.

Usage

```
calcPercentile(
  mydata,
  pollutant = "o3",
  avg.time = "month",
  percentile = 50,
  type = "default",
  data.thresh = 0,
  start.date = NA,
  end.date = NA,
  prefix = "percentile."
)
```

Arguments

`mydata`	A data frame containing a `date` field . Can be class `POSIXct` or `Date`.
`pollutant`	Name of column containing variable to summarise, likely a pollutant (e.g., `"o3"`).
`avg.time`	This defines the time period to average to. Can be `"sec"`, `"min"`, `"hour"`, `"day"`, `"DSTday"`, `"week"`, `"month"`, `"quarter"` or `"year"`. For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be `avg.time = "2 month"`. In addition, `avg.time` can equal `"season"`, in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that `avg.time` can be *less* than the time interval of the original series, in which case the series is expanded to the new time interval. This is useful, for example, for calculating a 15-minute time series from an hourly one where an hourly value is repeated for each new 15-minute period. Note that when expanding data in this way it is necessary to ensure that the time interval of the original series is an exact multiple of `avg.time` e.g. hour to 10 minutes, day to hour. Also, the input time series must have consistent time gaps between

successive intervals so that timeAverage() can work out how much 'padding' to apply. To pad-out data in this way choose fill = TRUE.

percentile — A vector of percentile values; for example, percentile = 50 will calculate median values. Multiple values may also be provided as a vector, e.g., percentile = c(5, 50, 95) or percentile = seq(0, 100, 10).

type — type allows timeAverage() to be applied to cases where there are groups of data that need to be split and the function applied to each group. The most common example is data with multiple sites identified with a column representing site name e.g. type = "site". More generally, type should be used where the date repeats for a particular grouping variable. However, if type is not supplied the data will still be averaged but the grouping variables (character or factor) will be dropped.

data.thresh — The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as NA. See also interval, start.date and end.date to see whether it is advisable to set these other options.

start.date — A string giving a start date to use. This is sometimes useful if a time series starts between obvious intervals. For example, for a 1-minute time series that starts 2009-11-29 12:07:00 that needs to be averaged up to 15-minute means, the intervals would be 2009-11-29 12:07:00, 2009-11-29 12:22:00, etc. Often, however, it is better to round down to a more obvious start point, e.g., 2009-11-29 12:00:00 such that the sequence is then 2009-11-29 12:00:00, 2009-11-29 12:15:00, and so on. start.date is therefore used to force this type of sequence. Note that this option does not truncate a time series if it already starts earlier than start.date; see selectByDate() for that functionality.

end.date — A string giving an end date to use. This is sometimes useful to make sure a time series extends to a known end point and is useful when data.thresh > 0 but the input time series does not extend up to the final full interval. For example, if a time series ends sometime in October but annual means are required with a data capture of >75 % then it is necessary to extend the time series up until the end of the year. Input in the format yyyy-mm-dd HH:MM. Note that this option does not truncate a time series if it already ends later than end.date; see selectByDate() for that functionality.

prefix — Each new column is named by appending a prefix to percentile. For example, the default "percentile." will name the new column as percentile.95 when percentile = 95.

Value

Returns a data.frame with a date column plus an additional column for each given percentile.

Author(s)

David Carslaw

See Also

timePlot(), timeAverage()

Examples

```
# 95th percentile monthly o3 concentrations
percentiles <- calcPercentile(mydata,
  pollutant = "o3",
  avg.time = "month", percentile = 95
)

head(percentiles)

# 5, 50, 95th percentile monthly o3 concentrations
## Not run:
percentiles <- calcPercentile(mydata,
  pollutant = "o3",
  avg.time = "month", percentile = c(5, 50, 95)
)

head(percentiles)

## End(Not run)
```

---

calendarPlot *Plot time series values in a conventional calendar format*

Description

This function will plot data by month laid out in a conventional calendar format. The main purpose is to help rapidly visualise potentially complex data in a familiar way. Users can also choose to show daily mean wind vectors if wind speed and direction are available.

Usage

```
calendarPlot(
  mydata,
  pollutant = "nox",
  year = NULL,
  month = NULL,
  type = "month",
  statistic = "mean",
  data.thresh = 0,
  percentile = NA,
  annotate = "date",
  windflow = NULL,
  cols = "heat",
  limits = NULL,
```

```
  lim = NULL,
  col.lim = c("grey30", "black"),
  col.na = "white",
  font.lim = c(1, 2),
  cex.lim = c(0.6, 0.9),
  cex.date = 0.6,
  digits = 0,
  labels = NULL,
  breaks = NULL,
  w.shift = 0,
  w.abbr.len = 1,
  remove.empty = TRUE,
  show.year = TRUE,
  key.title = paste(statistic, pollutant, sep = " "),
  key.position = "right",
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame of time series. Must include a date field and at least one variable to plot.
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox".
year	Year to plot e.g. year = 2003. If not supplied and mydata contains more than one year, the first year of the data will be automatically selected. Manually setting year to NULL will use all available years.
month	If only certain month are required. By default the function will plot an entire year even if months are missing. To only plot certain months use the month option where month is a numeric 1:12 e.g. month = c(1, 12) to only plot January and December.
type	type determines how the data are split, i.e., conditioned, and then plotted. Only one type can be used with this function, as one faceting 'direction' is reserved by the month of the year. If a single type is given, it will form the "rows" of the resulting grid. Alternatively, c(type, "month") can be used can be specified for type to be used as the "columns" instead. type = "year" is a special case for calendarPlot() and will automatically prevent a single year from being selected (unless specified using the year argument) and set show.year to FALSE.
statistic	Statistic passed to timeAverage(). Note that if statistic %in% c("max", "min") and annotate is "ws" or "wd", the hour corresponding to the maximum/minimum concentration of polluant is used to provide the associated ws or wd and not the maximum/minimum daily ws or wd.

`data.thresh` The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as `NA`. See also `interval`, `start.date` and `end.date` to see whether it is advisable to set these other options.

`percentile` The percentile level in percent used when `statistic = "percentile"` and when aggregating the data with `avg.time`. More than one percentile level is allowed for `type = "default"` e.g. `percentile = c(50, 95)`. Not used if `avg.time = "default"`.

`annotate` This option controls what appears on each day of the calendar. Can be:

- `"date"` — shows day of the month
- `"value"` — shows the daily mean value
- `"none"` — shows no label

`windflow` If `TRUE`, the vector-averaged wind speed and direction will be plotted using arrows. Alternatively, can be a list of arguments to control the appearance of the arrows (colour, linewidth, alpha value, etc.). See `windflowOpts()` for details.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`limits` Use this option to manually set the colour scale limits. This is useful in the case when there is a need for two or more plots and a consistent scale is needed on each. Set the limits to cover the maximum range of the data for all plots of interest. For example, if one plot had data covering 0–60 and another 0–100, then set `limits = c(0, 100)`. Note that data will be ignored if outside the limits range.

`lim` A threshold value to help differentiate values above and below `lim`. It is used when `annotate = "value"`. See next few options for control over the labels used.

`col.lim` For the annotation of concentration labels on each day. The first sets the colour of the text below `lim` and the second sets the colour of the text above `lim`.

`col.na` Colour to be used to show missing data.

`font.lim` For the annotation of concentration labels on each day. The first sets the font of the text below `lim` and the second sets the font of the text above `lim`. Note that font = 1 is normal text and font = 2 is bold text.

`cex.lim` For the annotation of concentration labels on each day. The first sets the size of the text below `lim` and the second sets the size of the text above `lim`.

`cex.date` The base size of the annotation text for the date.

`digits` The number of digits used to display concentration values when `annotate = "value"`.

`breaks, labels` If a categorical colour scale is required then `breaks` should be specified. These should be provided as a numeric vector, e.g., `breaks = c(0, 50, 100, 1000)`.

Users should set the maximum value of `breaks` to exceed the maximum data value to ensure it is within the maximum final range, e.g., 100–1000 in this case. Labels will automatically be generated, but can be customised by passing a character vector to `labels`, e.g., `labels = c("good", "bad", "very bad")`. In this example, `0 - 50` will be `"good"` and so on. Note there is one less label than break.

`w.shift` Controls the order of the days of the week. By default the plot shows Saturday first (`w.shift = 0`). To change this so that it starts on a Monday for example, set `w.shift = 2`, and so on.

`w.abbr.len` The default (`1`) abbreviates the days of the week to a single letter (e.g., in English, S/S/M/T/W/T/F). `w.abbr.len` defines the number of letters to abbreviate until. For example, `w.abbr.len = 3` will abbreviate "Monday" to "Mon".

`remove.empty` Should months with no data present be removed? Default is `TRUE`.

`show.year` If only a single year is being plotted, should the calendar labels include the year label? `TRUE` creates labels like "January-2000", `FALSE` labels just as "January". If multiple years of data are detected, this option is forced to be `TRUE`.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each tile.

Details

calendarPlot() will plot data in a conventional calendar format, i.e., by month and day of the week. Daily statistics are calculated using timeAverage(), which by default will calculate the daily mean concentration.

If wind direction is available it is then possible to plot the wind direction vector on each day. This is very useful for getting a feel for the meteorological conditions that affect pollutant concentrations. Note that if hourly or higher time resolution are supplied, then calendarPlot() will calculate daily averages using timeAverage(), which ensures that wind directions are vector-averaged.

If wind speed is also available, then setting the option annotate = "ws" will plot the wind vectors whose length is scaled to the wind speed. Thus information on the daily mean wind speed and direction are available.

It is also possible to plot categorical scales. This is useful where, for example, an air quality index defines concentrations as bands, e.g., "good", "poor". In these cases users must supply labels and corresponding breaks.

Note that is is possible to pre-calculate concentrations in some way before passing the data to calendarPlot(). For example rollingMean() could be used to calculate rolling 8-hour mean concentrations. The data can then be passed to calendarPlot() and statistic = "max" chosen, which will plot maximum daily 8-hour mean concentrations.

Value

an openair object

Author(s)

David Carslaw

See Also

Other time series and trend functions: TheilSen(), smoothTrend(), timePlot(), timeProp(), timeVariation()

Examples

```
# basic plot
calendarPlot(mydata, pollutant = "o3", year = 2003)

# show wind vectors
calendarPlot(mydata, pollutant = "o3", year = 2003, annotate = "wd")
## Not run:
# show wind vectors scaled by wind speed and different colours
calendarPlot(mydata,
  pollutant = "o3", year = 2003, annotate = "ws",
  cols = "heat"
)

# show only specific months with selectByDate
calendarPlot(selectByDate(mydata, month = c(3, 6, 10), year = 2003),
  pollutant = "o3", year = 2003, annotate = "ws", cols = "heat"
```

```
)

# categorical scale example
calendarPlot(mydata,
  pollutant = "no2", breaks = c(0, 50, 100, 150, 1000),
  labels = c("Very low", "Low", "High", "Very High"),
  cols = c("lightblue", "green", "yellow", "red"), statistic = "max"
)

# UK daily air quality index
pm10.breaks <- c(0, 17, 34, 50, 59, 67, 75, 84, 92, 100, 1000)
calendarPlot(
  mydata,
  "pm10",
  year = 1999,
  breaks = pm10.breaks,
  labels = c(1:10),
  cols = "daqi",
  statistic = "mean",
  key.title = "PM10 DAQI"
)

## End(Not run)
```

`conditionalEval` *Conditional quantile estimates with additional variables for model evaluation*

Description

This function enhances `conditionalQuantile()` by also considering how other variables vary over the same intervals. Conditional quantiles are very useful on their own for model evaluation, but provide no direct information on how other variables change at the same time. For example, a conditional quantile plot of ozone concentrations may show that low concentrations of ozone tend to be under-predicted. However, the cause of the under-prediction can be difficult to determine. However, by considering how well the model predicts other variables over the same intervals, more insight can be gained into the underlying reasons why model performance is poor.

Usage

```
conditionalEval(
  mydata,
  obs = "obs",
  mod = "mod",
  var.obs = "var.obs",
  var.mod = "var.mod",
  type = "default",
  bins = 31,
  statistic = "MB",
```

```
  cols = "YlOrRd",
  col.var = "Set1",
  var.names = NULL,
  auto.text = TRUE,
  plot = TRUE,
  ...
)
```

Arguments

mydata	A data frame containing the field obs and mod representing observed and modelled values.
obs	The name of the observations in mydata.
mod	The name of the predictions (modelled values) in mydata.
var.obs	Other variable observations for which statistics should be calculated. Can be more than length one e.g. var.obs = c("nox.obs", "ws.obs").
var.mod	Other variable predictions for which statistics should be calculated. Can be more than length one e.g. var.mod = c("nox.mod", "ws.mod").
type	Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of: • A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season. • The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4). • The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.
bins	Number of bins to be used in calculating the different quantile levels.
statistic	Statistic(s) to be plotted. Can be "MB", "NMB", "r", "COE", "MGE", "NMGE", "RMSE" and "FAC2". statistic can also be a variable name in the data frame or a date-based type (e.g. "season"), in which case the plot shows the proportions of that variable across the prediction intervals. A special case is "cluster".
cols	Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

col.var	Colours for the additional variables. See openColours for more details.
var.names	Variable names to be shown in the legend for var.obs and var.mod.
auto.text	Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().
plot	When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).
...	Other graphical parameters passed onto conditionalQuantile() and cutData().

Details

The conditionalEval function provides information on how other variables vary across the same intervals as shown on the conditional quantile plot. There are two types of variable that can be considered by setting the value of statistic. First, statistic can be another variable in the data frame. In this case the plot will show the different proportions of statistic across the range of predictions. For example statistic = "season" will show for each interval of mod the proportion of predictions that were spring, summer, autumn or winter. This is useful because if model performance is worse for example at high concentrations of mod then knowing that these tend to occur during a particular season etc. can be very helpful when trying to understand *why* a model fails. See cutData() for more details on the types of variable that can be statistic. Another example would be statistic = "ws" (if wind speed were available in the data frame), which would then split wind speed into four quantiles and plot the proportions of each.

Second, conditionalEval can simultaneously plot the model performance of other observed/predicted variable **pairs** according to different model evaluation statistics. These statistics derive from the modStats() function and include "MB", "NMB", "r", "COE", "MGE", "NMGE", "RMSE" and "FAC2". More than one statistic can be supplied e.g. statistic = c("NMB", "COE"). Bootstrap samples are taken from the corresponding values of other variables to be plotted and their statistics with 95\ intervals calculated. In this case, the model *performance* of other variables is shown across the same intervals of mod, rather than just the values of single variables. In this second case the model would need to provide observed/predicted pairs of other variables.

For example, a model may provide predictions of NOx and wind speed (for which there are also observations available). The conditionalEval function will show how well these other variables are predicted for the same intervals of the main variables assessed in the conditional quantile e.g. ozone. In this case, values are supplied to var.obs (observed values for other variables) and var.mod (modelled values for other variables). For example, to consider how well the model predicts NOx and wind speed var.obs = c("nox.obs", "ws.obs") and var.mod = c("nox.mod", "ws.mod") would be supplied (assuming nox.obs, nox.mod, ws.obs, ws.mod are present in the data frame). The analysis could show for example, when ozone concentrations are under-predicted, the model may also be shown to over-predict concentrations of NOx at the same time, or under-predict wind speeds. Such information can thus help identify the underlying causes of poor model performance.

A special case is statistic = "cluster". In this case a data frame is provided that contains the cluster calculated by trajCluster() and importTraj(). Note that statistic = "cluster" cannot be used together with the ordinary model evaluation statistics such as MB. The output will be a bar chart showing the proportion of each interval of mod by cluster number.

Author(s)

David Carslaw

References

Wilks, D. S., 2005. Statistical Methods in the Atmospheric Sciences, Volume 91, Second Edition (International Geophysics), 2nd Edition. Academic Press.

See Also

The `verification` package for comprehensive functions for forecast verification.

Other model evaluation functions: `TaylorDiagram()`, `conditionalQuantile()`, `modStats()`

`conditionalQuantile` *Conditional quantile estimates for model evaluation*

Description

Function to calculate conditional quantiles with flexible conditioning. The function is for use in model evaluation and more generally to help better understand forecast predictions and how well they agree with observations.

Usage

```
conditionalQuantile(
  mydata,
  obs = "obs",
  mod = "mod",
  type = "default",
  bins = 31,
  min.bin = c(10, 20),
  cols = "YlOrRd",
  key.columns = 2,
  key.position = "bottom",
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

`mydata`	A data frame containing the field `obs` and `mod` representing observed and modelled values.
`obs`	The name of the observations in `mydata`.

`mod` The name of the predictions (modelled values) in `mydata`.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`bins` Number of bins to be used in calculating the different quantile levels.

`min.bin` The minimum number of points required for the estimates of the 25/75th and 10/90th percentiles.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`key.columns` Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

... Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.

Details

Conditional quantiles are a very useful way of considering model performance against observations for continuous measurements (Wilks, 2005). The conditional quantile plot splits the data into evenly spaced bins. For each predicted value bin e.g. from 0 to 10~ppb the *corresponding* values of the observations are identified and the median, 25/75th and 10/90 percentile (quantile) calculated for that bin. The data are plotted to show how these values vary across all bins. For a time series of observations and predictions that agree precisely the median value of the predictions will equal that for the observations for each bin.

The conditional quantile plot differs from the quantile-quantile plot (Q-Q plot) that is often used to compare observations and predictions. A Q-Q~plot separately considers the distributions of observations and predictions, whereas the conditional quantile uses the corresponding observations for a particular interval in the predictions. Take as an example two time series, the first a series of real observations and the second a lagged time series of the same observations representing the predictions. These two time series will have identical (or very nearly identical) distributions (e.g. same median, minimum and maximum). A Q-Q plot would show a straight line showing perfect agreement, whereas the conditional quantile will not. This is because in any interval of the predictions the corresponding observations now have different values.

Plotting the data in this way shows how well predictions agree with observations and can help reveal many useful characteristics of how well model predictions agree with observations — across the full distribution of values. A single plot can therefore convey a considerable amount of information concerning model performance. The `conditionalQuantile` function in openair allows conditional quantiles to be considered in a flexible way e.g. by considering how they vary by season.

The function requires a data frame consisting of a column of observations and a column of predictions. The observations are split up into `bins` according to values of the predictions. The median prediction line together with the 25/75th and 10/90th quantile values are plotted together with a line showing a "perfect" model. Also shown is a histogram of predicted values (shaded grey) and a histogram of observed values (shown as a blue outline).

Far more insight can be gained into model performance through conditioning using `type`. For example, `type = "season"` will plot conditional quantiles by each season. `type` can also be a factor or character field e.g. representing different models used.

See Wilks (2005) for more details and the examples below.

Author(s)

David Carslaw

Jack Davison

References

Murphy, A. H., B.G. Brown and Y. Chen. (1989) Diagnostic Verification of Temperature Forecasts, Weather and Forecasting, Volume: 4, Issue: 4, Pages: 485-501.

Wilks, D. S., 2005. Statistical Methods in the Atmospheric Sciences, Volume 91, Second Edition (International Geophysics), 2nd Edition. Academic Press.

See Also

The `verification` package for comprehensive functions for forecast verification.

Other model evaluation functions: `TaylorDiagram()`, `conditionalEval()`, `modStats()`

Examples

```
# make some dummy prediction data based on 'nox'
mydata$mod <- mydata$nox * 1.1 + mydata$nox * runif(1:nrow(mydata))

# basic conditional quantile plot
# A "perfect" model is shown by the blue line
# predictions tend to be increasingly positively biased at high nox,
# shown by departure of median line from the blue one.
# The widening uncertainty bands with increasing NOx shows that
# hourly predictions are worse for higher NOx concentrations.
# Also, the red (median) line extends beyond the data (blue line),
# which shows in this case some predictions are much higher than
# the corresponding measurements. Note that the uncertainty bands
# do not extend as far as the median line because there is insufficient
# to calculate them
conditionalQuantile(mydata, obs = "nox", mod = "mod")

# can split by season to show seasonal performance (not very
# enlightening in this case - try some real data and it will be!)

## Not run:
conditionalQuantile(mydata, obs = "nox", mod = "mod", type = "season")

## End(Not run)
```

`corPlot` *Correlation matrices with conditioning*

Description

Function to to draw and visualise correlation matrices. The primary purpose is as a tool for exploratory data analysis. Hierarchical clustering is used to group similar variables.

Usage

```
corPlot(
  mydata,
  pollutants = NULL,
  type = "default",
  cluster = TRUE,
  method = "pearson",
  use = "pairwise.complete.obs",
  annotate = c("cor", "signif", "stars", "none"),
  dendrogram = FALSE,
  triangle = c("both", "upper", "lower"),
  diagonal = TRUE,
  cols = "default",
  r.thresh = 0.8,
  text.col = c("black", "black"),
  key.title = NULL,
  key.position = "none",
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

`mydata`	A data frame which should consist of some numeric columns.
`pollutants`	the names of data-series in `mydata` to be plotted by `corPlot`. The default option `NULL` and the alternative `"all"` use all available valid (numeric) data.
`type`	Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of: • A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season. • The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4). • The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`cluster` Should the data be ordered according to cluster analysis. If `TRUE` hierarchical clustering is applied to the correlation matrices using `hclust()` to group similar variables together. With many variables clustering can greatly assist interpretation.

`method` The correlation method to use. Can be `"pearson"`, `"spearman"` or `"kendall"`.

`use` How to handle missing values in the `cor` function. The default is `"pairwise.complete.obs"`. Care should be taken with the choice of how to handle missing data when considering pair-wise correlations.

`annotate` What to annotate each correlation tile with. One of:

- `"cor"`, the correlation coefficient to 2 decimal places.
- `"signif"`, an X marker if the correlation is significant.
- `"stars"`, standard significance stars.
- `"none"`, no annotation.

`dendrogram` Should a dendrogram be plotted? When `TRUE` a dendrogram is shown on the plot. Note that this will only work for `type = "default"`. Defaults to `FALSE`.

`triangle` Which 'triangles' of the correlation plot should be shown? Can be `"both"`, `"lower"` or `"upper"`. Defaults to `"both"`.

`diagonal` Should the 'diagonal' of the correlation plot be shown? The diagonal of a correlation matrix is axiomatically always 1 as it represents correlating a variable with itself. Defaults to `TRUE`.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`r.thresh` Values of greater than `r.thresh` will be shown in bold type. This helps to highlight high correlations.

`text.col` The colour of the text used to show the correlation values. The first value controls the colour of negative correlations and the second positive.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each ellipse.

Details

The `corPlot()` function plots correlation matrices. The implementation relies heavily on that shown in Sarkar (2007), with a few extensions.

Correlation matrices are a very effective way of understating relationships between many variables. The `corPlot()` shows the correlation coded in three ways: by shape (ellipses), colour and the numeric value. The ellipses can be thought of as visual representations of scatter plot. With a perfect positive correlation a line at 45 degrees positive slope is drawn. For zero correlation the shape becomes a circle. See examples below.

With many different variables it can be difficult to see relationships between variables, i.e., which variables tend to behave most like one another. For this reason hierarchical clustering is applied to the correlation matrices to group variables that are most similar to one another (if `cluster = TRUE`).

If clustering is chosen it is also possible to add a dendrogram using the option `dendrogram = TRUE`. Note that dendrogramscan only be plotted for `type = "default"` i.e. when there is only a single panel. The dendrogram can also be recovered from the plot object itself and plotted more clearly; see examples below.

It is also possible to use the `openair` type option to condition the data in many flexible ways, although this may become difficult to visualise with too many panels.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

Adapted from the approach taken by Sarkar (2007)

Examples

```
# basic plot
corPlot(mydata)

## Not run:
# plot by season
corPlot(mydata, type = "season")

# recover dendrogram when cluster = TRUE and plot it
res <- corPlot(mydata, plot = FALSE)
plot(res$clust)

# a more interesting are hydrocarbon measurements
hc <- importAURN(site = "my1", year = 2005, hc = TRUE)

# now it is possible to see the hydrocarbons that behave most
# similarly to one another
corPlot(hc)

## End(Not run)
```

---

cutData *Function to split data in different ways for conditioning*

Description

Utility function to split data frames up in various ways for conditioning plots. Widely used by many openair functions usually through the option type.

Usage

```
cutData(
  x,
  type = "default",
  names = NULL,
  suffix = NULL,
  hemisphere = "northern",
  n.levels = 4,
  start.day = 1,
  start.season = "spring",
  is.axis = FALSE,
  local.tz = NULL,
  latitude = 51,
  longitude = -0.5,
  drop = c("default", "empty", "outside", "none"),
  ...
)
```

Arguments

x	A data frame containing a field date.
type	A string giving the way in which the data frame should be split. Pre-defined values are: "default", "year", "hour", "month", "season", "weekday", "site", "weekend", "monthyear", "daylight", "dst" (daylight saving time). type can also be the name of a numeric or factor. If a numeric column name is supplied cutData() will split the data into four quantiles. Factors levels will be used to split the data without any adjustment.
names	By default, the columns created by cutData() are named after their type option. Specifying names defines other names for the columns, which map onto the type options in the same order they are given. The length of names should therefore be equal to the length of type.
suffix	If name is not specified, suffix will be appended to any added columns that would otherwise overwrite existing columns. For example, cutData(mydata, "nox", suffix = "_cuts") would append a nox_cuts column rather than overwriting nox.
hemisphere	Can be "northern" or "southern", used to split data into seasons.
n.levels	Number of quantiles to split numeric data into.
start.day	What day of the week should the type = "weekday" start on? The user can change the start day by supplying an integer between 0 and 6. Sunday = 0, Monday = 1, ... For example to start the weekday plots on a Saturday, choose start.day = 6.
start.season	What order should the season be. By default, the order is spring, summer, autumn, winter. start.season = "winter" would plot winter first.
is.axis	A logical (TRUE/FALSE), used to request shortened cut labels for axes.
local.tz	Used for identifying whether a date has daylight savings time (DST) applied or not. Examples include local.tz = "Europe/London", local.tz = "America/New_York", i.e., time zones that assume DST. https://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones shows time zones that should be valid for most systems. It is important that the original data are in GMT (UTC) or a fixed offset from GMT.
latitude, longitude	The decimal latitude and longitudes used when type = "daylight". Note that locations west of Greenwich have negative longitudes.
drop	How to handle empty factor levels. One of: • "default": Sensible defaults selected on a case-by-case basis for different type options. • "empty": Drop all empty factor levels. • "none": Retain all empty factor levels, where possible. For example, for type = "hour", all factor levels from 0 and 23 will be represented. • "outside": Retain empty factor levels within the range of the data. For example, for type = "hour" when the data only contains data for 1 AM and 5 AM, the factor levels, 1, 2, 3, 4 and 5 will be retained.

Some of these options only apply to certain type options. For example, for type = "year", "outside" is equivalent to "none" as there is no fixed range of years to use in the "none" case.

... All additional parameters are passed on to next function(s).

Details

This section give a brief description of each of the define levels of type. Note that all time dependent types require a column date.

- "default" does not split the data but will describe the levels as a date range in the format "day month year".
- "year" splits the data by each year.
- "month" splits the data by month of the year.
- "hour" splits the data by hour of the day.
- "monthyear" (or "yearmonth") splits the data by year and month. It differs from month in that a level is defined for each month of the data set. This is useful sometimes to show an ordered sequence of months if the data set starts half way through a year; rather than starting in January.
- "weekend" splits the data by weekday and weekend.
- "weekday" splits the data by day of the week - ordered to start Monday.
- "season" splits data up by season. In the northern hemisphere winter = December, January, February; spring = March, April, May etc. These definitions will change of hemisphere = "southern".
- "seasonyear" (or "yearseason") will split the data into year-season intervals, keeping the months of a season together. For example, December 2010 is considered as part of winter 2011 (with January and February 2011). This makes it easier to consider contiguous seasons. In contrast, type = "season" will just split the data into four seasons regardless of the year.
- "quarter" splits data up by quarter, where Q1 represents January, February, and March, Q2 represents April, May and June, and so on. While 'quarters' don't as elegantly reflect meteorology as seasons, they do fit more neatly into a single year and may better align with other relevant periods (e.g., data ratification calendars, or business/economic activity).
- "quarteryear" (or "yearquarter") will split the data into year-quarter intervals. This is perhaps easier to predict and interpret than yearseason which will assign December to the Winter *after* the year it is actually in.
- "daylight" splits the data relative to estimated sunrise and sunset to give either daylight or nighttime. The cut is made by cutDaylight but more conveniently accessed via cutData, e.g. cutData(mydata, type = "daylight", latitude = my.latitude, longitude = my.longitude). The daylight estimation, which is valid for dates between 1901 and 2099, is made using the measurement location, date, time and astronomical algorithms to estimate the relative positions of the Sun and the measurement location on the Earth's surface, and is based on NOAA methods. Measurement location should be set using latitude (+ to North; - to South) and longitude (+ to East; - to West).
- "dst" will split the data by hours that are in daylight saving time (DST) and hours that are not for appropriate time zones. The option also requires that the local time zone is given e.g.

`local.tz = "Europe/London"`, `local.tz = "America/New_York"`. Each of the two periods will be in *local time*. The main purpose of this option is to test whether there is a shift in the diurnal profile when DST and non-DST hours are compared. This option is particularly useful with the `timeVariation()` function. For example, close to the source of road vehicle emissions, "rush-hour" will tend to occur at the same *local time* throughout the year, e.g., 8 am and 5 pm. Therefore, comparing non-DST hours with DST hours will tend to show similar diurnal patterns (at least in the timing of the peaks, if not magnitude) when expressed in local time. By contrast a variable such as wind speed or temperature should show a clear shift when expressed in local time. In essence, this option when used with `timeVariation()` may help determine whether the variation in a pollutant is driven by man-made emissions or natural processes.

- `"wd"` splits the data by 8 wind sectors and requires a column `wd`: "NE", "E", "SE", "S", "SW", "W", "NW", "N".

Note that all the date-based types, e.g., `"month"`/`"year"` are derived from a column `date`. If a user already has a column with a name of one of the date-based types it will not be used.

Value

Returns the data frame, `x`, with columns appended as defined by `type` and `name`.

Author(s)

David Carslaw

Jack Davison

Karl Ropkins (`"daylight"` option)

Examples

```
# split data by day of the week
mydata <- cutData(mydata, type = "weekday")
names(mydata)
head(mydata)
```

`datePad` *Pad a time-series dataframe and optionally fill values by block*

Description

Expand a dataframe that contains a 'date' column to a regular sequence of timestamps between specified start and end dates. The function can operate in two modes:

- `fill = FALSE`: simply complete the sequence at the target interval.
- `fill = TRUE`: regularise the data at the native interval to create explicit blocks, then expand to the target interval and carry the block's values forward so that intra-block timestamps inherit the block's measured value (block-filling behaviour).

Usage

```
datePad(
  mydata,
  type = NULL,
  interval = NULL,
  start.date = NULL,
  end.date = NULL,
  fill = FALSE,
  print.int = FALSE,
  ...
)
```

Arguments

mydata	Data.frame or tibble containing at least a 'date' column (Date or POSIXt).
type	NULL or character vector of column names to group by.
interval	NULL or character string describing target interval (e.g. "1 min", "1 hour"). If NULL, the native interval is used.
start.date	Optional start date/time. If NULL, the group's minimum date is used.
end.date	Optional end date/time. If NULL, the group's maximum date is used.
fill	Logical; when TRUE performs block-based filling described above. When FALSE just completes the sequence leaving NA values.
print.int	Logical; when TRUE prints detected/selected interval messages.
...	Passed to cutData() for use with type.

Details

The function detects the native input interval automatically if interval is not supplied, supports grouping via type, and preserves timezones for POSIXt date columns.

Value

A dataframe expanded to the requested sequence with values filled according to 'fill'. The returned object preserves the 'date' column type and timezone (for POSIXt).

Examples

```
df <- mydata[-c(2, 4, 7), ] # Remove some rows to create gaps
datePad(df)
```

GaussianSmooth *Calculate rolling Gaussian smooth of pollutant values*

Description

This is a utility function designed to calculate rolling Gaussian smooth (kernel smoothing). The function will try and fill in missing time gaps to get a full time sequence but return a data frame with the same number of rows supplied.

Usage

```
GaussianSmooth(
  mydata,
  pollutant = "o3",
  sigma = 24L,
  type = "default",
  data.thresh = 0,
  new.name = NULL,
  date.pad = FALSE,
  ...
)
```

Arguments

mydata	A data frame containing a date field. mydata must contain a date field in Date or POSIXct format. The input time series must be regular, e.g., hourly, daily.
pollutant	The name of a pollutant, e.g., pollutant = "o3". More than one pollutant can be supplied as a vector, e.g., pollutant = c("o3", "nox").
sigma	The value of sigma to use in the Gaussian.
type	Used for splitting the data further. Passed to cutData().
data.thresh	The % data capture threshold. No values are calculated if data capture over the period of interest is less than this value.
new.name	The name given to the new column(s). If not supplied it will create a name based on the name of the pollutant and the averaging period used.
date.pad	Should missing dates be padded? Default is FALSE.
...	Passed to cutData() for use with type.

Details

The function provides centre-aligned smoothing out to 3 sigma, which captures 99.7% of the data.

Value

A tibble with two new columns for the Gaussian smooth value.

Author(s)

David Carslaw

Examples

```
# Gaussian smoother with sigma = 24
mydata <- GaussianSmooth(mydata,
  pollutant = "o3", sigma = 24, data.thresh = 75)
```

`importADMS` *CERC Atmospheric Dispersion Modelling System (ADMS) data import function(s) for openair*

Description

Function(s) to import various ADMS file types into openair. Currently handles ".met", ".bgd", ".mop" and ".pst" file structures. Uses `utils::read.csv()` to read in data, format for R and openair and apply some file structure testing.

Usage

```
importADMS(
  file = file.choose(),
  file.type = "unknown",
  drop.case = TRUE,
  drop.input.dates = TRUE,
  keep.units = TRUE,
  simplify.names = TRUE,
  test.file.structure = TRUE,
  drop.delim = TRUE,
  add.prefixes = TRUE,
  names = NULL,
  all = FALSE,
  ...
)
```

Arguments

`file`	The ADMS file to be imported. Default, `file.choose()` opens browser. Use of `utils::read.csv()` also allows this to be a readable text-mode connection or url (although these options are currently not fully tested).
`file.type`	Type of ADMS file to be imported. With default, "unknown", the import uses the file extension to identify the file type and, where recognised, uses this to identify the file structure and import method to be applied. Where file extension is not recognised the choice may be forced by setting `file.type` to one of the known `file.type` options: "bgd", "met", "mop" or "pst".

`drop.case` Option to convert all data names to lower case. Default, `TRUE`. Alternative, `FALSE`, returns data with name cases as defined in file.

`drop.input.dates`
Option to remove ADMS "hour", "day", and "year" data columns after generating openair "date" timeseries. Default, `TRUE`. Alternative, `FALSE`, returns both "date" and the associated ADMS data columns as part of openair data frame.

`keep.units` Option to retain ADMS data units. Default, `TRUE`, retains units (if recoverable) as character vector in data frame comment if defined in `file`. Alternative, `FALSE`, discards units. (NOTE: currently, only `.bgd` and `.pst` files assign units. So, this option is ignored when importing `.met` or `.mop` files.)

`simplify.names` Option to simplify data names in accordance with common `openair` practices. Default, `TRUE`. Alternative, `FALSE`, returns data with names as interpreted by standard R. (NOTE: Some ADMS file data names include symbols and structures that R does not allow as part of a name, so some renaming is automatic regardless of `simplify.names` setting. For example, brackets or symbols are removed from names or replaced with ".", and names in the form "1/x" may be returned as "X1.x" or "recip.x".)

`test.file.structure`
Option to test file structure before trying to import. Default, `TRUE`, tests for expected file structure and halts import operation if this is not found. Alternative, `FALSE`, attempts import regardless of structure.

`drop.delim` Option to remove delim columns from the data frame. ADMS .mop files include two columns, "INPUT_DATA:" and "PROCESSED_DATA:", to separate model input and output types. Default, `TRUE`, removes these. Alternative, `FALSE`, retains them as part of import. (Note: Option ignored when importing `.bgd`, `.met` or `.pst` files.)

`add.prefixes` Option to add prefixes to data names. ADMS .mop files include a number of input and process data types with shared names. Prefixes can be automatically added to these so individual data can be readily identified in the R/openair environment. Default, `TRUE`, adds "process." as a prefix to processed data. Other options include: `FALSE` which uses no prefixes and leave all name rationalisation to R, and character vectors which are treated as the required prefixes. If one vector is sent, this is treated as processed data prefix. If two (or more) vectors are sent, the first and second are treated as the input and processed data prefixes, respectively. For example, the argument (`add.prefixes="out"`) would add the "out" prefix to processed data names, while the argument (`add.prefixes=c("in","out")`) would add "in" and "out" prefixes to input and output data names, respectively. (Note: Option ignored when importing `.bgd`, `.met` or `.pst` files.)

`names` Option applied by `simplifyNamesADMS` when `simplify.names` is enabled. All names are simplified for the default setting, `NULL`.

`all` For .MOP files, return all variables or not. If `all = TRUE` a large number of processed variables are returned.

`...` Arguments passed on to `utils::read.csv`

`header` a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.

`sep` the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default for `read.table`) the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.

`quote` the set of quoting characters. To disable quoting altogether, use `quote = ""`. See `scan` for the behaviour on quotes embedded in quotes. Quoting is only considered for columns read as character, which is all of them unless `colClasses` is specified.

`dec` the character used in the file for decimal points.

`fill` logical. If `TRUE` then in case the rows have unequal length, blank fields are implicitly added. See 'Details'.

`comment.char` character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether.

Details

The `importADMS` function were developed to help import various ADMS file types into openair. In most cases the parent import function should work in default configuration, e.g. `mydata <- importADMS()`. The function currently recognises four file formats: `.bgd`, `.met`, `.mop` and `.pst`. Where other file extensions have been set but the file structure is known, the import call can be forced by, e.g, `mydata <- importADMS(file.type="bgd")`. Other options can be adjusted to provide fine control of the data structuring and renaming.

Value

In standard use `importADMS()` returns a data frame for use in openair. By comparison to the original file, the resulting data frame is modified as follows:

Time and date information will combined in a single column "date", formatted as a conventional timeseries (`as.POSIX*`). If `drop.input.dates` is enabled data series combined to generated the new "date" data series will also be removed.

If `simplify.names` is enabled common chemical names may be simplified, and some other parameters may be reset to openair standards (e.g. "ws", "wd" and "temp") according to operations defined in `simplifyNamesADMS`. A summary of simplification operations can be obtained using, e.g., the call `importADMS(simplify.names)`.

If `drop.case` is enabled all upper case characters in names will be converted to lower case.

If `keep.units` is enabled data units information may also be retained as part of the data frame comment if available.

With `.mop` files, input and processed data series names may also been modified on the basis of `drop.delim` and `add.prefixes` settings

Note

Times are assumed to be in GMT. Zero wind directions reset to 360 as part of `.mop` file import.

Author(s)

Karl Ropkins, David Carslaw and Matthew Williams (CERC).

See Also

Other import functions: importAURN(), importEurope(), importImperial(), importMeta(), importTraj(), importUKAQ()

Examples

```
##########
# example 1
##########
# To be confirmed

# all current simplify.names operations
importADMS(simplify.names)

# to see what simplify.names does to adms data series name PHI
new.name <- importADMS(simplify.names, names = "PHI")
new.name
```

---

importAURN *Import data from individual UK Air Pollution Networks*

Description

These functions act as wrappers for importUKAQ() to import air pollution data from a range of UK networks including the Automatic Urban and Rural Network (AURN), the individual England (AQE), Scotland (SAQN), Wales (WAQN) and Northern Ireland (NI) Networks, and many "locally managed" monitoring networks across England. While importUKAQ() allows for data to be imported more flexibly, including across multiple monitoring networks, these functions are provided for convenience and back-compatibility.

Usage

```
importAURN(
  site = "my1",
  year = 2009,
  data_type = "hourly",
  pollutant = "all",
  hc = FALSE,
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)

importAQE(
```

```
  site = "yk13",
  year = 2018,
  data_type = "hourly",
  pollutant = "all",
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)

importSAQN(
  site = "gla4",
  year = 2009,
  data_type = "hourly",
  pollutant = "all",
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)

importWAQN(
  site = "card",
  year = 2018,
  data_type = "hourly",
  pollutant = "all",
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)

importNI(
  site = "bel0",
  year = 2018,
  data_type = "hourly",
  pollutant = "all",
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
```

```
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)

importLocal(
  site = "ad1",
  year = 2018,
  data_type = "hourly",
  pollutant = "all",
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)
```

Arguments

site	Site code of the site to import, e.g., "my1" is Marylebone Road. Site codes can be discovered through the use of importMeta(). Several sites can be imported at once. For example, site = c("my1", "nott") imports both Marylebone Road and Nottingham.
year	Year(s) to import. To import a series of years use, e.g., 2000:2020. To import several specific years use year = c(2000, 2010, 2020).
data_type	The type of data to be returned, defaulting to "hourly" data. Alternative data types include: • "daily": Daily average data. • "monthly": Monthly average data with data capture information for the whole network. • "annual": Annual average data with data capture information for the whole network. • "15_min": 15-minute average SO2 concentrations. • "8_hour": 8-hour rolling mean concentrations for O3 and CO. • "24_hour": 24-hour rolling mean concentrations for particulates. • "daily_max_8": Maximum daily rolling 8-hour maximum for O3 and CO. • "daqi": Daily Air Quality Index (DAQI). See here for more details of how the index is defined. Note that this data_type is not available for locally managed monitoring networks.
pollutant	Pollutants to import. If omitted will import all pollutants from a site. To import only NOx and NO2 for example use pollutant = c("nox", "no2"). Pollutant names can be upper or lower case.
hc	Include hydrocarbon measurements in the imported data? Defaults to FALSE as most users will not be interested in using hydrocarbon data.

meta	Append metadata columns to data for each selected site? Defaults to FALSE. Columns are defined using meta_columns.
meta_columns	The specific columns to append when meta = TRUE. Defaults to site type, latitude and longitude. Can be any of "site_type", "latitude", "longitude", "zone", "agglomeration", and "local_authority" (as well as "provider" for locally managed data). See importMeta() for more complete information.
meteo	Append modelled meteorological data, if available? Defaults to TRUE, which will return wind speed (ws), wind direction (wd) and ambient temperature (air_temp). The variables are calculated from using the WRF model run by Ricardo Energy & Environment and are available for most but not all networks. Setting meteo = FALSE is useful if you have other meteorological data to use in preference, for example from the worldmet package.
ratified	Append qc column(s) to hourly data indicating whether each species was ratified (i.e., quality-checked)? Defaults to FALSE.
to_narrow	Return the data in a "narrow"/"long"/"tidy" format? By default the returned data is "wide" and has a column for each pollutant/variable. When to_narrow = TRUE the data are returned with a column identifying the pollutant name and a column containing the corresponding concentration/statistic. Defaults to FALSE.
verbose	Print messages to the console if hourly data cannot be imported? Default is FALSE. TRUE is useful for debugging as the specific year(s), site(s) and source(s) which cannot be imported will be returned.
progress	Show a progress bar when many sites/years are being imported? Defaults to TRUE.

Importing UK Air Pollution Data

This family of functions has been written to make it easy to import data from across several UK air quality networks. Ricardo have provided .RData files (R workspaces) of all individual sites and years, as well as up to date meta data. These files are updated on a daily basis. This approach requires a link to the Internet to work.

There are several advantages over the web portal approach where .csv files are downloaded.

- First, it is quick to select a range of sites, pollutants and periods (see examples below).
- Second, storing the data as .RData objects is very efficient as they are about four times smaller than .csv files — which means the data downloads quickly and saves bandwidth.
- Third, the function completely avoids any need for data manipulation or setting time formats, time zones etc. The function also has the advantage that the proper site name is imported and used in openair functions.

Users should take care if using data from both openair *and* web portals (for example, UK AIR). One key difference is that the data provided by openair is date *beginning*, whereas the web portal provides date *ending*. Hourly concentrations may therefore appear offset by an hour, for example.

The data are imported by stacking sites on top of one another and will have field names site, code (the site code) and pollutant.

By default, the function returns hourly average data. However, annual, monthly, daily and 15 minute data (for SO2) can be returned using the option data_type. Annual and monthly data provide whole network information including data capture statistics.

All units are expressed in mass terms for gaseous species (ug/m3 for NO, NO2, NOx (as NO2), SO2 and hydrocarbons; and mg/m3 for CO). PM10 concentrations are provided in gravimetric units of ug/m3 or scaled to be comparable with these units. Over the years a variety of instruments have been used to measure particulate matter and the technical issues of measuring PM10 are complex. In recent years the measurements rely on FDMS (Filter Dynamics Measurement System), which is able to measure the volatile component of PM. In cases where the FDMS system is in use there will be a separate volatile component recorded as 'v10' and non-volatile component 'nv10', which is already included in the absolute PM10 measurement. Prior to the use of FDMS the measurements used TEOM (Tapered Element Oscillating. Microbalance) and these concentrations have been multiplied by 1.3 to provide an estimate of the total mass including the volatile fraction.

Some sites report hourly and daily PM10 and / or PM2.5. When `data_type = "daily"` and there are both hourly and 'proper' daily measurements available, these will be returned as e.g. "pm2.5" and "gr_pm2.5"; the former corresponding to data based on original hourly measurements and the latter corresponding to daily gravimetric measurements.

The function returns modelled hourly values of wind speed (`ws`), wind direction (`wd`) and ambient temperature (`air_temp`) if available (generally from around 2010). These values are modelled using the WRF model operated by Ricardo.

The BAM (Beta-Attenuation Monitor) instruments that have been incorporated into the network throughout its history have been scaled by 1.3 if they have a heated inlet (to account for loss of volatile particles) and 0.83 if they do not have a heated inlet. The few TEOM instruments in the network after 2008 have been scaled using VCM (Volatile Correction Model) values to account for the loss of volatile particles. The object of all these scaling processes is to provide a reasonable degree of comparison between data sets and with the reference method and to produce a consistent data record over the operational period of the network, however there may be some discontinuity in the time series associated with instrument changes.

No corrections have been made to the PM2.5 data. The volatile component of FDMS PM2.5 (where available) is shown in the 'v2.5' column.

See Also

Other import functions: `importADMS()`, `importEurope()`, `importImperial()`, `importMeta()`, `importTraj()`, `importUKAQ()`

`importEurope` *Import air quality data from European database until February 2024*

Description

This function is a simplified version of the `saqgetr` package (see https://github.com/skgrange/saqgetr) for accessing European air quality data. As `saqgetr` was retired in February 2024, this function has also been retired, but can still access European air quality data up until that retirement date. Consider using the EEA Air Quality Download Service instead (https://eeadmz1-downloads-webapp.azurewebsites.net/).

Usage

```
importEurope(
  site = "debw118",
  year = 2018,
  tz = "UTC",
  meta = FALSE,
  to_narrow = FALSE,
  progress = TRUE
)
```

Arguments

site	The code of the site(s).
year	Year or years to import. To import a sequence of years from 1990 to 2000 use year = 1990:2000. To import several specific years use year = c(1990, 1995, 2000) for example.
tz	Not used
meta	Should meta data be returned? If TRUE the site type, latitude and longitude are returned.
to_narrow	By default the returned data has a column for each pollutant/variable. When to_narrow = TRUE the data are stacked into a narrow format with a column identifying the pollutant name.
progress	Show a progress bar when many sites/years are being imported? Defaults to TRUE.

Value

a tibble

See Also

Other import functions: importADMS(), importAURN(), importImperial(), importMeta(), importTraj(), importUKAQ()

Examples

```
# import data for Stuttgart Am Neckartor (S)
## Not run:
stuttgart <- importEurope("debw118", year = 2010:2019, meta = TRUE)

## End(Not run)
```

`importImperial` *Import data from Imperial College London networks*

Description

Function for importing hourly mean data from Imperial College London networks, formerly the King's College London networks. Files are imported from a remote server operated by Imperial College London that provides air quality data files as R data objects.

Usage

```
importImperial(
  site = "my1",
  year = 2009,
  pollutant = "all",
  meta = FALSE,
  meteo = FALSE,
  extra = FALSE,
  units = "mass",
  to_narrow = FALSE,
  progress = TRUE
)

importKCL(
  site = "my1",
  year = 2009,
  pollutant = "all",
  met = FALSE,
  units = "mass",
  extra = FALSE,
  meta = FALSE,
  to_narrow = FALSE,
  progress = TRUE
)
```

Arguments

`site`	Site code of the network site to import e.g. "my1" is Marylebone Road. Several sites can be imported with `site = c("my1", "kc1")` — to import Marylebone Road and North Kensignton for example.
`year`	Year(s) to import. To import a series of years use, e.g., `2000:2020`. To import several specific years use `year = c(2000, 2010, 2020)`.
`pollutant`	Pollutants to import. If omitted will import all pollutants from a site. To import only NOx and NO2 for example use `pollutant = c("nox", "no2")`. Pollutant names can be upper or lower case.

meta	Append metadata columns to data for each selected site? Defaults to FALSE. Columns are defined using meta_columns.
meteo, met	Should meteorological data be added to the import data? The default is FALSE. If TRUE wind speed (m/s), wind direction (degrees), solar radiation and rain amount are available. See details below.
extra	Defaults to FALSE. When TRUE, returns additional data.
units	By default the returned data frame expresses the units in mass terms (ug/m3 for NOx, NO2, O3, SO2; mg/m3 for CO). Use units = "volume" to use ppb etc. PM10_raw TEOM data are multiplied by 1.3 and PM2.5 have no correction applied. See details below concerning PM10 concentrations.
to_narrow	Return the data in a "narrow"/"long"/"tidy" format? By default the returned data is "wide" and has a column for each pollutant/variable. When to_narrow = TRUE the data are returned with a column identifying the pollutant name and a column containing the corresponding concentration/statistic. Defaults to FALSE.
progress	Show a progress bar when many sites/years are being imported? Defaults to TRUE.

Details

The importImperial() function has been written to make it easy to import data from the Imperial College London air pollution networks. Imperial have provided .RData files (R workspaces) of all individual sites and years for the Imperial networks. These files are updated on a weekly basis. This approach requires a link to the Internet to work.

There are several advantages over the web portal approach where .csv files are downloaded. First, it is quick to select a range of sites, pollutants and periods (see examples below). Second, storing the data as .RData objects is very efficient as they are about four times smaller than .csv files — which means the data downloads quickly and saves bandwidth. Third, the function completely avoids any need for data manipulation or setting time formats, time zones etc. Finally, it is easy to import many years of data beyond the current limit of about 64,000 lines. The final point makes it possible to download several long time series in one go. The function also has the advantage that the proper site name is imported and used in 'openair" functions.

The site codes and pollutant names can be upper or lower case. The function will issue a warning when data less than six months old is downloaded, which may not be ratified.

The data are imported by stacking sites on top of one another and will have field names date, site, code (the site code) and pollutant(s). Sometimes it is useful to have columns of site data. This can be done using the reshape() function — see examples below.

The situation for particle measurements is not straightforward given the variety of methods used to measure particle mass and changes in their use over time. The importImperial() function imports two measures of PM10 where available. PM10_raw are TEOM measurements with a 1.3 factor applied to take account of volatile losses. The PM10 data is a current best estimate of a gravimetric equivalent measure as described below. NOTE! many sites have several instruments that measure PM10 or PM2.5. In the case of FDMS measurements, these are given as separate site codes (see below). For example "MY1" will be TEOM with VCM applied and "MY7" is the FDMS data.

Where FDMS data are used the volatile and non-volatile components are separately reported i.e. v10 = volatile PM10, v2.5 = volatile PM2.5, nv10 = non-volatile PM10 and nv2.5 = non-volatile PM2.5. Therefore, PM10 = v10 + nv10 and PM2.5 = v2.5 + nv2.5.

For the assessment of the EU Limit Values, PM10 needs to be measured using the reference method or one shown to be equivalent to the reference method. Defra carried out extensive trials between 2004 and 2006 to establish which types of particulate analysers in use in the UK were equivalent. These trials found that measurements made using Partisol, FDMS, BAM and SM200 instruments were shown to be equivalent to the PM10 reference method. However, correction factors need to be applied to measurements from the SM200 and BAM instruments. Importantly, the TEOM was demonstrated as not being equivalent to the reference method due to the loss of volatile PM, even when the 1.3 correction factor was applied. The Volatile Correction Model (VCM) was developed for Defra at King's College to allow measurements of PM10 from TEOM instruments to be converted to reference equivalent; it uses the measurements of volatile PM made using nearby FDMS instruments to correct the measurements made by the TEOM. It passed the equivalence testing using the same methodology used in the Defra trials and is now the recommended method for correcting TEOM measurements (Defra, 2009). VCM correction of TEOM measurements can only be applied after 1st January 2004, when sufficiently widespread measurements of volatile PM became available. The 1.3 correction factor is now considered redundant for measurements of PM10 made after 1st January 2004. Further information on the VCM can be found at http://www.volatile-correction-model.info/.

All PM10 statistics on the LondonAir web site, including the bulletins and statistical tools (and in the RData objects downloaded using `importImperial()`), now report PM10 results as reference equivalent. For PM10 measurements made by BAM and SM200 analysers the applicable correction factors have been applied. For measurements from TEOM analysers the 1.3 factor has been applied up to 1st January 2004, then the VCM method has been used to convert to reference equivalent.

The meteorological data are meant to represent 'typical' conditions in London, but users may prefer to use their own data. The data provide a an estimate of general meteorological conditions across Greater London. For meteorological species (wd, ws, rain, solar) each data point is formed by averaging measurements from a subset of LAQN monitoring sites that have been identified as having minimal disruption from local obstacles and a long term reliable dataset. The exact sites used varies between species, but include between two and five sites per species. Therefore, the data should represent 'London scale' meteorology, rather than local conditions.

`importKCL()` is equivalent to `importImperial()` and is provided for back-compatibility reasons only. New users should use `importImperial()`.

Value

Returns a data frame of hourly mean values with date in POSIXct class and time zone GMT.

Author(s)

David Carslaw and Ben Barratt

See Also

Other import functions: `importADMS()`, `importAURN()`, `importEurope()`, `importMeta()`, `importTraj()`, `importUKAQ()`

Examples

```
## import all pollutants from Marylebone Rd from 1990:2009
## Not run:
```

```
mary <- importImperial(site = "my1", year = 2000:2009)

## End(Not run)

## import nox, no2, o3 from Marylebone Road and North Kensington for 2000
## Not run:
thedata <-
  importImperial(
    site = c("my1", "kc1"),
    year = 2000,
    pollutant = c("nox", "no2", "o3")
  )

## End(Not run)

## import met data too...
## Not run:
my1 <- importImperial(site = "my1", year = 2008, meteo = TRUE)

## End(Not run)
```

`importMeta` *Import monitoring site meta data for UK and European networks*

Description

Function to import meta data for air quality monitoring sites. By default, the function will return the site latitude, longitude and site type, as well as the code used in functions like `importUKAQ()`, `importImperial()` and `importEurope()`. Additional information may optionally be returned.

Usage

```
importMeta(source = "aurn", all = FALSE, year = NA, duplicate = FALSE)
```

Arguments

`source` One or more air quality networks for which data is available through openair. Available networks include:

- `"aurn"`, The UK Automatic Urban and Rural Network.
- `"aqe"`, The Air Quality England Network.
- `"saqn"`, The Scottish Air Quality Network.
- `"waqn"`, The Welsh Air Quality Network.
- `"ni"`, The Northern Ireland Air Quality Network.
- `"local"`, Locally managed air quality networks in England.
- `"imperial"`, Imperial College London (formerly King's College London) networks.
- `"europe"`, European AirBase/e-reporting data.

There are two additional options provided for convenience:

- "ukaq" will return metadata for all networks for which data is imported by importUKAQ() (i.e., AURN, AQE, SAQN, WAQN, NI, and the local networks).
- "all" will import all available metadata (i.e., "ukaq" plus "imperial" and "europe").

all — When all = FALSE only the site code, site name, latitude and longitude and site type are imported. Setting all = TRUE will import all available meta data and provide details (when available) or the individual pollutants measured at each site.

year — If a single year is selected, only sites that were open at some point in that year are returned. If all = TRUE only sites that measured a particular pollutant in that year are returned. Year can also be a sequence e.g. year = 2010:2020 or of length 2 e.g. year = c(2010, 2020), which will return only sites that were open over the duration. Note that year is ignored when the source is either "imperial" or "europe".

duplicate — Some UK air quality sites are part of multiple networks, so could appear more than once when source is a vector of two or more. The default argument, FALSE, drops duplicate sites. TRUE will return them.

Details

This function imports site meta data from several networks in the UK and Europe:

- "aurn", The UK Automatic Urban and Rural Network.
- "aqe", The Air Quality England Network.
- "saqn", The Scottish Air Quality Network.
- "waqn", The Welsh Air Quality Network.
- "ni", The Northern Ireland Air Quality Network.
- "local", Locally managed air quality networks in England.
- "imperial", Imperial College London (formerly King's College London) networks.
- "europe", Hourly European data (Air Quality e-Reporting) based on a simplified version of the {saqgetr} package.

By default, the function will return the site latitude, longitude and site type. If the option all = TRUE is used, much more detailed information is returned. The following metadata columns are available in the complete dataset:

- **source**: The network with which the site is associated. Note that some monitoring sites are part of multiple networks (e.g., the AURN & SAQN) so the same site may feature twice under different sources.
- **code**: The site code, used to import data from specific sites of interest.
- **site**: The site name, which is more human-readable than the site code.
- **site_type**: A description of the site environment. Read more at https://uk-air.defra.gov.uk/networks/site-types.

- **latitude** and **longitude**: The coordinates of the monitoring station, using the World Geodetic System (https://epsg.io/4326).
- **start_date** and **end_date**: The opening and closing dates of the monitoring station. If by_pollutant = TRUE, these dates are instead the first and last dates at which specific pollutants were measured. A missing value, NA, indicates that monitoring is ongoing.
- **ratified_to**: The date to which data has been ratified (i.e., 'quality checked'). Data after this date is subject to change.
- **zone** and **agglomeration**: The UK is divided into agglomeration zones (large urban areas) and non-agglomeration zones for air quality assessment, which are given in these columns.
- **local_authority**: The local authority in which the monitoring station is found.
- **provider** and **code**: The specific provider of the locally managed dataset (e.g., "londonair").

Thanks go to Trevor Davies (Ricardo), Dr Stuart Grange (EMPA) and Dr Ben Barratt (KCL) and for making these data available.

Value

A data frame with meta data.

Author(s)

David Carslaw

See Also

the networkMap() function from the openairmaps package which can visualise site metadata on an interactive map.

Other import functions: importADMS(), importAURN(), importEurope(), importImperial(), importTraj(), importUKAQ()

Examples

```
## Not run:
# basic info:
meta <- importMeta(source = "aurn")

# more detailed information:
meta <- importMeta(source = "aurn", all = TRUE)

# from the Scottish Air Quality Network:
meta <- importMeta(source = "saqn", all = TRUE)

# from multiple networks:
meta <- importMeta(source = c("aurn", "aqe", "local"))

## End(Not run)
```

`importTraj` *Import pre-calculated HYSPLIT 96-hour back trajectories*

Description

Function to import pre-calculated back trajectories using the NOAA HYSPLIT model. The trajectories have been calculated for a select range of locations which will expand in time. They cover the last 20 years or so and can be used together with other `openair` functions.

Usage

```
importTraj(site = "london", year = 2009, local = NA, progress = TRUE)
```

Arguments

`site` Site code of the network site to import e.g. "london". Only one site can be imported at a time. The following sites are typically available from 2000-2012, although some UK ozone sites go back to 1988 (code, location, lat, lon, year):

abudhabi	Abu Dhabi	24.43000	54.408000	2012-2013
ah	Aston Hill	52.50385	-3.041780	1988-2013
auch	Auchencorth Moss	55.79283	-3.242568	2006-2013
berlin	Berlin, Germany	52.52000	13.400000	2000-2013
birm	Birmigham Centre	52.47972	-1.908078	1990-2013
boston	Boston, USA	42.32900	-71.083000	2008-2013
bot	Bottesford	52.93028	-0.814722	1990-2013
bukit	Bukit Kototabang, Indonesia	-0.19805	100.318000	1996-2013
chittagong	Chittagong, Bangladesh	22.37000	91.800000	2010-2013
dhaka	Dhaka, Bangladesh	23.70000	90.375000	2010-2013
ed	Edinburgh	55.95197	-3.195775	1990-2013
elche	Elche, Spain	38.27000	-0.690000	2004-2013
esk	Eskdalemuir	55.31530	-3.206110	1998-2013
gibraltar	Gibraltar	36.13400	-5.347000	2005-2010
glaz	Glazebury	53.46008	-2.472056	1998-2013
groningen	Groningen	53.40000	6.350000	2007-2013
har	Harwell	51.57108	-1.325283	1988-2013
hk	Hong Kong	22.29000	114.170000	1998-2013
hm	High Muffles	54.33500	-0.808600	1988-2013
kuwait	Kuwait City	29.36700	47.967000	2008-2013
lb	Ladybower	53.40337	-1.752006	1988-2013
london	Central London	51.50000	-0.100000	1990-2013
lh	Lullington Heath	50.79370	0.181250	1988-2013
ln	Lough Navar	54.43951	-7.900328	1988-2013
mh	Mace Head	53.33000	-9.900000	1988-2013
ny-alesund	Ny-Alesund, Norway	78.91763	11.894653	2009-2013
oslo	Oslo	59.90000	10.750000	2010-2013
paris	Paris, France	48.86200	2.339000	2000-2013

roch	Rochester Stoke	51.45617	0.634889	1988-2013
rotterdam	Rotterdam	51.91660	4.475000	2010-2013
saopaulo	Sao Paulo	-23.55000	-46.640000	2000-2013
sib	Sibton	52.29440	1.463970	1988-2013
sv	Strath Vaich	57.73446	-4.776583	1988-2013
wuhan	Wuhan, China	30.58300	114.280000	2008-2013
yw	Yarner Wood	50.59760	-3.716510	1988-2013

`year` Year or years to import. To import a sequence of years from 1990 to 2000 use `year = 1990:2000`. To import several specific years use `year = c(1990, 1995, 2000)` for example.

`local` File path to .RData trajectory files run by user and not stored on the Ricardo web server. These files would have been generated from the Hysplit trajectory code shown in the appendix of the openair manual. An example would be `local = 'c:/users/david/TrajFiles/'`.

`progress` Show a progress bar when many receptors/years are being imported? Defaults to `TRUE`.

Details

This function imports pre-calculated back trajectories using the HYSPLIT trajectory model (Hybrid Single Particle Lagrangian Integrated Trajectory Model. Back trajectories provide some very useful information for air quality data analysis. However, while they are commonly calculated by researchers it is generally difficult for them to be calculated on a routine basis and used easily. In addition, the availability of back trajectories over several years can be very useful, but again difficult to calculate.

Trajectories are run at 3-hour intervals and stored in yearly files (see below). The trajectories are started at ground-level (10m) and propagated backwards in time.

These trajectories have been calculated using the Global NOAA-NCEP/NCAR reanalysis data archives. The global data are on a latitude-longitude grid (2.5 degree). Note that there are many different meteorological data sets that can be used to run HYSPLIT e.g. including ECMWF data. However, in order to make it practicable to run and store trajectories for many years and sites, the NOAA-NCEP/NCAR reanalysis data is most useful. In addition, these archives are available for use widely, which is not the case for many other data sets e.g. ECMWF. HYSPLIT calculated trajectories based on archive data may be distributed without permission. For those wanting, for example, to consider higher resolution meteorological data sets it may be better to run the trajectories separately.

We are extremely grateful to NOAA for making HYSPLIT available to produce back trajectories in an open way. We ask that you cite HYSPLIT if used in published work.

Users can supply their own trajectory files to plot in openair. These files must have the following fields: date, lat, lon and hour.inc (see details below).

The files consist of the following information:

date This is the arrival point time and is repeated the number of times equal to the length of the back trajectory — typically 96 hours (except early on in the file). The format is `POSIXct`. It is this field that should be used to link with air quality data. See example below.

receptor Receptor number, currently only 1.

year The year

month Month 1-12

day Day of the month 1-31

hour Hour of the day 0-23 GMT

hour.inc Number of hours back in time e.g. 0 to -96.

lat Latitude in decimal format.

lon Longitude in decimal format.

height Height of trajectory (m).

pressure Pressure of trajectory (kPa).

Value

Returns a data frame with pre-calculated back trajectories.

Note

The trajectories were run using the February 2011 HYSPLIT model. The function is primarily written to investigate a single site at a time for a single year. The trajectory files are quite large and care should be exercised when importing several years and/or sites.

Author(s)

David Carslaw

See Also

Other import functions: `importADMS()`, `importAURN()`, `importEurope()`, `importImperial()`, `importMeta()`, `importUKAQ()`

Other trajectory analysis functions: `trajCluster()`, `trajLevel()`, `trajPlot()`

Examples

```
## import trajectory data for London in 2009
## Not run:
mytraj <- importTraj(site = "london", year = 2009)

## End(Not run)

## combine with measurements
## Not run:
theData <- importAURN(site = "kc1", year = 2009)
mytraj <- merge(mytraj, theData, by = "date")

## End(Not run)
```

importUKAQ *Import data from the UK Air Pollution Networks*

Description

Functions for importing air pollution data from a range of UK networks including the Automatic Urban and Rural Network (AURN), the individual England (AQE), Scotland (SAQN), Wales (WAQN) and Northern Ireland (NI) Networks, and many "locally managed" monitoring networks across England. Files are imported from a remote server operated by Ricardo that provides air quality data files as R data objects. For an up to date list of available sites that can be imported, see `importMeta()`.

Usage

```
importUKAQ(
  site = "my1",
  year = 2022,
  source = NULL,
  data_type = "hourly",
  pollutant = "all",
  hc = FALSE,
  meta = FALSE,
  meta_columns = c("site_type", "latitude", "longitude"),
  meteo = TRUE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE,
  progress = TRUE
)
```

Arguments

`site`	Site code of the site to import, e.g., `"my1"` is Marylebone Road. Site codes can be discovered through the use of `importMeta()`. Several sites can be imported at once. For example, `site = c("my1", "nott")` imports both Marylebone Road and Nottingham. Sites from different networks can be imported through also providing multiple `sources`. Site codes can be upper or lower case.
`year`	Year(s) to import. To import a series of years use, e.g., `2000:2020`. To import several specific years use `year = c(2000, 2010, 2020)`.
`source`	The network to which the `site`(s) belong. The default, `NULL`, allows `importUKAQ()` to guess the correct `source`, preferring national networks over locally managed networks. Alternatively, users can define a `source`. Providing a single network will attempt to import all of the given `sites` from the provided network. Alternatively, a vector of sources can be provided of the same length as `site` to indicate which network each `site` individually belongs. Available networks include: • `"aurn"`, The UK Automatic Urban and Rural Network.

- "aqe", The Air Quality England Network.
- "saqn", The Scottish Air Quality Network.
- "waqn", The Welsh Air Quality Network.
- "ni", The Northern Ireland Air Quality Network.
- "local", Locally managed air quality networks in England.

data_type — The type of data to be returned, defaulting to "hourly" data. Alternative data types include:

- "daily": Daily average data.
- "monthly": Monthly average data with data capture information for the whole network.
- "annual": Annual average data with data capture information for the whole network.
- "15_min": 15-minute average SO2 concentrations.
- "8_hour": 8-hour rolling mean concentrations for O3 and CO.
- "24_hour": 24-hour rolling mean concentrations for particulates.
- "daily_max_8": Maximum daily rolling 8-hour maximum for O3 and CO.
- "daqi": Daily Air Quality Index (DAQI). See here for more details of how the index is defined. Note that this data_type is not available for locally managed monitoring networks.

pollutant — Pollutants to import. If omitted will import all pollutants from a site. To import only NOx and NO2 for example use pollutant = c("nox", "no2"). Pollutant names can be upper or lower case.

hc — Include hydrocarbon measurements in the imported data? Defaults to FALSE as most users will not be interested in using hydrocarbon data.

meta — Append metadata columns to data for each selected site? Defaults to FALSE. Columns are defined using meta_columns.

meta_columns — The specific columns to append when meta = TRUE. Defaults to site type, latitude and longitude. Can be any of "site_type", "latitude", "longitude", "zone", "agglomeration", and "local_authority" (as well as "provider" for locally managed data). See importMeta() for more complete information.

meteo — Append modelled meteorological data, if available? Defaults to TRUE, which will return wind speed (ws), wind direction (wd) and ambient temperature (air_temp). The variables are calculated from using the WRF model run by Ricardo Energy & Environment and are available for most but not all networks. Setting meteo = FALSE is useful if you have other meteorological data to use in preference, for example from the worldmet package.

ratified — Append qc column(s) to hourly data indicating whether each species was ratified (i.e., quality-checked)? Defaults to FALSE.

to_narrow — Return the data in a "narrow"/"long"/"tidy" format? By default the returned data is "wide" and has a column for each pollutant/variable. When to_narrow = TRUE the data are returned with a column identifying the pollutant name and a column containing the corresponding concentration/statistic. Defaults to FALSE.

verbose — Print messages to the console if hourly data cannot be imported? Default is FALSE. TRUE is useful for debugging as the specific year(s), site(s) and source(s) which cannot be imported will be returned.

progress Show a progress bar when many sites/years are being imported? Defaults to TRUE.

Value

a tibble

Importing UK Air Pollution Data

This family of functions has been written to make it easy to import data from across several UK air quality networks. Ricardo have provided .RData files (R workspaces) of all individual sites and years, as well as up to date meta data. These files are updated on a daily basis. This approach requires a link to the Internet to work.

There are several advantages over the web portal approach where .csv files are downloaded.

- First, it is quick to select a range of sites, pollutants and periods (see examples below).
- Second, storing the data as .RData objects is very efficient as they are about four times smaller than .csv files — which means the data downloads quickly and saves bandwidth.
- Third, the function completely avoids any need for data manipulation or setting time formats, time zones etc. The function also has the advantage that the proper site name is imported and used in openair functions.

Users should take care if using data from both openair *and* web portals (for example, UK AIR). One key difference is that the data provided by openair is date *beginning*, whereas the web portal provides date *ending*. Hourly concentrations may therefore appear offset by an hour, for example.

The data are imported by stacking sites on top of one another and will have field names site, code (the site code) and pollutant.

By default, the function returns hourly average data. However, annual, monthly, daily and 15 minute data (for SO2) can be returned using the option data_type. Annual and monthly data provide whole network information including data capture statistics.

All units are expressed in mass terms for gaseous species (ug/m3 for NO, NO2, NOx (as NO2), SO2 and hydrocarbons; and mg/m3 for CO). PM10 concentrations are provided in gravimetric units of ug/m3 or scaled to be comparable with these units. Over the years a variety of instruments have been used to measure particulate matter and the technical issues of measuring PM10 are complex. In recent years the measurements rely on FDMS (Filter Dynamics Measurement System), which is able to measure the volatile component of PM. In cases where the FDMS system is in use there will be a separate volatile component recorded as 'v10' and non-volatile component 'nv10', which is already included in the absolute PM10 measurement. Prior to the use of FDMS the measurements used TEOM (Tapered Element Oscillating. Microbalance) and these concentrations have been multiplied by 1.3 to provide an estimate of the total mass including the volatile fraction.

Some sites report hourly and daily PM10 and / or PM2.5. When data_type = "daily" and there are both hourly and 'proper' daily measurements available, these will be returned as e.g. "pm2.5" and "gr_pm2.5"; the former corresponding to data based on original hourly measurements and the latter corresponding to daily gravimetric measurements.

The function returns modelled hourly values of wind speed (ws), wind direction (wd) and ambient temperature (air_temp) if available (generally from around 2010). These values are modelled using the WRF model operated by Ricardo.

The BAM (Beta-Attenuation Monitor) instruments that have been incorporated into the network throughout its history have been scaled by 1.3 if they have a heated inlet (to account for loss of volatile particles) and 0.83 if they do not have a heated inlet. The few TEOM instruments in the network after 2008 have been scaled using VCM (Volatile Correction Model) values to account for the loss of volatile particles. The object of all these scaling processes is to provide a reasonable degree of comparison between data sets and with the reference method and to produce a consistent data record over the operational period of the network, however there may be some discontinuity in the time series associated with instrument changes.

No corrections have been made to the PM2.5 data. The volatile component of FDMS PM2.5 (where available) is shown in the 'v2.5' column.

Author(s)

David Carslaw, Trevor Davies, and Jack Davison

See Also

Other import functions: `importADMS()`, `importAURN()`, `importEurope()`, `importImperial()`, `importMeta()`, `importTraj()`

Examples

```
## Not run:
# import a single site from the AURN
importUKAQ("my1", year = 2022)

# import sites from another network
importUKAQ(c("bn1", "bn2"), year = 2022, source = "aqe")

# import sites across multiple networks
importUKAQ(c("my1", "bn1", "bn2"),
  year = 2022,
  source = c("aurn", "aqe", "aqe")
)

# get "long" format hourly data with a ratification flag
importUKAQ(
  "card",
  source = "waqn",
  year = 2022,
  to_narrow = TRUE,
  ratified = TRUE
)

# import other data types, filtering by pollutant
importUKAQ(
  data_type = "annual",
  pollutant = c("no2", "pm2.5", "pm10"),
  source = c("aurn", "aqe")
)
```

```
## End(Not run)
```

`modStats` *Calculate common model evaluation statistics*

Description

Function to calculate common numerical model evaluation statistics with flexible conditioning.

Usage

```
modStats(
  mydata,
  mod = "mod",
  obs = "obs",
 statistic = c("n", "FAC2", "MB", "MGE", "NMB", "NMGE", "RMSE", "r", "COE", "IOA"),
  type = "default",
  rank.name = NULL,
  ...
)
```

Arguments

`mydata`	A data frame.
`mod`	Name of a variable in `mydata` that represents modelled values.
`obs`	Name of a variable in `mydata` that represents measured values.
`statistic`	The statistic to be calculated. See details below for a description of each.
`type`	`type` determines how the data are split i.e. conditioned, and then plotted. The default is will produce statistics using the entire data. `type` can be one of the built-in types as detailed in `cutData` e.g. "season", "year", "weekday" and so on. For example, `type = "season"` will produce four sets of statistics — one for each season. It is also possible to choose `type` as another variable in the data frame. If that variable is numeric, then the data will be split into four quantiles (if possible) and labelled accordingly. If type is an existing character or factor variable, then those categories/levels will be used directly. This offers great flexibility for understanding the variation of different variables and how they depend on one another. More than one type can be considered e.g. `type = c("season", "weekday")` will produce statistics split by season and day of the week.
`rank.name`	Simple model ranking can be carried out if `rank.name` is supplied. `rank.name` will generally refer to a column representing a model name, which is to ranked. The ranking is based the COE performance, as that indicator is arguably the best single model performance indicator available.
`...`	Arguments passed on to `cutData`

x A data frame containing a field date.

names By default, the columns created by cutData() are named after their type option. Specifying names defines other names for the columns, which map onto the type options in the same order they are given. The length of names should therefore be equal to the length of type.

suffix If name is not specified, suffix will be appended to any added columns that would otherwise overwrite existing columns. For example, cutData(mydata, "nox", suffix = "_cuts") would append a nox_cuts column rather than overwriting nox.

hemisphere Can be "northern" or "southern", used to split data into seasons.

n.levels Number of quantiles to split numeric data into.

start.day What day of the week should the type = "weekday" start on? The user can change the start day by supplying an integer between 0 and 6. Sunday = 0, Monday = 1, ... For example to start the weekday plots on a Saturday, choose start.day = 6.

start.season What order should the season be. By default, the order is spring, summer, autumn, winter. start.season = "winter" would plot winter first.

is.axis A logical (TRUE/FALSE), used to request shortened cut labels for axes.

local.tz Used for identifying whether a date has daylight savings time (DST) applied or not. Examples include local.tz = "Europe/London", local.tz = "America/New_York", i.e., time zones that assume DST. https://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones shows time zones that should be valid for most systems. It is important that the original data are in GMT (UTC) or a fixed offset from GMT.

latitude,longitude The decimal latitude and longitudes used when type = "daylight". Note that locations west of Greenwich have negative longitudes.

drop How to handle empty factor levels. One of:

- "default": Sensible defaults selected on a case-by-case basis for different type options.
- "empty": Drop all empty factor levels.
- "none": Retain all empty factor levels, where possible. For example, for type = "hour", all factor levels from 0 and 23 will be represented.
- "outside": Retain empty factor levels within the range of the data. For example, for type = "hour" when the data only contains data for 1 AM and 5 AM, the factor levels, 1, 2, 3, 4 and 5 will be retained.

Some of these options only apply to certain type options. For example, for type = "year", "outside" is equivalent to "none" as there is no fixed range of years to use in the "none" case.

Details

This function is under development and currently provides some common model evaluation statistics. These include (to be mathematically defined later):

- n, the number of complete pairs of data.
- $FAC2$, fraction of predictions within a factor of two.
- MB, the mean bias.
- MGE, the mean gross error.
- NMB, the normalised mean bias.
- $NMGE$, the normalised mean gross error.
- $RMSE$, the root mean squared error.
- r, the Pearson correlation coefficient. Note, can also supply and argument `method` e.g. `method = "spearman"`. Also returned is the P value of the correlation coefficient, P, which may present as `0` for very low values.
- COE, the *Coefficient of Efficiency* based on Legates and McCabe (1999, 2012). There have been many suggestions for measuring model performance over the years, but the COE is a simple formulation which is easy to interpret.

  A perfect model has a COE = 1. As noted by Legates and McCabe although the COE has no lower bound, a value of COE = 0.0 has a fundamental meaning. It implies that the model is no more able to predict the observed values than does the observed mean. Therefore, since the model can explain no more of the variation in the observed values than can the observed mean, such a model can have no predictive advantage.

  For negative values of COE, the model is less effective than the observed mean in predicting the variation in the observations.
- IOA, the Index of Agreement based on Willmott et al. (2011), which spans between -1 and +1 with values approaching +1 representing better model performance.

  An IOA of 0.5, for example, indicates that the sum of the error-magnitudes is one half of the sum of the observed-deviation magnitudes. When IOA = 0.0, it signifies that the sum of the magnitudes of the errors and the sum of the observed-deviation magnitudes are equivalent. When IOA = -0.5, it indicates that the sum of the error-magnitudes is twice the sum of the perfect model-deviation and observed-deviation magnitudes. Values of IOA near -1.0 can mean that the model-estimated deviations about O are poor estimates of the observed deviations; but, they also can mean that there simply is little observed variability - so some caution is needed when the IOA approaches -1.

All statistics are based on complete pairs of `mod` and `obs`.

Conditioning is possible through setting `type`, which can be a vector e.g. `type = c("weekday", "season")`.

Value

Returns a data frame with model evaluation statistics.

Author(s)

David Carslaw

References

Legates DR, McCabe GJ. (1999). Evaluating the use of goodness-of-fit measures in hydrologic and hydroclimatic model validation. Water Resources Research 35(1): 233-241.

Legates DR, McCabe GJ. (2012). A refined index of model performance: a rejoinder, International Journal of Climatology.

Willmott, C.J., Robeson, S.M., Matsuura, K., 2011. A refined index of model performance. International Journal of Climatology.

See Also

Other model evaluation functions: `TaylorDiagram()`, `conditionalEval()`, `conditionalQuantile()`

Examples

```
## the example below is somewhat artificial --- assuming the observed
## values are given by NOx and the predicted values by NO2.

modStats(mydata, mod = "no2", obs = "nox")

## evaluation stats by season

modStats(mydata, mod = "no2", obs = "nox", type = "season")
```

---

mydata *Example air quality monitoring data for openair*

Description

The `mydata` dataset is provided as an example dataset as part of the openair package. The dataset contains hourly measurements of air pollutant concentrations, wind speed and wind direction collected at the Marylebone (London) air quality monitoring supersite between 1st January 1998 and 23rd June 2005.

Usage

```
mydata
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 65533 rows and 10 columns.

Details

date Observation date/time stamp in year-month-day hour:minute:second format (POSIXct).

ws Wind speed, in m/s, as numeric vector.

wd Wind direction, in degrees from North, as a numeric vector.

nox Oxides of nitrogen concentration, in ppb, as a numeric vector.

no2 Nitrogen dioxide concentration, in ppb, as a numeric vector.

o3 Ozone concentration, in ppb, as a numeric vector.

pm10 Particulate PM10 fraction measurement, in ug/m3 (raw TEOM), as a numeric vector.

so2 Sulfur dioxide concentration, in ppb, as a numeric vector.

co Carbon monoxide concentration, in ppm, as a numeric vector.

pm25 Particulate PM2.5 fraction measurement, in ug/m3, as a numeric vector.

Note

openair functions generally require data frames with a field "date" that can be in either `POSIXct` or `Date` format

Source

`mydata` was compiled from data archived in the London Air Quality Archive. See https://londonair.org.uk for site details.

Examples

```
# basic structure
head(mydata)
```

---

openColours *Pre-defined openair colours and definition of user-defined colours*

Description

This in primarily an internal openair function to make it easy for users to select particular colour schemes, or define their own range of colours of a user-defined length.

Usage

```
openColours(scheme = "default", n = 100)
```

Arguments

`scheme`	Any one of the pre-defined `openair` schemes (e.g., `"increment"`) or a user-defined palette (e.g., `c("red", "orange", "gold")`). See `?openColours` for a full list of available schemes.
`n`	number of colours required.

Value

A character vector of hex codes

Schemes

The following schemes are made available by `openColours()`:

Sequential Colours:

- "default", "increment", "brewer1", "heat", "jet", "turbo", "hue", "greyscale".
- Simplified versions of the `viridis` colours: "viridis", "plasma", "magma", "inferno", "cividis", and "turbo".
- Simplified versions of the `RColorBrewer` sequential palettes: "Blues", "BuGn", "BuPu", "GnBu", "Greens", "Greys", "Oranges", "OrRd", "PuBu", "PuBuGn", "PuRd", "Purples", "RdPu", "Reds", "YlGn", "YlGnBu", "YlOrBr", "YlOrRd".

Diverging Palettes:

- Simplified versions of the `RColorBrewer` diverging palettes: "BrBG", "PiYG", "PRGn", "PuOr", "RdBu", "RdGy", "RdYlBu", "RdYlGn", "Spectral".

Qualitative Palettes:

- Simplified versions of the `RColorBrewer` qualitative palettes: "Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3".
- "okabeito" (or "cbPalette"), a colour-blind safe palette based on the work of Masataka Okabe and Kei Ito (https://jfly.uni-koeln.de/color/)
- "tol.bright" (or "tol"), "tol.muted" and "tol.light", colour-blind safe palettes based on the work of Paul Tol.
- "tableau" and "observable", aliases for the "Tableau10" (https://www.tableau.com/blog/colors-upgrade-tableau-10-56782) and "Observable10" (https://observablehq.com/blog/crafting-data-colors) colour palettes. These could be useful for consistency between openair plots and with figures made in Tableau or Observable Plot.

UK Government Palettes:

- "daqi" and "daqi.bands", the colours associated with the UK daily air quality index; "daqi" (a palette of 10 colours, corresponding to each index value) or "daqi.bands" (4 colours, corresponding to each band - Low, Moderate, High, and Very High). These colours were taken directly from https://check-air-quality.service.gov.uk/ and may be useful in figures like `calendarPlot()`.
- "gaf.cat", "gaf.focus" and "gaf.seq", colours recommended by the UK Government Analysis function (https://analysisfunction.civilservice.gov.uk/policy-store/data-visualisation-colours-ir "gaf.cat" will return the 'categorical' palette (max 6 colours), "gaf.focus" the 'focus' palette (max 2 colours), and "gaf.seq" the 'sequential' palette.

Details

Because of the way many of the schemes have been developed they only exist over certain number of colour gradations (typically 3–10) — see `?brewer.pal` for actual details. If less than or more than the required number of colours is supplied then `openair` will interpolate the colours.

Each of the pre-defined schemes have merits and their use will depend on a particular situation. For showing incrementing concentrations, e.g., high concentrations emphasised, then "default", "heat", "jet", "turbo", and "increment" are very useful. See also the description of `RColorBrewer` schemes for the option `scheme`.

To colour-code categorical-type problems, e.g., colours for different pollutants, "hue" and "brewer1" are useful.

When publishing in black and white, "greyscale" is often convenient. With most openair functions, as well as generating a greyscale colour gradient, it also resets strip background and other coloured text and lines to greyscale values.

Failing that, the user can define their own schemes based on R colour names. To see the full list of names, type `colors()` into R.

Author(s)

David Carslaw

Jack Davison

References

https://colorbrewer2.org/

https://check-air-quality.service.gov.uk/

https://analysisfunction.civilservice.gov.uk/policy-store/data-visualisation-colours-in-charts/

Examples

```
# to return 5 colours from the "jet" scheme:
cols <- openColours("jet", 5)
cols

# to interpolate between named colours e.g. 10 colours from yellow to
#  green to red:
cols <- openColours(c("yellow", "green", "red"), 10)
cols
```

percentileRose *Function to plot percentiles by wind direction*

Description

`percentileRose()` plots percentiles by wind direction with flexible conditioning. The plot can display multiple percentile lines or filled areas.

Usage

```
percentileRose(
  mydata,
  pollutant = "nox",
  wd = "wd",
  type = "default",
  percentile = c(25, 50, 75, 90, 95),
  smooth = FALSE,
  method = "default",
  cols = "default",
  angle = 10,
  mean = TRUE,
  mean.lty = 1,
  mean.lwd = 3,
  mean.col = "grey",
  fill = TRUE,
  intervals = NULL,
  angle.scale = 45,
  offset = 0,
  auto.text = TRUE,
  key.title = NULL,
  key.position = "bottom",
  strip.position = "top",
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

`mydata`	A data frame minimally containing `wd` and a numeric field to plot — `pollutant`.
`pollutant`	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. `pollutant = "nox"`. More than one pollutant can be supplied e.g. `pollutant = c("no2", "o3")` provided there is only one `type`.
`wd`	Name of wind direction field.
`type`	Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other

options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`percentile` The percentile value(s) to plot. Must be between 0–100. If `percentile = NA` then only a mean line will be shown.

`smooth` Should the wind direction data be smoothed using a cyclic spline?

`method` When `method = "default"` the supplied percentiles by wind direction are calculated. When `method = "cpf"` the conditional probability function (CPF) is plotted and a single (usually high) percentile level is supplied. The CPF is defined as CPF = my/ny, where my is the number of samples in the wind sector y with mixing ratios greater than the *overall* percentile concentration, and ny is the total number of samples in the same wind sector (see Ashbaugh et al., 1985).

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`angle` Default angle of "spokes" is when `smooth = FALSE`.

`mean` Show the mean by wind direction as a line?

`mean.lty` Line type for mean line.

`mean.lwd` Line width for mean line.

`mean.col` Line colour for mean line.

`fill` Should the percentile intervals be filled (default) or should lines be drawn (`fill = FALSE`).

`intervals` User-supplied intervals for the scale e.g. `intervals = c(0, 10, 30, 50)`.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

`offset` `offset` controls the size of the 'hole' in the middle and is expressed on a scale of `0` to `100`, where `0` is no hole and `100` is a hole that takes up the entire plotting area.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `lwd` overrides the line width for percentile lines when `fill = FALSE`.
- `annotate = FALSE` will not plot the N/E/S/W labels.

Details

`percentileRose()` calculates percentile levels of a pollutant and plots them by wind direction. One or more percentile levels can be calculated and these are displayed as either filled areas or as lines.

The wind directions are rounded to the nearest 10 degrees, consistent with surface data from the UK Met Office before a smooth is fitted. The levels by wind direction are optionally calculated using a cyclic smooth cubic spline using the option `smooth`. If `smooth = FALSE` then the data are shown in 10 degree sectors.

The `percentileRose` function compliments other similar functions including `windRose()`, `pollutionRose()`, `polarFreq()` or `polarPlot()`. It is most useful for showing the distribution of concentrations by wind direction and often can reveal different sources e.g. those that only affect high percentile concentrations such as a chimney stack.

Similar to other functions, flexible conditioning is available through the `type` option. It is easy for example to consider multiple percentile values for a pollutant by season, year and so on. See examples below.

`percentileRose` also offers great flexibility with the scale used and the user has fine control over both the range, interval and colour.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

References

Ashbaugh, L.L., Malm, W.C., Sadeh, W.Z., 1985. A residence time probability analysis of sulfur concentrations at ground canyon national park. Atmospheric Environment 19 (8), 1263-1270.

See Also

Other polar directional analysis functions: `polarAnnulus()`, `polarCluster()`, `polarDiff()`, `polarFreq()`, `polarPlot()`, `pollutionRose()`, `windRose()`

Examples

```
# basic percentile plot
percentileRose(mydata, pollutant = "o3")

# 50/95th percentiles of ozone, with different colours
percentileRose(mydata, pollutant = "o3", percentile = c(50, 95), col = "brewer1")

## Not run:
# percentiles of ozone by year, with different colours
percentileRose(
  mydata,
  type = "year",
  pollutant = "o3",
  col = "brewer1",
  layout = c(4, 2)
)

# percentile concentrations by season and day/nighttime..
percentileRose(
  mydata,
  type = c("daylight", "season"),
  pollutant = "o3",
  col = "brewer1"
)

## End(Not run)
```

polarAnnulus *Bivariate polarAnnulus plot*

Description

Typically plots the concentration of a pollutant by wind direction and as a function of time as an annulus. The function is good for visualising how concentrations of pollutants vary by wind direction and a time period e.g. by month, day of week.

Usage

```
polarAnnulus(
  mydata,
  pollutant = "nox",
  resolution = "fine",
  local.tz = NULL,
  period = "hour",
  type = "default",
  statistic = "mean",
  percentile = NA,
  limits = NULL,
  cols = "default",
  col.na = "white",
  offset = 50,
  angle.scale = 0,
  min.bin = 1,
  exclude.missing = TRUE,
  date.pad = FALSE,
  force.positive = TRUE,
  k = c(20, 10),
  normalise = FALSE,
  strip.position = "top",
  key.title = paste(statistic, pollutant, sep = " "),
  key.position = "right",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame minimally containing date, wd and a pollutant.
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox". There can also be more than one pollutant specified e.g. pollutant = c("nox", "no2"). The main use of using two or more pollutants is for model evaluation where two species would be expected to

have similar concentrations. This saves the user stacking the data and it is possible to work with columns of data directly. A typical use would be `pollutant = c("obs", "mod")` to compare two columns "obs" (the observations) and "mod" (modelled values).

`resolution` Two plot resolutions can be set: "normal" and "fine" (the default).

`local.tz` Should the results be calculated in local time that includes a treatment of daylight savings time (DST)? The default is not to consider DST issues, provided the data were imported without a DST offset. Emissions activity tends to occur at local time e.g. rush hour is at 8 am every day. When the clocks go forward in spring, the emissions are effectively released into the atmosphere typically 1 hour earlier during the summertime i.e. when DST applies. When plotting diurnal profiles, this has the effect of "smearing-out" the concentrations. Sometimes, a useful approach is to express time as local time. This correction tends to produce better-defined diurnal profiles of concentration (or other variables) and allows a better comparison to be made with emissions/activity data. If set to `FALSE` then GMT is used. Examples of usage include `local.tz = "Europe/London"`, `local.tz = "America/New_York"`. See `cutData` and `import` for more details.

`period` This determines the temporal period to consider. Options are "hour" (the default, to plot diurnal variations), "season" to plot variation throughout the year, "weekday" to plot day of the week variation and "trend" to plot the trend by wind direction.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`statistic` The statistic that should be applied to each wind speed/direction bin. Can be "mean" (default), "median", "max" (maximum), "frequency". "stdev" (standard deviation), "weighted.mean" or "cpf" (Conditional Probability Function). Because of the smoothing involved, the colour scale for some of these statistics is only to provide an indication of overall pattern and should not be interpreted in concentration units e.g. for `statistic = "weighted.mean"` where the bin mean is multiplied by the bin frequency and divided by the total frequency. In many cases using `polarFreq` will be better. Setting `statistic = "weighted.mean"`

can be useful because it provides an indication of the concentration * frequency of occurrence and will highlight the wind speed/direction conditions that dominate the overall mean.

`percentile` If `statistic = "percentile"` or `statistic = "cpf"` then `percentile` is used, expressed from 0 to 100. Note that the percentile value is calculated in the wind speed, wind direction 'bins'. For this reason it can also be useful to set `min.bin` to ensure there are a sufficient number of points available to estimate a percentile. See `quantile` for more details of how percentiles are calculated.

`limits` The function does its best to choose sensible limits automatically. However, there are circumstances when the user will wish to set different ones. An example would be a series of plots showing each year of data separately. The limits are set in the form `c(lower, upper)`, so `limits = c(0, 100)` would force the plot limits to span 0-100.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`col.na` Colour to be used to show missing data.

`offset` `offset` controls the size of the 'hole' in the middle and is expressed on a scale of `0` to `100`, where `0` is no hole and `100` is a hole that takes up the entire plotting area.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

`min.bin` The minimum number of points allowed in a wind speed/wind direction bin. The default is 1. A value of two requires at least 2 valid records in each bin an so on; bins with less than 2 valid records are set to NA. Care should be taken when using a value > 1 because of the risk of removing real data points. It is recommended to consider your data with care. Also, the `polarFreq` function can be of use in such circumstances.

`exclude.missing`

Setting this option to `TRUE` (the default) removes points from the plot that are too far from the original data. The smoothing routines will produce predictions at points where no data exist i.e. they predict. By removing the points too far from the original data produces a plot where it is clear where the original data lie. If set to `FALSE` missing data will be interpolated.

`date.pad` For `type = "trend"` (default), `date.pad = TRUE` will pad-out missing data to the beginning of the first year and the end of the last year. The purpose is to ensure that the trend plot begins and ends at the beginning or end of year.

`force.positive` The default is `TRUE`. Sometimes if smoothing data with steep gradients it is possible for predicted values to be negative. `force.positive = TRUE` ensures that predictions remain positive. This is useful for several reasons. First, with lots

of missing data more interpolation is needed and this can result in artefacts because the predictions are too far from the original data. Second, if it is known beforehand that the data are all positive, then this option carries that assumption through to the prediction. The only likely time where setting `force.positive = FALSE` would be if background concentrations were first subtracted resulting in data that is legitimately negative. For the vast majority of situations it is expected that the user will not need to alter the default option.

`k` The smoothing value supplied to `gam` for the temporal and wind direction components, respectively. In some cases e.g. a trend plot with less than 1-year of data the smoothing with the default values may become too noisy and affected more by outliers. Choosing a lower value of `k` (say 10) may help produce a better plot.

`normalise` If `TRUE` concentrations are normalised by dividing by their mean value. This is done *after* fitting the smooth surface. This option is particularly useful if one is interested in the patterns of concentrations for several pollutants on different scales e.g. NOx and CO. Often useful if more than one `pollutant` is chosen.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `annotate = FALSE` will not plot the N/E/S/W labels.

Details

The `polarAnnulus` function shares many of the properties of the `polarPlot`. However, `polarAnnulus` is focussed on displaying information on how concentrations of a pollutant (values of another variable) vary with wind direction and time. Plotting as an annulus helps to reduce compression of

information towards the centre of the plot. The circular plot is easy to interpret because wind direction is most easily understood in polar rather than Cartesian coordinates.

The inner part of the annulus represents the earliest time and the outer part of the annulus the latest time. The time dimension can be shown in many ways including "trend", "hour" (hour or day), "season" (month of the year) and "weekday" (day of the week). Taking hour as an example, the plot will show how concentrations vary by hour of the day and wind direction. Such plots can be very useful for understanding how different source influences affect a location.

For `type = "trend"` the amount of smoothing does not vary linearly with the length of the time series i.e. a certain amount of smoothing per unit interval in time. This is a deliberate choice because should one be interested in a subset (in time) of data, more detail will be provided for the subset compared with the full data set. This allows users to investigate specific periods in more detail. Full flexibility is given through the smoothing parameter `k`.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

See Also

Other polar directional analysis functions: `percentileRose()`, `polarCluster()`, `polarDiff()`, `polarFreq()`, `polarPlot()`, `pollutionRose()`, `windRose()`

Examples

```
# diurnal plot for PM10 at Marylebone Rd
## Not run:
polarAnnulus(mydata,
  pollutant = "pm10",
  main = "diurnal variation in pm10 at Marylebone Road"
)

## End(Not run)

# seasonal plot for PM10 at Marylebone Rd
## Not run:
polarAnnulus(mydata, poll = "pm10", period = "season")

## End(Not run)

# trend in coarse particles (PMc = PM10 - PM2.5), calculate PMc first

mydata$pmc <- mydata$pm10 - mydata$pm25
## Not run:
polarAnnulus(mydata,
  poll = "pmc", period = "trend",
```

```
  main = "trend in pmc at Marylebone Road"
)

## End(Not run)
```

`polarCluster` *K-means clustering of bivariate polar plots*

Description

Function for identifying clusters in bivariate polar plots (`polarPlot()`); identifying clusters in the original data for subsequent processing.

Usage

```
polarCluster(
  mydata,
  pollutant = "nox",
  x = "ws",
  wd = "wd",
  n.clusters = 6,
  after = NA,
  cols = "Paired",
  angle.scale = 315,
  units = x,
  auto.text = TRUE,
  plot = TRUE,
  plot.data = FALSE,
  ...
)
```

Arguments

`mydata`	A data frame minimally containing `wd`, another variable to plot in polar coordinates (the default is a column "ws" — wind speed) and a pollutant. Should also contain `date` if plots by time period are required.
`pollutant`	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. `pollutant = "nox"`. Only one pollutant can be chosen.
`x`	Name of variable to plot against wind direction in polar coordinates, the default is wind speed, "ws".
`wd`	Name of wind direction field.
`n.clusters`	Number of clusters to use. If `n.clusters` is more than length 1, then a faceted plot will be output showing the clusters identified for each one of `n.clusters`.
`after`	The function can be applied to differences between polar plot surfaces (see polarDiff for details). If an `after` data frame is supplied, the clustering will be carried out on the differences between `after` and `mydata` in the same way as polarDiff.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

`units` The units shown on the polar axis scale.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`plot.data` By default, the `data` component of `polarCluster()` contains the original data frame appended with a new "cluster" column. When `plot.data = TRUE`, the `data` component instead contains data to reproduce the clustered polar plot itself (similar to `data` returned by `polarPlot()`). This may be useful for re-plotting the `polarCluster()` plot in other ways.

`...` Arguments passed on to `polarPlot`

`statistic` The statistic that should be applied to each wind speed/direction bin. Because of the smoothing involved, the colour scale for some of these statistics is only to provide an indication of overall pattern and should not be interpreted in concentration units e.g. for `statistic = "weighted.mean"` where the bin mean is multiplied by the bin frequency and divided by the total frequency. In many cases using `polarFreq` will be better. Setting `statistic = "weighted.mean"` can be useful because it provides an indication of the concentration * frequency of occurrence and will highlight the wind speed/direction conditions that dominate the overall mean.Can be:

- “mean” (default), “median”, “max” (maximum), “frequency”. “stdev” (standard deviation), “weighted.mean”.
- `statistic = "nwr"` Implements the Non-parametric Wind Regression approach of Henry et al. (2009) that uses kernel smoothers. The `openair` implementation is not identical because Gaussian kernels are used for both wind direction and speed. The smoothing is controlled by `ws_spread` and `wd_spread`.
- `statistic = "cpf"` the conditional probability function (CPF) is plotted and a single (usually high) percentile level is supplied. The CPF is defined as CPF = my/ny, where my is the number of samples in the y bin (by default a wind direction, wind speed interval) with mixing ratios greater than the *overall* percentile concentration, and ny is the total number of samples in the same wind sector (see Ashbaugh et al., 1985).

Note that percentile intervals can also be considered; see `percentile` for details.

- When `statistic = "r"` or `statistic = "Pearson"`, the Pearson correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Pearson correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed). More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.
- When `statistic = "Spearman"`, the Spearman correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Spearman correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed). More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.
- `"robust_slope"` is another option for pair-wise statistics and `"quantile.slope"`, which uses quantile regression to estimate the slope for a particular quantile level (see also `tau` for setting the quantile level).
- `"york_slope"` is another option for pair-wise statistics which uses the *York regression method* to estimate the slope. In this method the uncertainties in `x` and `y` are used in the determination of the slope. The uncertainties are provided by `x_error` and `y_error` — see below.

`limits` The function does its best to choose sensible limits automatically. However, there are circumstances when the user will wish to set different ones. An example would be a series of plots showing each year of data separately. The limits are set in the form `c(lower, upper)`, so `limits = c(0, 100)` would force the plot limits to span 0-100.

`exclude.missing` Setting this option to `TRUE` (the default) removes points from the plot that are too far from the original data. The smoothing routines will produce predictions at points where no data exist i.e. they predict. By removing the points too far from the original data produces a plot where it is clear where the original data lie. If set to `FALSE` missing data will be interpolated.

`uncertainty` Should the uncertainty in the calculated surface be shown? If `TRUE` three plots are produced on the same scale showing the predicted surface together with the estimated lower and upper uncertainties at the 95% confidence interval. Calculating the uncertainties is useful to understand whether features are real or not. For example, at high wind speeds where there are few data there is greater uncertainty over the predicted values. The uncertainties are calculated using the GAM and weighting is done by the frequency of measurements in each wind speed-direction bin. Note that if uncertainties are calculated then the type is set to "default".

`percentile` If `statistic = "percentile"` then `percentile` is used, expressed from 0 to 100. Note that the percentile value is calculated in the wind speed,

wind direction 'bins'. For this reason it can also be useful to set `min.bin` to ensure there are a sufficient number of points available to estimate a percentile. See `quantile` for more details of how percentiles are calculated.
`percentile` is also used for the Conditional Probability Function (CPF) plots. `percentile` can be of length two, in which case the percentile *interval* is considered for use with CPF. For example, `percentile = c(90, 100)` will plot the CPF for concentrations between the 90 and 100th percentiles. Percentile intervals can be useful for identifying specific sources. In addition, `percentile` can also be of length 3. The third value is the 'trim' value to be applied. When calculating percentile intervals many can cover very low values where there is no useful information. The trim value ensures that values greater than or equal to the trim * mean value are considered *before* the percentile intervals are calculated. The effect is to extract more detail from many source signatures. See the manual for examples. Finally, if the trim value is less than zero the percentile range is interpreted as absolute concentration values and subsetting is carried out directly.

`weights` At the edges of the plot there may only be a few data points in each wind speed-direction interval, which could in some situations distort the plot if the concentrations are high. `weights` applies a weighting to reduce their influence. For example and by default if only a single data point exists then the weighting factor is 0.25 and for two points 0.5. To not apply any weighting and use the data as is, use `weights = c(1, 1, 1)`.
An alternative to down-weighting these points they can be removed altogether using `min.bin`.

`min.bin` The minimum number of points allowed in a wind speed/wind direction bin. The default is 1. A value of two requires at least 2 valid records in each bin an so on; bins with less than 2 valid records are set to NA. Care should be taken when using a value > 1 because of the risk of removing real data points. It is recommended to consider your data with care. Also, the `polarFreq` function can be of use in such circumstances.

`mis.col` When `min.bin` is > 1 it can be useful to show where data are removed on the plots. This is done by shading the missing data in `mis.col`. To not highlight missing data when `min.bin` > 1 choose `mis.col = "transparent"`.

`upper` This sets the upper limit wind speed to be used. Often there are only a relatively few data points at very high wind speeds and plotting all of them can reduce the useful information in the plot.

`force.positive` The default is `TRUE`. Sometimes if smoothing data with steep gradients it is possible for predicted values to be negative. `force.positive = TRUE` ensures that predictions remain positive. This is useful for several reasons. First, with lots of missing data more interpolation is needed and this can result in artefacts because the predictions are too far from the original data. Second, if it is known beforehand that the data are all positive, then this option carries that assumption through to the prediction. The only likely time where setting `force.positive = FALSE` would be if background concentrations were first subtracted resulting in data that is legitimately negative. For the vast majority of situations it is expected that the user will not need to alter the default option.

`k` This is the smoothing parameter used by the `gam` function in package `mgcv`.

Typically, value of around 100 (the default) seems to be suitable and will resolve important features in the plot. The most appropriate choice of k is problem-dependent; but extensive testing of polar plots for many different problems suggests a value of k of about 100 is suitable. Setting k to higher values will not tend to affect the surface predictions by much but will add to the computation time. Lower values of k will increase smoothing. Sometimes with few data to plot polarPlot will fail. Under these circumstances it can be worth lowering the value of k.

normalise If TRUE concentrations are normalised by dividing by their mean value. This is done *after* fitting the smooth surface. This option is particularly useful if one is interested in the patterns of concentrations for several pollutants on different scales e.g. NOx and CO. Often useful if more than one pollutant is chosen.

ws_spread The value of sigma used for Gaussian kernel weighting of wind speed when statistic = "nwr" or when correlation and regression statistics are used such as *r*. Default is 0.5.

wd_spread The value of sigma used for Gaussian kernel weighting of wind direction when statistic = "nwr" or when correlation and regression statistics are used such as *r*. Default is 4.

x_error The x error / uncertainty used when statistic = "york_slope".

y_error The y error / uncertainty used when statistic = "york_slope".

kernel Type of kernel used for the weighting procedure for when correlation or regression techniques are used. Only "gaussian" is supported but this may be enhanced in the future.

formula.label When pair-wise statistics such as regression slopes are calculated and plotted, should a formula label be displayed? formula.label will also determine whether concentration information is printed when statistic = "cpf".

tau The quantile to be estimated when statistic is set to "quantile.slope". Default is 0.5 which is equal to the median and will be ignored if "quantile.slope" is not used.

type Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

key.position Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

key.title Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.

strip.position Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

key Deprecated; please use key.position. If FALSE, sets key.position to "none".

Details

Bivariate polar plots generated using the polarPlot function provide a very useful graphical technique for identifying and characterising different air pollution sources. While bivariate polar plots provide a useful graphical indication of potential sources, their location and wind-speed or other variable dependence, they do have several limitations. Often, a 'feature' will be detected in a plot but the subsequent analysis of data meeting particular wind speed/direction criteria will be based only on the judgement of the investigator concerning the wind speed-direction intervals of interest. Furthermore, the identification of a feature can depend on the choice of the colour scale used, making the process somewhat arbitrary.

polarCluster applies Partition Around Medoids (PAM) clustering techniques to polarPlot() surfaces to help identify potentially interesting features for further analysis. Details of PAM can be found in the cluster package (a core R package that will be pre-installed on all R systems). PAM clustering is similar to k-means but has several advantages e.g. is more robust to outliers. The clustering is based on the equal contribution assumed from the u and v wind components and the associated concentration. The data are standardized before clustering takes place.

The function works best by first trying different numbers of clusters and plotting them. This is achieved by setting n.clusters to be of length more than 1. For example, if n.clusters = 2:10 then a plot will be output showing the 9 cluster levels 2 to 10.

The clustering can also be applied to differences in polar plot surfaces (see polarDiff()). On this case a second data frame (after) should be supplied.

Note that clustering is computationally intensive and the function can take a long time to run — particularly when the number of clusters is increased. For this reason it can be a good idea to run a few clusters first to get a feel for it e.g. n.clusters = 2:5.

Once the number of clusters has been decided, the user can then run polarCluster to return the original data frame together with a new column cluster, which gives the cluster number as a character (see example). Note that any rows where the value of pollutant is NA are ignored so that the returned data frame may have fewer rows than the original.

Note that there are no automatic ways in ensuring the most appropriate number of clusters as this is application dependent. However, there is often a-priori information available on what different features in polar plots correspond to. Nevertheless, the appropriateness of different clusters is best

determined by post-processing the data. The Carslaw and Beevers (2012) paper discusses these issues in more detail.

Note that unlike most other openair functions only a single type "default" is allowed.

Value

an openair object. The object includes four main components: call, the command used to generate the plot; data, by default the original data frame with a new field cluster identifying the cluster, clust_stats giving the contributions made by each cluster to number of measurements, their percentage and the percentage by pollutant; and plot, the plot itself. Note that any rows where the value of pollutant is NA are ignored so that the returned data frame may have fewer rows than the original.

If the clustering is carried out considering differences, i.e., an after data frame is supplied, the output also includes the after data frame with cluster identified.

Author(s)

David Carslaw

References

Carslaw, D.C., Beevers, S.D, Ropkins, K and M.C. Bell (2006). Detecting and quantifying aircraft and other on-airport contributions to ambient nitrogen oxides in the vicinity of a large international airport. Atmospheric Environment. 40/28 pp 5424-5434.

Carslaw, D.C., & Beevers, S.D. (2013). Characterising and understanding emission sources using bivariate polar plots and k-means clustering. Environmental Modelling & Software, 40, 325-329. doi:10.1016/j.envsoft.2012.09.005

See Also

Other polar directional analysis functions: percentileRose(), polarAnnulus(), polarDiff(), polarFreq(), polarPlot(), pollutionRose(), windRose()

Other cluster analysis functions: timeProp(), trajCluster()

Examples

```
## Not run:
# plot 2-8 clusters. Warning! This can take several minutes...
polarCluster(mydata, pollutant = "nox", n.clusters = 2:8)

# basic plot with 6 clusters
results <- polarCluster(mydata, pollutant = "nox", n.clusters = 6)

# get results, could read into a new data frame to make it easier to refer to
# e.g. results <- results$data...
head(results$data)

# how many points are there in each cluster?
table(results$data$cluster)
```

```
# plot clusters 3 and 4 as a timeVariation plot using SAME colours as in
# cluster plot
timeVariation(subset(results$data, cluster %in% c("3", "4")),
  pollutant = "nox",
  group = "cluster", col = openColours("Paired", 6)[c(3, 4)]
)

## End(Not run)
```

`polarDiff` *Polar plots considering changes in concentrations between two time periods*

Description

This function provides a way of showing the differences in concentrations between two time periods as a polar plot. There are several uses of this function, but the most common will be to see how source(s) may have changed between two periods.

Usage

```
polarDiff(
  before,
  after,
  pollutant = "nox",
  type = "default",
  x = "ws",
  limits = NULL,
  auto.text = TRUE,
  plot = TRUE,
  ...
)
```

Arguments

`before, after`	Data frames representing the "before" and "after" cases. See `polarPlot()` for details of different input requirements.
`pollutant`	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. `pollutant = "nox"`. There can also be more than one pollutant specified e.g. `pollutant = c("nox", "no2")`. The main use of using two or more pollutants is for model evaluation where two species would be expected to have similar concentrations. This saves the user stacking the data and it is possible to work with columns of data directly. A typical use would be `pollutant = c("obs", "mod")` to compare two columns "obs" (the observations) and "mod" (modelled values). When pair-wise statistics such as Pearson correlation and regression techniques are to be plotted, `pollutant` takes two elements too. For example, `pollutant = c("bc", "pm25")` where `"bc"` is a function of `"pm25"`.

type Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

x Name of variable to plot against wind direction in polar coordinates, the default is wind speed, "ws".

limits The function does its best to choose sensible limits automatically. However, there are circumstances when the user will wish to set different ones. An example would be a series of plots showing each year of data separately. The limits are set in the form c(lower, upper), so limits = c(0, 100) would force the plot limits to span 0-100.

auto.text Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().

plot When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

... Arguments passed on to polarPlot

wd Name of wind direction field.

statistic The statistic that should be applied to each wind speed/direction bin. Because of the smoothing involved, the colour scale for some of these statistics is only to provide an indication of overall pattern and should not be interpreted in concentration units e.g. for statistic = "weighted.mean" where the bin mean is multiplied by the bin frequency and divided by the total frequency. In many cases using polarFreq will be better. Setting statistic = "weighted.mean" can be useful because it provides an indication of the concentration * frequency of occurrence and will highlight the wind speed/direction conditions that dominate the overall mean.Can be:

- "mean" (default), "median", "max" (maximum), "frequency". "stdev" (standard deviation), "weighted.mean".
- statistic = "nwr" Implements the Non-parametric Wind Regression approach of Henry et al. (2009) that uses kernel smoothers. The

openair implementation is not identical because Gaussian kernels are used for both wind direction and speed. The smoothing is controlled by ws_spread and wd_spread.

- statistic = "cpf" the conditional probability function (CPF) is plotted and a single (usually high) percentile level is supplied. The CPF is defined as CPF = my/ny, where my is the number of samples in the y bin (by default a wind direction, wind speed interval) with mixing ratios greater than the *overall* percentile concentration, and ny is the total number of samples in the same wind sector (see Ashbaugh et al., 1985). Note that percentile intervals can also be considered; see percentile for details.
- When statistic = "r" or statistic = "Pearson", the Pearson correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Pearson correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed). More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.
- When statistic = "Spearman", the Spearman correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Spearman correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed). More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.
- "robust_slope" is another option for pair-wise statistics and "quantile.slope", which uses quantile regression to estimate the slope for a particular quantile level (see also tau for setting the quantile level).
- "york_slope" is another option for pair-wise statistics which uses the *York regression method* to estimate the slope. In this method the uncertainties in x and y are used in the determination of the slope. The uncertainties are provided by x_error and y_error — see below.

exclude.missing Setting this option to TRUE (the default) removes points from the plot that are too far from the original data. The smoothing routines will produce predictions at points where no data exist i.e. they predict. By removing the points too far from the original data produces a plot where it is clear where the original data lie. If set to FALSE missing data will be interpolated.

uncertainty Should the uncertainty in the calculated surface be shown? If TRUE three plots are produced on the same scale showing the predicted surface together with the estimated lower and upper uncertainties at the 95% confidence interval. Calculating the uncertainties is useful to understand whether features are real or not. For example, at high wind speeds where there are few data there is greater uncertainty over the predicted values. The uncertainties are calculated using the GAM and weighting is done by the

frequency of measurements in each wind speed-direction bin. Note that if uncertainties are calculated then the type is set to "default".

`percentile` If `statistic = "percentile"` then `percentile` is used, expressed from 0 to 100. Note that the percentile value is calculated in the wind speed, wind direction 'bins'. For this reason it can also be useful to set `min.bin` to ensure there are a sufficient number of points available to estimate a percentile. See `quantile` for more details of how percentiles are calculated. `percentile` is also used for the Conditional Probability Function (CPF) plots. `percentile` can be of length two, in which case the percentile *interval* is considered for use with CPF. For example, `percentile = c(90, 100)` will plot the CPF for concentrations between the 90 and 100th percentiles. Percentile intervals can be useful for identifying specific sources. In addition, `percentile` can also be of length 3. The third value is the 'trim' value to be applied. When calculating percentile intervals many can cover very low values where there is no useful information. The trim value ensures that values greater than or equal to the trim * mean value are considered *before* the percentile intervals are calculated. The effect is to extract more detail from many source signatures. See the manual for examples. Finally, if the trim value is less than zero the percentile range is interpreted as absolute concentration values and subsetting is carried out directly.

`weights` At the edges of the plot there may only be a few data points in each wind speed-direction interval, which could in some situations distort the plot if the concentrations are high. `weights` applies a weighting to reduce their influence. For example and by default if only a single data point exists then the weighting factor is 0.25 and for two points 0.5. To not apply any weighting and use the data as is, use `weights = c(1, 1, 1)`.
An alternative to down-weighting these points they can be removed altogether using `min.bin`.

`min.bin` The minimum number of points allowed in a wind speed/wind direction bin. The default is 1. A value of two requires at least 2 valid records in each bin an so on; bins with less than 2 valid records are set to NA. Care should be taken when using a value > 1 because of the risk of removing real data points. It is recommended to consider your data with care. Also, the `polarFreq` function can be of use in such circumstances.

`mis.col` When `min.bin` is > 1 it can be useful to show where data are removed on the plots. This is done by shading the missing data in `mis.col`. To not highlight missing data when `min.bin` > 1 choose `mis.col = "transparent"`.

`upper` This sets the upper limit wind speed to be used. Often there are only a relatively few data points at very high wind speeds and plotting all of them can reduce the useful information in the plot.

`units` The units shown on the polar axis scale.

`force.positive` The default is `TRUE`. Sometimes if smoothing data with steep gradients it is possible for predicted values to be negative. `force.positive = TRUE` ensures that predictions remain positive. This is useful for several reasons. First, with lots of missing data more interpolation is needed and this can result in artefacts because the predictions are too far from the original data. Second, if it is known beforehand that the data are all positive, then this option carries that assumption through to the prediction. The

only likely time where setting `force.positive = FALSE` would be if background concentrations were first subtracted resulting in data that is legitimately negative. For the vast majority of situations it is expected that the user will not need to alter the default option.

`k` This is the smoothing parameter used by the `gam` function in package `mgcv`. Typically, value of around 100 (the default) seems to be suitable and will resolve important features in the plot. The most appropriate choice of `k` is problem-dependent; but extensive testing of polar plots for many different problems suggests a value of `k` of about 100 is suitable. Setting `k` to higher values will not tend to affect the surface predictions by much but will add to the computation time. Lower values of `k` will increase smoothing. Sometimes with few data to plot `polarPlot` will fail. Under these circumstances it can be worth lowering the value of `k`.

`normalise` If `TRUE` concentrations are normalised by dividing by their mean value. This is done *after* fitting the smooth surface. This option is particularly useful if one is interested in the patterns of concentrations for several pollutants on different scales e.g. NOx and CO. Often useful if more than one `pollutant` is chosen.

`ws_spread` The value of sigma used for Gaussian kernel weighting of wind speed when `statistic = "nwr"` or when correlation and regression statistics are used such as *r*. Default is `0.5`.

`wd_spread` The value of sigma used for Gaussian kernel weighting of wind direction when `statistic = "nwr"` or when correlation and regression statistics are used such as *r*. Default is `4`.

`x_error` The `x` error / uncertainty used when `statistic = "york_slope"`.

`y_error` The `y` error / uncertainty used when `statistic = "york_slope"`.

`kernel` Type of kernel used for the weighting procedure for when correlation or regression techniques are used. Only `"gaussian"` is supported but this may be enhanced in the future.

`formula.label` When pair-wise statistics such as regression slopes are calculated and plotted, should a formula label be displayed? `formula.label` will also determine whether concentration information is printed when `statistic = "cpf"`.

`tau` The quantile to be estimated when `statistic` is set to `"quantile.slope"`. Default is `0.5` which is equal to the median and will be ignored if `"quantile.slope"` is not used.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

key.position Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

key.title Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.

strip.position Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

key Deprecated; please use key.position. If FALSE, sets key.position to "none".

Details

While the function is primarily intended to compare two time periods at the same location, it can be used for any two data sets that contain the same pollutant. For example, data from two sites that are close to one another, or two co-located instruments.

The analysis works by calculating the polar plot surface for the before and after periods and then subtracting the before surface from the after surface.

Value

an openair plot.

See Also

Other polar directional analysis functions: percentileRose(), polarAnnulus(), polarCluster(), polarFreq(), polarPlot(), pollutionRose(), windRose()

Examples

```
## Not run:
before_data <- selectByDate(mydata, year = 2002)
after_data <- selectByDate(mydata, year = 2003)

polarDiff(before_data, after_data, pollutant = "no2")

# with some options
polarDiff(
  before_data,
  after_data,
  pollutant = "no2",
  cols = "RdYlBu",
  limits = c(-20, 20)
)

## End(Not run)
```

polarFreq *Function to plot wind speed/direction frequencies and other statistics*

Description

polarFreq primarily plots wind speed-direction frequencies in 'bins'. Each bin is colour-coded depending on the frequency of measurements. Bins can also be used to show the concentration of pollutants using a range of commonly used statistics.

Usage

```
polarFreq(
  mydata,
  pollutant = NULL,
  statistic = "frequency",
  ws.int = 1,
  wd.nint = 36,
  grid.line = 5,
  breaks = NULL,
  labels = NULL,
  cols = "default",
  trans = TRUE,
  type = "default",
  min.bin = 1,
  ws.upper = NA,
  offset = 10,
  border.col = "transparent",
  key.title = paste(statistic, pollutant, sep = " "),
  key.position = "right",
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame minimally containing ws, wd and date.
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox"
statistic	The statistic that should be applied to each wind speed/direction bin. Can be one of: • "frequency": the simplest and plots the frequency of wind speed/direction in different bins. The scale therefore shows the counts in each bin.

- "mean", "median", "max" (maximum), "stdev" (standard deviation): Plots the relevant summary statistic of a pollutant in wind speed/direction bins.
- "weighted.mean" will plot the concentration of a pollutant weighted by wind speed/direction. Each segment therefore provides the percentage overall contribution to the total concentration.

Note that for options other than "frequency", it is necessary to also provide the name of a pollutant.

ws.int — Wind speed interval assumed. In some cases e.g. a low met mast, an interval of 0.5 may be more appropriate.

wd.nint — Number of intervals of wind direction.

grid.line — Radial spacing of grid lines.

breaks, labels — If a categorical colour scale is required then breaks should be specified. These should be provided as a numeric vector, e.g., breaks = c(0, 50, 100, 1000). Users should set the maximum value of breaks to exceed the maximum data value to ensure it is within the maximum final range, e.g., 100–1000 in this case. Labels will automatically be generated, but can be customised by passing a character vector to labels, e.g., labels = c("good", "bad", "very bad"). In this example, 0 - 50 will be "good" and so on. Note there is one less label than break.

cols — Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

trans — Should a transformation be applied? Sometimes when producing plots of this kind they can be dominated by a few high points. The default therefore is TRUE and a square-root transform is applied. This results in a non-linear scale and (usually) a better representation of the distribution. If set to FALSE a linear scale is used.

type — Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

`min.bin` The minimum number of points allowed in a wind speed/wind direction bin. The default is 1. A value of two requires at least 2 valid records in each bin an so on; bins with less than 2 valid records are set to NA. Care should be taken when using a value > 1 because of the risk of removing real data points. It is recommended to consider your data with care. Also, the `polarFreq` function can be of use in such circumstances.

`ws.upper` A user-defined upper wind speed to use. This is useful for ensuring a consistent scale between different plots. For example, to always ensure that wind speeds are displayed between 1-10, set `ws.int = 10`.

`offset` `offset` controls the size of the 'hole' in the middle and is expressed on a scale of `0` to `100`, where `0` is no hole and `100` is a hole that takes up the entire plotting area.

`border.col` The colour of the boundary of each wind speed/direction bin. The default is transparent. Another useful choice sometimes is "white".

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `annotate = FALSE` will not plot the N/E/S/W labels.
- `limits` sets the colour bar limits, if `breaks` is not used.

Details

`polarFreq` is its default use provides details of wind speed and direction frequencies. In this respect it is similar to `windRose()`, but considers wind direction intervals of 10 degrees and a user-specified

wind speed interval. The frequency of wind speeds/directions formed by these 'bins' is represented on a colour scale.

The `polarFreq` function is more flexible than either `windRose()` or `polarPlot()`. It can, for example, also consider pollutant concentrations (see examples below). Instead of the number of data points in each bin, the concentration can be shown. Further, a range of statistics can be used to describe each bin - see `statistic` above. Plotting mean concentrations is useful for source identification and is the same as `polarPlot()` but without smoothing, which may be preferable for some data. Plotting with `statistic = "weighted.mean"` is particularly useful for understanding the relative importance of different source contributions. For example, high mean concentrations may be observed for high wind speed conditions, but the weighted mean concentration may well show that the contribution to overall concentrations is very low.

`polarFreq` also offers great flexibility with the scale used and the user has fine control over both the range, interval and colour.

Value

an openair object

Author(s)

David Carslaw

See Also

Other polar directional analysis functions: `percentileRose()`, `polarAnnulus()`, `polarCluster()`, `polarDiff()`, `polarPlot()`, `pollutionRose()`, `windRose()`

Examples

```
# basic wind frequency plot
polarFreq(mydata)

# wind frequencies by year
## Not run:
polarFreq(mydata, type = "year")

## End(Not run)


# mean SO2 by year, showing only bins with at least 2 points
## Not run:
polarFreq(mydata, pollutant = "so2", type = "year", statistic = "mean", min.bin = 2)

## End(Not run)

# weighted mean SO2 by year, showing only bins with at least 2 points
## Not run:
polarFreq(mydata,
  pollutant = "so2", type = "year", statistic = "weighted.mean",
  min.bin = 2
)
```

```
## End(Not run)

# windRose for just 2000 and 2003 with different colours
## Not run:
polarFreq(subset(mydata, format(date, "%Y") %in% c(2000, 2003)),
  type = "year", cols = "turbo"
)

## End(Not run)

# user defined breaks from 0-700 in intervals of 100 (note linear scale)
## Not run:
polarFreq(mydata, breaks = seq(0, 700, 100))

## End(Not run)

# more complicated user-defined breaks - useful for highlighting bins
# with a certain number of data points
## Not run:
polarFreq(mydata, breaks = c(0, 10, 50, 100, 250, 500, 700))

## End(Not run)

# source contribution plot and use of offset option
## Not run:
polarFreq(mydata,
  pollutant = "pm25",
  statistic = "weighted.mean", offset = 50, ws.int = 25, trans = FALSE
)

## End(Not run)
```

`polarPlot` *Function for plotting bivariate polar plots with smoothing.*

Description

Function for plotting pollutant concentration in polar coordinates showing concentration by wind speed (or another numeric variable) and direction. Mean concentrations are calculated for wind speed-direction 'bins' (e.g. 0-1, 1-2 m/s,... and 0-10, 10-20 degrees etc.). To aid interpretation, `gam` smoothing is carried out using `mgcv`.

Usage

```
polarPlot(
  mydata,
  pollutant = "nox",
  x = "ws",
  wd = "wd",
```

```
  type = "default",
  statistic = "mean",
  limits = NULL,
  exclude.missing = TRUE,
  uncertainty = FALSE,
  percentile = NA,
  cols = "default",
  weights = c(0.25, 0.5, 0.75),
  min.bin = 1,
  mis.col = "grey",
  upper = NA,
  angle.scale = 315,
  units = x,
  force.positive = TRUE,
  k = 100,
  normalise = FALSE,
  key.title = paste(statistic, pollutant, sep = " "),
  key.position = "right",
  strip.position = "top",
  auto.text = TRUE,
  ws_spread = 1.5,
  wd_spread = 5,
  x_error = NA,
  y_error = NA,
  kernel = "gaussian",
  formula.label = TRUE,
  tau = 0.5,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame minimally containing wd, another variable to plot in polar coordinates (the default is a column "ws" — wind speed) and a pollutant. Should also contain date if plots by time period are required.
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox". There can also be more than one pollutant specified e.g. pollutant = c("nox", "no2"). The main use of using two or more pollutants is for model evaluation where two species would be expected to have similar concentrations. This saves the user stacking the data and it is possible to work with columns of data directly. A typical use would be pollutant = c("obs", "mod") to compare two columns "obs" (the observations) and "mod" (modelled values). When pair-wise statistics such as Pearson correlation and regression techniques are to be plotted, pollutant takes two elements too. For example, pollutant = c("bc", "pm25") where "bc" is a function of "pm25".
x	Name of variable to plot against wind direction in polar coordinates, the default

is wind speed, "ws".

wd — Name of wind direction field.

type — Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

statistic — The statistic that should be applied to each wind speed/direction bin. Because of the smoothing involved, the colour scale for some of these statistics is only to provide an indication of overall pattern and should not be interpreted in concentration units e.g. for statistic = "weighted.mean" where the bin mean is multiplied by the bin frequency and divided by the total frequency. In many cases using polarFreq will be better. Setting statistic = "weighted.mean" can be useful because it provides an indication of the concentration * frequency of occurrence and will highlight the wind speed/direction conditions that dominate the overall mean.Can be:

- "mean" (default), "median", "max" (maximum), "frequency". "stdev" (standard deviation), "weighted.mean".
- statistic = "nwr" Implements the Non-parametric Wind Regression approach of Henry et al. (2009) that uses kernel smoothers. The openair implementation is not identical because Gaussian kernels are used for both wind direction and speed. The smoothing is controlled by ws_spread and wd_spread.
- statistic = "cpf" the conditional probability function (CPF) is plotted and a single (usually high) percentile level is supplied. The CPF is defined as CPF = my/ny, where my is the number of samples in the y bin (by default a wind direction, wind speed interval) with mixing ratios greater than the *overall* percentile concentration, and ny is the total number of samples in the same wind sector (see Ashbaugh et al., 1985). Note that percentile intervals can also be considered; see percentile for details.
- When statistic = "r" or statistic = "Pearson", the Pearson correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Pearson correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed).

More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.

- When `statistic = "Spearman"`, the Spearman correlation coefficient is calculated for *two* pollutants. The calculation involves a weighted Spearman correlation coefficient, which is weighted by Gaussian kernels for wind direction an the radial variable (by default wind speed). More weight is assigned to values close to a wind speed-direction interval. Kernel weighting is used to ensure that all data are used rather than relying on the potentially small number of values in a wind speed-direction interval.
- `"robust_slope"` is another option for pair-wise statistics and `"quantile.slope"`, which uses quantile regression to estimate the slope for a particular quantile level (see also `tau` for setting the quantile level).
- `"york_slope"` is another option for pair-wise statistics which uses the *York regression method* to estimate the slope. In this method the uncertainties in `x` and `y` are used in the determination of the slope. The uncertainties are provided by `x_error` and `y_error` — see below.

`limits` The function does its best to choose sensible limits automatically. However, there are circumstances when the user will wish to set different ones. An example would be a series of plots showing each year of data separately. The limits are set in the form `c(lower, upper)`, so `limits = c(0, 100)` would force the plot limits to span 0-100.

`exclude.missing`

Setting this option to `TRUE` (the default) removes points from the plot that are too far from the original data. The smoothing routines will produce predictions at points where no data exist i.e. they predict. By removing the points too far from the original data produces a plot where it is clear where the original data lie. If set to `FALSE` missing data will be interpolated.

`uncertainty` Should the uncertainty in the calculated surface be shown? If `TRUE` three plots are produced on the same scale showing the predicted surface together with the estimated lower and upper uncertainties at the 95% confidence interval. Calculating the uncertainties is useful to understand whether features are real or not. For example, at high wind speeds where there are few data there is greater uncertainty over the predicted values. The uncertainties are calculated using the GAM and weighting is done by the frequency of measurements in each wind speed-direction bin. Note that if uncertainties are calculated then the type is set to "default".

`percentile` If `statistic = "percentile"` then `percentile` is used, expressed from 0 to 100. Note that the percentile value is calculated in the wind speed, wind direction 'bins'. For this reason it can also be useful to set `min.bin` to ensure there are a sufficient number of points available to estimate a percentile. See `quantile` for more details of how percentiles are calculated.

`percentile` is also used for the Conditional Probability Function (CPF) plots. `percentile` can be of length two, in which case the percentile *interval* is considered for use with CPF. For example, `percentile = c(90, 100)` will plot the CPF for concentrations between the 90 and 100th percentiles. Percentile intervals can be useful for identifying specific sources. In addition, `percentile` can

also be of length 3. The third value is the 'trim' value to be applied. When calculating percentile intervals many can cover very low values where there is no useful information. The trim value ensures that values greater than or equal to the trim * mean value are considered *before* the percentile intervals are calculated. The effect is to extract more detail from many source signatures. See the manual for examples. Finally, if the trim value is less than zero the percentile range is interpreted as absolute concentration values and subsetting is carried out directly.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`weights` At the edges of the plot there may only be a few data points in each wind speed-direction interval, which could in some situations distort the plot if the concentrations are high. `weights` applies a weighting to reduce their influence. For example and by default if only a single data point exists then the weighting factor is 0.25 and for two points 0.5. To not apply any weighting and use the data as is, use `weights = c(1, 1, 1)`.

An alternative to down-weighting these points they can be removed altogether using `min.bin`.

`min.bin` The minimum number of points allowed in a wind speed/wind direction bin. The default is 1. A value of two requires at least 2 valid records in each bin an so on; bins with less than 2 valid records are set to NA. Care should be taken when using a value > 1 because of the risk of removing real data points. It is recommended to consider your data with care. Also, the `polarFreq` function can be of use in such circumstances.

`mis.col` When `min.bin` is > 1 it can be useful to show where data are removed on the plots. This is done by shading the missing data in `mis.col`. To not highlight missing data when `min.bin` > 1 choose `mis.col = "transparent"`.

`upper` This sets the upper limit wind speed to be used. Often there are only a relatively few data points at very high wind speeds and plotting all of them can reduce the useful information in the plot.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

`units` The units shown on the polar axis scale.

`force.positive` The default is `TRUE`. Sometimes if smoothing data with steep gradients it is possible for predicted values to be negative. `force.positive = TRUE` ensures that predictions remain positive. This is useful for several reasons. First, with lots of missing data more interpolation is needed and this can result in artefacts because the predictions are too far from the original data. Second, if it is known beforehand that the data are all positive, then this option carries that assumption through to the prediction. The only likely time where setting `force.positive`

= FALSE would be if background concentrations were first subtracted resulting in data that is legitimately negative. For the vast majority of situations it is expected that the user will not need to alter the default option.

k — This is the smoothing parameter used by the gam function in package mgcv. Typically, value of around 100 (the default) seems to be suitable and will resolve important features in the plot. The most appropriate choice of k is problem-dependent; but extensive testing of polar plots for many different problems suggests a value of k of about 100 is suitable. Setting k to higher values will not tend to affect the surface predictions by much but will add to the computation time. Lower values of k will increase smoothing. Sometimes with few data to plot polarPlot will fail. Under these circumstances it can be worth lowering the value of k.

normalise — If TRUE concentrations are normalised by dividing by their mean value. This is done *after* fitting the smooth surface. This option is particularly useful if one is interested in the patterns of concentrations for several pollutants on different scales e.g. NOx and CO. Often useful if more than one pollutant is chosen.

key.title — Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.

key.position — Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

strip.position — Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

auto.text — Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().

ws_spread — The value of sigma used for Gaussian kernel weighting of wind speed when statistic = "nwr" or when correlation and regression statistics are used such as *r*. Default is 0.5.

wd_spread — The value of sigma used for Gaussian kernel weighting of wind direction when statistic = "nwr" or when correlation and regression statistics are used such as *r*. Default is 4.

x_error — The x error / uncertainty used when statistic = "york_slope".

y_error — The y error / uncertainty used when statistic = "york_slope".

kernel — Type of kernel used for the weighting procedure for when correlation or regression techniques are used. Only "gaussian" is supported but this may be enhanced in the future.

formula.label — When pair-wise statistics such as regression slopes are calculated and plotted, should a formula label be displayed? formula.label will also determine whether concentration information is printed when statistic = "cpf".

tau — The quantile to be estimated when statistic is set to "quantile.slope". Default is 0.5 which is equal to the median and will be ignored if "quantile.slope" is not used.

plot — When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

key — Deprecated; please use key.position. If FALSE, sets key.position to "none".

... — Addition options are passed on to cutData() for type handling. Some additional arguments are also available:

- xlab, ylab and main override the x-axis label, y-axis label, and plot title.
- layout sets the layout of facets - e.g., layout(2, 5) will have 2 columns and 5 rows.
- fontsize overrides the overall font size of the plot.
- annotate = FALSE will not plot the N/E/S/W labels.

Details

The bivariate polar plot is a useful diagnostic tool for quickly gaining an idea of potential sources. Wind speed is one of the most useful variables to use to separate source types (see references). For example, ground-level concentrations resulting from buoyant plumes from chimney stacks tend to peak under higher wind speed conditions. Conversely, ground-level, non-buoyant plumes such as from road traffic, tend to have highest concentrations under low wind speed conditions. Other sources such as from aircraft engines also show differing characteristics by wind speed.

The function has been developed to allow variables other than wind speed to be plotted with wind direction in polar coordinates. The key issue is that the other variable plotted against wind direction should be discriminating in some way. For example, temperature can help reveal high-level sources brought down to ground level in unstable atmospheric conditions, or show the effect a source emission dependent on temperature e.g. biogenic isoprene.

The plots can vary considerably depending on how much smoothing is done. The approach adopted here is based on the very flexible and capable mgcv package that uses *Generalized Additive Models*. While methods do exist to find an optimum level of smoothness, they are not necessarily useful. The principal aim of polarPlot is as a graphical analysis rather than for quantitative purposes. In this respect the smoothing aims to strike a balance between revealing interesting (real) features and overly noisy data. The defaults used in polarPlot() are based on the analysis of data from many different sources. More advanced users may wish to modify the code and adopt other smoothing approaches.

Various statistics are possible to consider e.g. mean, maximum, median. statistic = "max" is often useful for revealing sources. Pair-wise statistics between two pollutants can also be calculated.

The function can also be used to compare two pollutant species through a range of pair-wise statistics (see help on statistic) and Grange et al. (2016) (open-access publication link below).

Wind direction is split up into 10 degree intervals and the other variable (e.g. wind speed) 30 intervals. These 2D bins are then used to calculate the statistics.

These plots often show interesting features at higher wind speeds (see references below). For these conditions there can be very few measurements and therefore greater uncertainty in the calculation of the surface. There are several ways in which this issue can be tackled. First, it is possible to avoid smoothing altogether and use polarFreq(). Second, the effect of setting a minimum number of measurements in each wind speed-direction bin can be examined through min.bin. It is

possible that a single point at high wind speed conditions can strongly affect the surface prediction. Therefore, setting `min.bin = 3`, for example, will remove all wind speed-direction bins with fewer than 3 measurements *before* fitting the surface. Third, consider setting `uncertainty = TRUE`. This option will show the predicted surface together with upper and lower 95% confidence intervals, which take account of the frequency of measurements.

Value

an openair object. `data` contains four set columns: `cond`, conditioning based on `type`; `u` and `v`, the translational vectors based on `ws` and `wd`; and the local `pollutant` estimate.

Author(s)

David Carslaw

References

Ashbaugh, L.L., Malm, W.C., Sadeh, W.Z., 1985. A residence time probability analysis of sulfur concentrations at ground canyon national park. Atmospheric Environment 19 (8), 1263-1270.

Carslaw, D.C., Beevers, S.D, Ropkins, K and M.C. Bell (2006). Detecting and quantifying aircraft and other on-airport contributions to ambient nitrogen oxides in the vicinity of a large international airport. Atmospheric Environment. 40/28 pp 5424-5434.

Carslaw, D.C., & Beevers, S.D. (2013). Characterising and understanding emission sources using bivariate polar plots and k-means clustering. Environmental Modelling & Software, 40, 325-329. DOI: 10.1016/j.envsoft.2012.09.005.

Henry, R.C., Chang, Y.S., Spiegelman, C.H., 2002. Locating nearby sources of air pollution by nonparametric regression of atmospheric concentrations on wind direction. Atmospheric Environment 36 (13), 2237-2244.

Henry, R., Norris, G.A., Vedantham, R., Turner, J.R., 2009. Source region identification using Kernel smoothing. Environ. Sci. Technol. 43 (11), 4090e4097. DOI: 10.1021/es8011723.

Uria-Tellaetxe, I. and D.C. Carslaw (2014). Source identification using a conditional bivariate Probability function. Environmental Modelling & Software, Vol. 59, 1-9.

Westmoreland, E.J., N. Carslaw, D.C. Carslaw, A. Gillah and E. Bates (2007). Analysis of air quality within a street canyon using statistical and dispersion modelling techniques. Atmospheric Environment. Vol. 41(39), pp. 9195-9205.

Yu, K.N., Cheung, Y.P., Cheung, T., Henry, R.C., 2004. Identifying the impact of large urban airports on local air quality by nonparametric regression. Atmospheric Environment 38 (27), 4501-4507.

Grange, S. K., Carslaw, D. C., & Lewis, A. C. 2016. Source apportionment advances with bivariate polar plots, correlation, and regression techniques. Atmospheric Environment. 145, 128-134. DOI: 10.1016/j.atmosenv.2016.09.016.

See Also

Other polar directional analysis functions: `percentileRose()`, `polarAnnulus()`, `polarCluster()`, `polarDiff()`, `polarFreq()`, `pollutionRose()`, `windRose()`

Examples

```
# basic plot
polarPlot(mydata, pollutant = "nox")
## Not run:

# polarPlots by year on same scale
polarPlot(mydata, pollutant = "so2", type = "year", main = "polarPlot of so2")

# set minimum number of bins to be used to see if pattern remains similar
polarPlot(mydata, pollutant = "nox", min.bin = 3)

# plot by day of the week
polarPlot(mydata, pollutant = "pm10", type = "weekday")

# show the 95% confidence intervals in the surface fitting
polarPlot(mydata, pollutant = "so2", uncertainty = TRUE)


# Pair-wise statistics
# Pearson correlation
polarPlot(mydata, pollutant = c("pm25", "pm10"), statistic = "r")

# Robust regression slope, takes a bit of time
polarPlot(mydata, pollutant = c("pm25", "pm10"), statistic = "robust.slope")

# Least squares regression works too but it is not recommended, use robust
# regression
# polarPlot(mydata, pollutant = c("pm25", "pm10"), statistic = "slope")

## End(Not run)
```

`pollutionRose` *Pollution rose variation of the traditional wind rose plot*

Description

The traditional wind rose plot that plots wind speed and wind direction by different intervals. The pollution rose applies the same plot structure but substitutes other measurements, most commonly a pollutant time series, for wind speed.

Usage

```
pollutionRose(
  mydata,
  pollutant = "nox",
  key.title = pollutant,
  key.position = "right",
  breaks = 6,
```

```
  paddle = FALSE,
  seg = 0.9,
  normalise = FALSE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame containing fields ws and wd
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox".
key.title	Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.
key.position	Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.
breaks	Most commonly, the number of break points for pollutant concentrations. The default, 6, attempts to breaks the supplied data at approximately 6 sensible break points. However, breaks can also be used to set specific break points. For example, the argument breaks = c(0, 1, 10, 100) breaks the data into segments <1, 1-10, 10-100, >100.
paddle	Either TRUE or FALSE. If TRUE plots rose using 'paddle' style spokes. If FALSE plots rose using 'wedge' style spokes.
seg	seg determines with width of the segments. For example, seg = 0.5 will produce segments 0.5 * angle.
normalise	If TRUE each wind direction segment is normalised to equal one. This is useful for showing how the concentrations (or other parameters) contribute to each wind sector when the proportion of time the wind is from that direction is low. A line showing the probability that the wind directions is from a particular wind sector is also shown.
plot	When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).
key	Deprecated; please use key.position. If FALSE, sets key.position to "none".
...	Arguments passed on to windRose ws Name of the column representing wind speed. wd Name of the column representing wind direction. ws2,wd2 The user can supply a second set of wind speed and wind direction values with which the first can be compared. See pollutionRose() for more details. ws.int The Wind speed interval. Default is 2 m/s but for low met masts with low mean wind speeds a value of 1 or 0.5 m/s may be better.

`angle` Default angle of "spokes" is 30. Other potentially useful angles are 45 and 10. Note that the width of the wind speed interval may need adjusting using `width`.

`calm.thresh` By default, conditions are considered to be calm when the wind speed is zero. The user can set a different threshold for calms be setting `calm.thresh` to a higher value. For example, `calm.thresh = 0.5` will identify wind speeds **below** 0.5 as calm.

`bias.corr` When `angle` does not divide exactly into 360 a bias is introduced in the frequencies when the wind direction is already supplied rounded to the nearest 10 degrees, as is often the case. For example, if `angle = 22.5`, N, E, S, W will include 3 wind sectors and all other angles will be two. A bias correction can made to correct for this problem. A simple method according to Applequist (2012) is used to adjust the frequencies.

`grid.line` Grid line interval to use. If `NULL`, as in default, this is assigned based on the available data range. However, it can also be forced to a specific value, e.g. `grid.line = 10`. `grid.line` can also be a list to control the interval, line type and colour. For example `grid.line = list(value = 10, lty = 5, col = "purple")`.

`width` For paddle = TRUE, the adjustment factor for width of wind speed intervals. For example, width = 1.5 will make the paddle width 1.5 times wider.

`max.freq` Controls the scaling used by setting the maximum value for the radial limits. This is useful to ensure several plots use the same radial limits.

`dig.lab` The number of significant figures at which scientific number formatting is used in break point and key labelling. Default 5.

`include.lowest` Logical. If `FALSE` (the default), the first interval will be left exclusive and right inclusive. If `TRUE`, the first interval will be left and right inclusive. Passed to the `include.lowest` argument of `cut()`.

`statistic` The `statistic` to be applied to each data bin in the plot. Options currently include "prop.count", "prop.mean" and "abs.count". The default "prop.count" sizes bins according to the proportion of the frequency of measurements. Similarly, "prop.mean" sizes bins according to their relative contribution to the mean. "abs.count" provides the absolute count of measurements in each bin.

`annotate` If `TRUE` then the percentage calm and mean values are printed in each panel together with a description of the statistic below the plot. If `FALSE` then only the statistic will be printed.

`border` Border colour for shaded areas. Default is no border.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.

- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

cols Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

angle.scale In radial plots (e.g., polarPlot()), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. angle.scale can take any value between 0 and 360 to place the scale at a different angle, or FALSE to move it to the side of the plots.

offset offset controls the size of the 'hole' in the middle and is expressed on a scale of 0 to 100, where 0 is no hole and 100 is a hole that takes up the entire plotting area.

strip.position Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

auto.text Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().

Details

pollutionRose() is a windRose() wrapper which brings pollutant forward in the argument list, and attempts to sensibly rescale break points based on the pollutant data range by by-passing ws.int.

By default, pollutionRose() will plot a pollution rose of nox using "wedge" style segments and placing the scale key to the right of the plot.

It is possible to compare two wind speed-direction data sets using pollutionRose(). There are many reasons for doing so e.g. to see how one site compares with another or for meteorological model evaluation. In this case, ws and wd are considered to the the reference data sets with which a second set of wind speed and wind directions are to be compared (ws2 and wd2). The first set of values is subtracted from the second and the differences compared. If for example, wd2 was biased positive compared with wd then pollutionRose will show the bias in polar coordinates. In its default use, wind direction bias is colour-coded to show negative bias in one colour and positive bias in another.

Value

an openair object. Summarised proportions can be extracted directly using the `$data` operator, e.g. `object$data` for `output <- windRose(mydata)`. This returns a data frame with three set columns: `cond`, conditioning based on `type`; `wd`, the wind direction; and `calm`, the `statistic` for the proportion of data unattributed to any specific wind direction because it was collected under calm conditions; and then several (one for each range binned for the plot) columns giving proportions of measurements associated with each `ws` or `pollutant` range plotted as a discrete panel.

See Also

Other polar directional analysis functions: `percentileRose()`, `polarAnnulus()`, `polarCluster()`, `polarDiff()`, `polarFreq()`, `polarPlot()`, `windRose()`

Examples

```
# pollutionRose of nox
pollutionRose(mydata, pollutant = "nox")

# source apportionment plot - contribution to mean
## Not run:
pollutionRose(mydata, pollutant = "pm10", type = "year", statistic = "prop.mean")

# example of comparing 2 met sites
# first we will make some new ws/wd data with a postive bias
mydata$ws2 <- mydata$ws + 2 * rnorm(nrow(mydata)) + 1
mydata$wd2 <- mydata$wd + 30 * rnorm(nrow(mydata)) + 30

# need to correct negative wd
id <- which(mydata$wd2 < 0)
mydata$wd2[id] <- mydata$wd2[id] + 360

# results show postive bias in wd and ws
pollutionRose(mydata, ws = "ws", wd = "wd", ws2 = "ws2", wd2 = "wd2")

## add some wd bias to some nighttime hours
id <- which(as.numeric(format(mydata$date, "%H")) %in% c(23, 1, 2, 3, 4, 5))
mydata$wd2[id] <- mydata$wd[id] + 30 * rnorm(length(id)) + 120
id <- which(mydata$wd2 < 0)
mydata$wd2[id] <- mydata$wd2[id] + 360

pollutionRose(
  mydata,
  ws = "ws",
  wd = "wd",
  ws2 = "ws2",
  wd2 = "wd2",
  breaks = c(-11, -2, -1, -0.5, 0.5, 1, 2, 11),
  cols = c("dodgerblue4", "white", "firebrick"),
  type = "daylight"
)
```

```
## End(Not run)
```

---

quickText *Automatic text formatting for openair*

Description

Workhorse function that automatically applies routine text formatting to common expressions and data names used in openair.

Usage

```
quickText(text, auto.text = TRUE, ...)
```

Arguments

text	A character vector.
auto.text	A logical option. The default, TRUE, applies quickText() to text and returns the result. The alternative, FALSE, returns text unchanged. (A number of openair functions enable/disable quickText() using this option).
...	Not used.

Details

quickText() is routine formatting lookup table. It screens the supplied character vector text and automatically applies formatting to any recognised character sub-series. The function is used in a number of openair functions and can also be used directly by users to format text components of their own graphs (see below).

Value

The function returns an expression for graphical evaluation.

Author(s)

Karl Ropkins

David Carslaw

Jack Davison

Examples

```
# see axis formatting in an openair plot, e.g.:
scatterPlot(
  mydata,
  x = "no2",
  y = "pm10"
)

# using quickText in other plots
plot(
  mydata$no2,
  mydata$pm10,
  xlab = quickText("my no2 label"),
  ylab = quickText("pm10 [ ug.m-3 ]")
)
```

`rollingMean` *Calculate rolling mean pollutant values*

Description

This is a utility function mostly designed to calculate rolling mean statistics relevant to some pollutant limits, e.g., 8 hour rolling means for ozone and 24 hour rolling means for PM10. However, the function has a more general use in helping to display rolling mean values in flexible ways with the rolling window width left, right or centre aligned. The function will try and fill in missing time gaps to get a full time sequence but return a data frame with the same number of rows supplied.

Usage

```
rollingMean(
  mydata,
  pollutant = "o3",
  width = 8L,
  type = "default",
  data.thresh = 75,
  align = c("centre", "center", "left", "right"),
  new.name = NULL,
  date.pad = FALSE,
  ...
)
```

Arguments

`mydata`	A data frame containing a `date` field. `mydata` must contain a `date` field in `Date` or `POSIXct` format. The input time series must be regular, e.g., hourly, daily.
`pollutant`	The name of a pollutant, e.g., `pollutant = "o3"`.
`width`	The averaging period (rolling window width) to use, e.g., `width = 8` will generate 8-hour rolling mean values when hourly data are analysed.

type	Used for splitting the data further. Passed to cutData().
data.thresh	The % data capture threshold. No values are calculated if data capture over the period of interest is less than this value. For example, with width = 8 and data.thresh = 75 at least 6 hours are required to calculate the mean, else NA is returned.
align	Specifies how the moving window should be aligned. "right" means that the previous hours (including the current) are averaged. "left" means that the forward hours are averaged. "centre" (or "center" - the default) centres the current hour in the window.
new.name	The name given to the new column. If not supplied it will create a name based on the name of the pollutant and the averaging period used.
date.pad	Should missing dates be padded? Default is FALSE.
...	Passed to cutData() for use with type.

Value

A tibble with two new columns for the rolling value and the number of valid values used.

Author(s)

David Carslaw

Examples

```
# rolling 8-hour mean for ozone
mydata <- rollingMean(mydata,
  pollutant = "o3", width = 8, new.name =
    "rollingo3", data.thresh = 75, align = "right"
)
```

---

rollingQuantile *Calculate rolling quantile pollutant values*

Description

This is a utility function mostly designed to calculate rolling quantile statistics. The function will try and fill in missing time gaps to get a full time sequence but return a data frame with the same number of rows supplied.

Usage

```
rollingQuantile(
  mydata,
  pollutant = "o3",
  width = 8L,
  type = "default",
```

```
  data.thresh = 75,
  align = c("centre", "center", "left", "right"),
  probs = 0.5,
  date.pad = FALSE,
  ...
)
```

Arguments

mydata	A data frame containing a date field. mydata must contain a date field in Date or POSIXct format. The input time series must be regular, e.g., hourly, daily.
pollutant	The name of a pollutant, e.g., pollutant = "o3".
width	The averaging period (rolling window width) to use, e.g., width = 8 will generate 8-hour rolling mean values when hourly data are analysed.
type	Used for splitting the data further. Passed to cutData().
data.thresh	The % data capture threshold. No values are calculated if data capture over the period of interest is less than this value. For example, with width = 8 and data.thresh = 75 at least 6 hours are required to calculate the mean, else NA is returned.
align	Specifies how the moving window should be aligned. "right" means that the previous hours (including the current) are averaged. "left" means that the forward hours are averaged. "centre" (or "center" - the default) centres the current hour in the window.
probs	Probability for quantile calculate. A number between 0 and 1. Can be more than length one e.g. probs = c(0.05, 0.95).
date.pad	Should missing dates be padded? Default is FALSE.
...	Passed to cutData() for use with type.

Value

A tibble with new columns for the rolling quantile value and the number of valid values used.

Author(s)

David Carslaw

Examples

```
# rolling 24-hour 0.05 and 0.95 quantile for ozone
mydata <- rollingQuantile(mydata,
  pollutant = "o3", width = 24, data.thresh = 75, align = "right", probs = c(0.05, 0.95)
)
```

`runRegression` *Rolling regression for pollutant source characterisation.*

Description

This function calculates rolling regressions for input data with a set window width. The principal use of the function is to identify "dilution lines" where the ratio between two pollutant concentrations is invariant. The original idea is based on the work of Bentley (2004).

Usage

```
runRegression(mydata, x = "nox", y = "pm10", run.len = 3, date.pad = TRUE)
```

Arguments

`mydata`	A data frame with columns for `date` and at least two variables for use in a regression.
`x`	The column name of the `x` variable for use in a linear regression `y = m.x + c`.
`y`	The column name of the `y` variable for use in a linear regression `y = m.x + c`.
`run.len`	The window width to be used for a rolling regression. A value of 3 for example for hourly data will consider 3 one-hour time sequences.
`date.pad`	Should gaps in time series be filled before calculations are made?

Details

The intended use is to apply the approach to air pollution data to extract consecutive points in time where the ratio between two pollutant concentrations changes by very little. By filtering the output for high R2 values (typically more than 0.90 to 0.95), conditions where local source dilution is dominant can be isolated for post processing. The function is more fully described and used in the `openair` online manual, together with examples.

Value

A tibble with `date` and calculated regression coefficients and other information to plot dilution lines.

References

For original inspiration:

Bentley, S. T. (2004). Graphical techniques for constraining estimates of aerosol emissions from motor vehicles using air monitoring network data. Atmospheric Environment,(10), 1491–1500. https://doi.org/10.1016/j.atmosenv.2003.11.033

Example for vehicle emissions high time resolution data:

Farren, N. J., Schmidt, C., Juchem, H., Pöhler, D., Wilde, S. E., Wagner, R. L., Wilson, S., Shaw, M. D., & Carslaw, D. C. (2023). Emission ratio determination from road vehicles using a range of remote emission sensing techniques. Science of The Total Environment, 875. https://doi.org/10.1016/j.scitotenv.2023.162621.

Examples

```
# Just use part of a year of data
output <- runRegression(selectByDate(mydata, year = 2004, month = 1:3),
  x = "nox", y = "pm10", run.len = 3
)

output
```

---

scatterPlot *Flexible scatter plots*

Description

Scatter plots with conditioning and three main approaches: conventional scatterPlot, hexagonal binning and kernel density estimates. The former also has options for fitting smooth fits and linear models with uncertainties shown.

Usage

```
scatterPlot(
  mydata,
  x = "nox",
  y = "no2",
  z = NA,
  method = "scatter",
  group = NA,
  avg.time = "default",
  data.thresh = 0,
  statistic = "mean",
  percentile = NA,
  type = "default",
  smooth = FALSE,
  spline = FALSE,
  linear = FALSE,
  ci = TRUE,
  mod.line = FALSE,
  cols = "hue",
  plot.type = "p",
  key.title = group,
  key.columns = 1,
  key.position = "right",
  strip.position = "top",
  log.x = FALSE,
  log.y = FALSE,
  x.inc = NULL,
  y.inc = NULL,
  limits = NULL,
```

```
  windflow = NULL,
  y.relation = "same",
  x.relation = "same",
  ref.x = NULL,
  ref.y = NULL,
  k = NA,
  dist = 0.02,
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame containing at least two numeric variables to plot.
x	Name of the x-variable to plot. Note that x can be a date field or a factor. For example, x can be one of the openair built in types such as "year" or "season".
y	Name of the numeric y-variable to plot.
z	Name of the numeric z-variable to plot for method = "scatter" or method = "level". Note that for method = "scatter" points will be coloured according to a continuous colour scale, whereas for method = "level" the surface is coloured.
method	Methods include "scatter" (conventional scatter plot), "hexbin" (hexagonal binning using the hexbin package). "level" for a binned or smooth surface plot and "density" (2D kernel density estimates).
group	The grouping variable to use, if any. Setting this to a variable in the data frame has the effect of plotting several series in the same panel using different symbols/colours etc. If set to a variable that is a character or factor, those categories or factor levels will be used directly. If set to a numeric variable, it will split that variable in to quantiles.
avg.time	This defines the time period to average to. Can be "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year". For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be avg.time = "2 month". In addition, avg.time can equal "season", in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that avg.time can be *less* than the time interval of the original series, in which case the series is expanded to the new time interval. This is useful, for example, for calculating a 15-minute time series from an hourly one where an hourly value is repeated for each new 15-minute period. Note that when expanding data in this way it is necessary to ensure that the time interval of the original series is an exact multiple of avg.time e.g. hour to 10 minutes, day to hour. Also, the input time series must have consistent time gaps between successive intervals so that timeAverage() can work out how much 'padding' to apply. To pad-out data in this way choose fill = TRUE.

`data.thresh` The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as `NA`. See also `interval`, `start.date` and `end.date` to see whether it is advisable to set these other options.

`statistic` The statistic to apply when aggregating the data; default is the mean. Can be one of `"mean"`, `"max"`, `"min"`, `"median"`, `"frequency"`, `"sum"`, `"sd"`, `"percentile"`. Note that `"sd"` is the standard deviation, `"frequency"` is the number (frequency) of valid records in the period and `"data.cap"` is the percentage data capture. `"percentile"` is the percentile level (%) between 0-100, which can be set using the `"percentile"` option — see below. Not used if `avg.time = "default"`.

`percentile` The percentile level used when `statistic = "percentile"`. The default is 95%.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`smooth` A smooth line is fitted to the data if `TRUE`; optionally with 95 percent confidence intervals shown. For `method = "level"` a smooth surface will be fitted to binned data.

`spline` A smooth spline is fitted to the data if `TRUE`. This is particularly useful when there are fewer data points or when a connection line between a sequence of points is required.

`linear` A linear model is fitted to the data if `TRUE`; optionally with 95 percent confidence intervals shown. The equation of the line and R2 value is also shown.

`ci` Should the confidence intervals for the smooth/linear fit be shown?

`mod.line` If `TRUE` three lines are added to the scatter plot to help inform model evaluation. The 1:1 line is solid and the 1:0.5 and 1:2 lines are dashed. Together these lines help show how close a group of points are to a 1:1 relationship and also show the points that are within a factor of two (FAC2).

cols	Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.
plot.type	Type of plot: "p" (points, default), "l" (lines) or "b" (both points and lines).
key.title	Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.
key.columns	Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting key.columns to be less than the number of categories.
key.position	Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.
strip.position	Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.
log.x, log.y	Should the x-axis/y-axis appear on a log scale? The default is FALSE. If TRUE a well-formatted log10 scale is used. This can be useful for checking linearity once logged.
x.inc, y.inc	The x/y-interval to be used for binning data when method = "level".
limits	For method = "level" the function does its best to choose sensible limits automatically. However, there are circumstances when the user will wish to set different ones. The limits are set in the form c(lower, upper), so limits = c(0, 100) would force the plot limits to span 0-100.
windflow	If TRUE, the vector-averaged wind speed and direction will be plotted using arrows. Alternatively, can be a list of arguments to control the appearance of the arrows (colour, linewidth, alpha value, etc.). See windflowOpts() for details.
x.relation, y.relation	This determines how the x- and y-axis scales are plotted. "same" ensures all panels use the same scale and "free" will use panel-specific scales. The latter is a useful setting when plotting data with very different values.
ref.x, ref.y	A list with details of the horizontal or vertical lines to be added representing reference line(s). For example, ref.y = list(h = 50, lty = 5) will add a dashed horizontal line at 50. Several lines can be plotted e.g. ref.y = list(h = c(50, 100), lty = c(1, 5), col = c("green", "blue")).
k	Smoothing parameter supplied to gam for fitting a smooth surface when method = "level".
dist	When plotting smooth surfaces (method = "level" and smooth = TRUE), dist controls how far from the original data the predictions should be made. See exclude.too.far from the mgcv package. Data are first transformed to a unit square. Values should be between 0 and 1.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `cex`, `lwd`, `lty`, `alpha`, `pch` and `border` control various graphical parameters.
- For `method = "hexbin"` a log-scale fill is applied by default; pass `trans = NULL` to disable or provide custom `trans` and `inv` transform functions. `bins` controls the number of bins.
- `date.format` controls the format of date-time x-axes.

Details

`scatterPlot()` is the basic function for plotting scatter plots in flexible ways in `openair`. It is flexible enough to consider lots of conditioning variables and takes care of fitting smooth or linear relationships to the data.

There are four main ways of plotting the relationship between two variables, which are set using the `method` option. The default `"scatter"` will plot a conventional scatterPlot. In cases where there are lots of data and over-plotting becomes a problem, then `method = "hexbin"` or `method = "density"` can be useful. The former requires the `hexbin` package to be installed.

There is also a `method = "level"` which will bin the `x` and `y` data according to the intervals set for `x.inc` and `y.inc` and colour the bins according to levels of a third variable, `z`. Sometimes however, a far better understanding of the relationship between three variables (`x`, `y` and `z`) is gained by fitting a smooth surface through the data. See examples below.

A smooth fit is shown if `smooth = TRUE` which can help show the overall form of the data e.g. whether the relationship appears to be linear or not. Also, a linear fit can be shown using `linear = TRUE` as an option.

The user has fine control over the choice of colours and symbol type used.

Another way of reducing the number of points used in the plots which can sometimes be useful is to aggregate the data. For example, hourly data can be aggregated to daily data. See `timePlot()` for examples here.

Value

an openair object

Author(s)

David Carslaw

See Also

timePlot() and timeAverage() for details on selecting averaging times and other statistics in a flexible way

Examples

```
# load openair data if not loaded already
dat2004 <- selectByDate(mydata, year = 2004)

# basic use, single pollutant

scatterPlot(dat2004, x = "nox", y = "no2")
## Not run:
# scatterPlot by year
scatterPlot(mydata, x = "nox", y = "no2", type = "year")

## End(Not run)

# scatterPlot by day of the week, removing key at bottom
scatterPlot(dat2004,
  x = "nox", y = "no2", type = "weekday", key =
    FALSE
)

# example of the use of continuous where colour is used to show
# different levels of a third (numeric) variable
# plot daily averages and choose a filled plot symbol (pch = 16)
# select only 2004
## Not run:

scatterPlot(dat2004, x = "nox", y = "no2", z = "co", avg.time = "day", pch = 16)

# show linear fit, by year
scatterPlot(mydata,
  x = "nox", y = "no2", type = "year", smooth =
    FALSE, linear = TRUE
)

# do the same, but for daily means...
scatterPlot(mydata,
  x = "nox", y = "no2", type = "year", smooth =
    FALSE, linear = TRUE, avg.time = "day"
)

# log scales
scatterPlot(mydata,
  x = "nox", y = "no2", type = "year", smooth =
    FALSE, linear = TRUE, avg.time = "day", log.x = TRUE, log.y = TRUE
```

```
)

# also works with the x-axis in date format (alternative to timePlot)
scatterPlot(mydata,
  x = "date", y = "no2", avg.time = "month",
  key = FALSE
)

## multiple types and grouping variable and continuous colour scale
scatterPlot(mydata, x = "nox", y = "no2", z = "o3", type = c("season", "weekend"))

# use hexagonal binning
scatterPlot(mydata, x = "nox", y = "no2", method = "hexbin")

# scatterPlot by year
scatterPlot(mydata,
  x = "nox", y = "no2", type = "year", method =
    "hexbin"
)

## bin data and plot it - can see how for high NO2, O3 is also high
scatterPlot(mydata, x = "nox", y = "no2", z = "o3", method = "level", dist = 0.02)

## fit surface for clearer view of relationship
scatterPlot(mydata,
  x = "nox", y = "no2", z = "o3", method = "level",
  x.inc = 10, y.inc = 2, smooth = TRUE
)

## End(Not run)
```

`selectByDate` *Subset a data frame based on date*

Description

Utility function to filter a data frame by a date range or specific date periods (month, year, etc.). All options are applied in turn, meaning this function can be used to select quite complex dates simply.

Usage

```
selectByDate(
  mydata,
  start = NULL,
  end = NULL,
  year = NULL,
  month = NULL,
  day = NULL,
  hour = NULL
)
```

Arguments

mydata	A data frame containing a date field in Date or POSIXct format.
start	A start date or date-time string in the form d/m/yyyy, m/d/yyyy, d/m/yyyy HH:MM, m/d/yyyy HH:MM, d/m/yyyy HH:MM:SS, m/d/yyyy HH:MM:SS, yyyy-mm-dd, yyyy-mm-dd HH:MM or yyyy-mm-dd HH:MM:SS.
end	See start for format.
year	A year or years to select e.g. year = 1998:2004 to select 1998-2004 inclusive or year = c(1998, 2004) to select 1998 and 2004.
month	A month or months to select. Can either be numeric e.g. month = 1:6 to select months 1-6 (January to June), or by name e.g. month = c("January", "December"). Names can be abbreviated to 3 letters and be in lower or upper case.
day	A day name or or days to select. day can be numeric (1 to 31) or character. For example day = c("Monday", "Wednesday") or day = 1:10 (to select the 1st to 10th of each month). Names can be abbreviated to 3 letters and be in lower or upper case. Also accepts "weekday" (Monday - Friday) and "weekend" for convenience.
hour	An hour or hours to select from 0-23 e.g. hour = 0:12 to select hours 0 to 12 inclusive.

Author(s)

David Carslaw

Examples

```
## select all of 1999
data.1999 <- selectByDate(mydata, start = "1/1/1999", end = "31/12/1999 23:00")
head(data.1999)
tail(data.1999)

# or...
data.1999 <- selectByDate(mydata, start = "1999-01-01", end = "1999-12-31 23:00")

# easier way
data.1999 <- selectByDate(mydata, year = 1999)


# more complex use: select weekdays between the hours of 7 am to 7 pm
sub.data <- selectByDate(mydata, day = "weekday", hour = 7:19)

# select weekends between the hours of 7 am to 7 pm in winter (Dec, Jan, Feb)
sub.data <- selectByDate(mydata,
  day = "weekend", hour = 7:19, month =
    c("dec", "jan", "feb")
)
```

`selectRunning` *Function to extract run lengths greater than a threshold*

Description

This is a utility function to extract runs of values above a certain threshold. For example, for a data frame of hourly NOx values we would like to extract all those hours where the concentration is at least 500 for contiguous periods of 5 or more hours.

Usage

```
selectRunning(
  mydata,
  pollutant = "nox",
  criterion = ">",
  run.len = 5L,
  threshold = 500,
  type = "default",
  name = "criterion",
  result = c("yes", "no"),
  mode = c("flag", "filter"),
  ...
)
```

Arguments

`mydata`	A data frame with a `date` field and at least one numeric `pollutant` field to analyse.
`pollutant`	Name of variable to process.
`criterion`	Condition to select run lengths e.g. `">"` with select data more than `threshold`.
`run.len`	Run length for extracting contiguous values of `pollutant` meeting the `criterion` in relation to the `threshold`.
`threshold`	The threshold value for `pollutant` above which data should be extracted.
`type`	Used for splitting the data further. Passed to `cutData()`.
`name`	The name of the column to be appended to the data frame when `mode = "flag"`.
`result`	A vector of length 2, defining how to label the run lengths when `mode = "flag"`. The first object should be the label for the `TRUE` label, and the second the `FALSE` label - e.g., `c("yes", "no")`.
`mode`	Changes how the function behaves. When `mode = "flag"`, the default, the function appends a column flagging where the criteria was met. Alternatively, `"filter"` will filter `mydata` to only return rows where the criteria was met.
`...`	Passed to `cutData()` for use with `type`.

Details

This function is useful, for example, for selecting pollution episodes from a data frame where concentrations remain elevated for a certain period of time. It may also be of more general use when analysing air pollution and atmospheric composition data. For example, `selectRunning()` could be used to extract continuous periods of rainfall — which could be important for particle concentrations.

Value

A data frame

Author(s)

David Carslaw

Examples

```
# extract those hours where there are at least 5 consecutive NOx
# concentrations above 500 units
mydata <- selectRunning(mydata, run.len = 5, threshold = 500)

# make a polar plot of those conditions, which shows that those
# conditions are dominated by low wind speeds, not
# in-canyon recirculation
## Not run:
polarPlot(mydata, pollutant = "nox", type = "criterion")

## End(Not run)
```

---

smoothTrend *Calculate nonparametric smooth trends*

Description

Use non-parametric methods to calculate time series trends

Usage

```
smoothTrend(
  mydata,
  pollutant = "nox",
  avg.time = "month",
  data.thresh = 0,
  statistic = "mean",
  percentile = NA,
  k = NULL,
  deseason = FALSE,
  simulate = FALSE,
```

```
  n = 200,
  autocor = FALSE,
  type = "default",
  cols = "brewer1",
  x.relation = "same",
  y.relation = "same",
  ref.x = NULL,
  ref.y = NULL,
  key.columns = 1,
  key.position = "bottom",
  strip.position = "top",
  name.pol = NULL,
  date.breaks = 7,
  date.format = NULL,
  auto.text = TRUE,
  ci = TRUE,
  alpha = 0.2,
  progress = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame of time series. Must include a date field and at least one variable to plot.
pollutant	Name of variable to plot. Two or more pollutants can be plotted, in which case a form like pollutant = c("nox", "co") should be used.
avg.time	This defines the time period to average to. Can be "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year". For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be avg.time = "2 month". In addition, avg.time can equal "season", in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that avg.time can be *less* than the time interval of the original series, in which case the series is expanded to the new time interval. This is useful, for example, for calculating a 15-minute time series from an hourly one where an hourly value is repeated for each new 15-minute period. Note that when expanding data in this way it is necessary to ensure that the time interval of the original series is an exact multiple of avg.time e.g. hour to 10 minutes, day to hour. Also, the input time series must have consistent time gaps between successive intervals so that timeAverage() can work out how much 'padding' to apply. To pad-out data in this way choose fill = TRUE.
data.thresh	The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as NA. See also

interval, start.date and end.date to see whether it is advisable to set these other options.

statistic — Statistic used for calculating monthly values. Default is "mean", but can also be "percentile". See timeAverage() for more details.

percentile — Percentile value(s) to use if statistic = "percentile" is chosen. Can be a vector of numbers e.g. percentile = c(5, 50, 95) will plot the 5th, 50th and 95th percentile values together on the same plot.

k — This is the smoothing parameter used by the mgcv::gam() function in package mgcv. By default it is not used and the amount of smoothing is optimised automatically. However, sometimes it is useful to set the smoothing amount manually using k.

deseason — Should the data be de-deasonalized first? If TRUE the function stl is used (seasonal trend decomposition using loess). Note that if TRUE missing data are first imputed using a Kalman filter and Kalman smooth.

simulate — Should simulations be carried out to determine the Mann-Kendall tau and p-value. The default is FALSE. If TRUE, bootstrap simulations are undertaken, which also account for autocorrelation.

n — Number of bootstrap simulations if simulate = TRUE.

autocor — Should autocorrelation be considered in the trend uncertainty estimates? The default is FALSE. Generally, accounting for autocorrelation increases the uncertainty of the trend estimate sometimes by a large amount.

type — Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

cols — Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

`x.relation`, `y.relation`
: This determines how the x- and y-axis scales are plotted. `"same"` ensures all panels use the same scale and `"free"` will use panel-specific scales. The latter is a useful setting when plotting data with very different values.

`ref.x`
: See `ref.y` for details. In this case the correct date format should be used for a vertical line e.g. `ref.x = list(v = as.POSIXct("2000-06-15"), lty = 5)`.

`ref.y`
: A list with details of the horizontal lines to be added representing reference line(s). For example, `ref.y = list(h = 50, lty = 5)` will add a dashed horizontal line at 50. Several lines can be plotted e.g. `ref.y = list(h = c(50, 100), lty = c(1, 5), col = c("green", "blue"))`.

`key.columns`
: Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`key.position`
: Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position`
: Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`name.pol`
: This option can be used to give alternative names for the variables plotted. Instead of taking the column headings as names, the user can supply replacements. For example, if a column had the name "nox" and the user wanted a different description, then setting `name.pol = "nox before change"` can be used. If more than one pollutant is plotted then use `c` e.g. `name.pol = c("nox here", "o3 there")`.

`date.breaks`
: Number of major x-axis intervals to use. The function will try and choose a sensible number of dates/times as well as formatting the date/time appropriately to the range being considered. The user can override this behaviour by adjusting the value of `date.breaks` up or down.

`date.format`
: This option controls the date format on the x-axis. A sensible format is chosen by default, but the user can set `date.format` to override this. For format types see `strptime()`. For example, to format the date like "Jan-2012" set `date.format = "\%b-\%Y"`.

`auto.text`
: Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`ci`
: Should confidence intervals be plotted? The default is `TRUE`.

`alpha`
: The alpha transparency of shaded confidence intervals - if plotted. A value of 0 is fully transparent and 1 is fully opaque.

`progress`
: Show a progress bar when many groups make up `type`? Defaults to `TRUE`.

`plot`
: When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

key — Deprecated; please use key.position. If FALSE, sets key.position to "none".

... — Addition options are passed on to cutData() for type handling. Some additional arguments are also available:

- xlab, ylab and main override the x-axis label, y-axis label, and plot title.
- ylim and xlim control axis limits.
- layout sets the layout of facets - e.g., layout(2, 5) will have 2 columns and 5 rows.
- fontsize overrides the overall font size of the plot.
- cex, lwd, lty, alpha, and pch control various graphical parameters.

Details

The smoothTrend() function provides a flexible way of estimating the trend in the concentration of a pollutant or other variable. Monthly mean values are calculated from an hourly (or higher resolution) or daily time series. There is the option to deseasonalise the data if there is evidence of a seasonal cycle.

smoothTrend() uses a Generalized Additive Model (GAM) from the mgcv::gam() package to find the most appropriate level of smoothing. The function is particularly suited to situations where trends are not monotonic (see discussion with TheilSen() for more details on this). The smoothTrend() function is particularly useful as an exploratory technique e.g. to check how linear or non-linear trends are.

95% confidence intervals are shown by shading. Bootstrap estimates of the confidence intervals are also available through the simulate option. Residual resampling is used.

Trends can be considered in a very wide range of ways, controlled by setting type - see examples below.

Value

an openair object

Author(s)

David Carslaw

See Also

Other time series and trend functions: TheilSen(), calendarPlot(), timePlot(), timeProp(), timeVariation()

Examples

```
# trend plot for nox
smoothTrend(mydata, pollutant = "nox")

# trend plot by each of 8 wind sectors
## Not run:
smoothTrend(mydata, pollutant = "o3", type = "wd", ylab = "o3 (ppb)")
```

```
# several pollutants, no plotting symbol
smoothTrend(mydata, pollutant = c("no2", "o3", "pm10", "pm25"), pch = NA)

# percentiles
smoothTrend(mydata,
  pollutant = "o3", statistic = "percentile",
  percentile = 95
)

# several percentiles with control over lines used
smoothTrend(mydata,
  pollutant = "o3", statistic = "percentile",
  percentile = c(5, 50, 95), lwd = c(1, 2, 1), lty = c(5, 1, 5)
)

## End(Not run)
```

`splitByDate` *Divide up a data frame by time*

Description

This function partitions a data frame up into different time segments. It produces a new column called controlled by `name` that can be used in many `openair` functions. Note that there must be one more `labels` than there are `dates`.

Usage

```
splitByDate(
  mydata,
  dates = "1/1/2003",
  labels = c("before", "after"),
  name = "split.by",
  format = c("%d/%m/%Y", "%Y/%m/%d", "%d/%m/%Y %H:%M:%S",
    "%Y/%m/%d %H:%M:%S")
)
```

Arguments

`mydata`	A data frame containing a `date` field in an hourly or high resolution format.
`dates`	A date or dates to split data by. Can be passed as R date(time) objects or as characters. If passed as a character, `splitByDate()` expects either "DD/MM/YYYY" or "YYYY/MM/DD" by default, but this can be changed using the `format` argument.
`labels`	Labels for each time partition. Should always be one more `label` than there are `dates`; for example, if `dates = "2020/01/01`, `splitByDate()` requires one label for *before* that date and one label for *after*.

name — The name to give the new column to identify the periods split. Defaults to "split.by".

format — When dates are provided as character strings, this option defines the formats splitByDate() will use to coerce dates into R Date or POSIXCt objects. Passed to lubridate::as_date() or lubridate::as_datetime(). See strptime() for more information.

Author(s)

David Carslaw

Examples

```
# split data up into "before" and "after"
mydata <- splitByDate(mydata,
  dates = "1/04/2000",
  labels = c("before", "after")
)

# split data into 3 partitions
mydata <- splitByDate(mydata,
  dates = c("1/1/2000", "1/3/2003"),
  labels = c("before", "during", "after")
)

# if you have modelled data - could split into modelled and measured by the
# break date
dummy <- data.frame(
  date = Sys.Date() + (-5:5),
  nox = 100 + seq(-50, 50, 10)
)
splitByDate(dummy,
  dates = Sys.Date(),
  labels = c("measured", "modelled"),
  name = "data_type"
)
```

TaylorDiagram *Taylor Diagram for model evaluation with conditioning*

Description

Function to draw Taylor Diagrams for model evaluation. The function allows conditioning by any categorical or numeric variables, which makes the function very flexible.

Usage

```
TaylorDiagram(
  mydata,
  obs = "obs",
  mod = "mod",
  group = NULL,
  type = "default",
  normalise = FALSE,
  pos.cor = NULL,
  cols = "brewer1",
  rms.col = "darkgoldenrod",
  cor.col = "black",
  arrow.lwd = 3,
  annotate = "centred\nRMS error",
  text.obs = "observed",
  key.title = group,
  key.columns = 1,
  key.position = "right",
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame minimally containing a column of observations and a column of predictions.
obs	A column of observations with which the predictions (mod) will be compared.
mod	A column of model predictions. Note, mod can be of length 2 i.e. two lots of model predictions. If two sets of predictions are are present e.g. mod = c("base", "revised"), then arrows are shown on the Taylor Diagram which show the change in model performance in going from the first to the second. This is useful where, for example, there is interest in comparing how one model run compares with another using different assumptions e.g. input data or model set up. See examples below.
group	The group column is used to differentiate between different models and can be a factor or character. The total number of models compared will be equal to the number of unique values of group. group can also be of length two e.g. group = c("model", "site"). In this case all model-site combinations will be shown but they will only be differentiated by colour/symbol by the first grouping variable ("model" in this case). In essence the plot removes the differentiation by the second grouping variable. Because there will be different values of obs for each group, normalise = TRUE should be used.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`normalise` Should the data be normalised by dividing the standard deviation of the observations? The statistics can be normalised (and non-dimensionalised) by dividing both the RMS difference and the standard deviation of the `mod` values by the standard deviation of the observations (`obs`). In this case the "observed" point is plotted on the x-axis at unit distance from the origin. This makes it possible to plot statistics for different species (maybe with different units) on the same plot. The normalisation is done by each `group`/`type` combination.

`pos.cor` Show only positive correlations (`TRUE`) or include negative correlations (`FALSE`). If negative correlations are shown, the Taylor Diagram will show two quadrants. The default, `NULL`, will use two quadrants if any negative correlations are present in the data and one quadrant if all correlations are positive.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`rms.col` Colour for centred-RMS lines and text.

`cor.col` Colour for correlation coefficient lines and text.

`arrow.lwd` Width of arrow used when used for comparing two model outputs.

`annotate` Annotation shown for RMS error.

`text.obs` The plot annotation for observed values; default is "observed".

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.columns` Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `cex`, `lwd`, and `pch` control various graphical parameters.

Details

The Taylor Diagram is a very useful model evaluation tool. The diagram provides a way of showing how three complementary model performance statistics vary simultaneously. These statistics are the correlation coefficient R, the standard deviation (sigma) and the (centred) root-mean-square error. These three statistics can be plotted on one (2D) graph because of the way they are related to one another which can be represented through the Law of Cosines.

The `openair` version of the Taylor Diagram has several enhancements that increase its flexibility. In particular, the straightforward way of producing conditioning plots should prove valuable under many circumstances (using the `type` option). Many examples of Taylor Diagrams focus on model-observation comparisons for several models using all the available data. However, more insight can be gained into model performance by partitioning the data in various ways e.g. by season, daylight/nighttime, day of the week, by levels of a numeric variable e.g. wind speed or by land-use type etc.

To consider several pollutants on one plot, a column identifying the pollutant name can be used e.g. `pollutant`. Then the Taylor Diagram can be plotted as (assuming a data frame `thedata`):

```
TaylorDiagram(thedata, obs = "obs", mod = "mod", group = "model", type = "pollutant")
```

which will give the model performance by pollutant in each panel.

Note that it is important that each panel represents data with the same mean observed data across different groups. Therefore `TaylorDiagram(mydata, group = "model", type = "season")` is OK, whereas `TaylorDiagram(mydata, group = "season", type = "model")` is not because each panel

(representing a model) will have four different mean values — one for each season. Generally, the option group is either missing (one model being evaluated) or represents a column giving the model name. However, the data can be normalised using the normalise option. Normalisation is carried out on a per group/type basis making it possible to compare data on different scales e.g. TaylorDiagram(mydata, group = "season", type = "model", normalise = TRUE). In this way it is possible to compare different pollutants, sites etc. in the same panel.

Also note that if multiple sites are present it makes sense to use type = "site" to ensure that each panel represents an individual site with its own specific standard deviation etc. If this is not the case then select a single site from the data first e.g. subset(mydata, site == "Harwell").

Value

an openair object. If retained, e.g., using output <- TaylorDiagram(thedata, obs = "nox", mod = "mod"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis. For example, output$data will be a data frame consisting of the group, type, correlation coefficient (R), the standard deviation of the observations and measurements.

Author(s)

David Carslaw

Jack Davison

References

Taylor, K.E.: Summarizing multiple aspects of model performance in a single diagram. J. Geophys. Res., 106, 7183-7192, 2001 (also see PCMDI Report 55).

See Also

Other model evaluation functions: conditionalEval(), conditionalQuantile(), modStats()

Examples

```
# in the examples below, most effort goes into making some artificial data
# the function itself can be run very simply

## Not run:
library(dplyr)

dummy model data for 2003
dat <- selectByDate(mydata, year = 2003) |>
  transmute(date, obs = nox, mod = nox, month = as.integer(format(date, "%m")))

# now make mod worse by adding bias and noise according to the month
# do this for 3 different models
mod1 <- dat |>
  mutate(
    mod = mod + 10 * month + 10 * month * rnorm(n()),
    model = "model 1"
  ) |>
```

```
  # lag the results to make the correlation coefficient worse without affecting the sd
  mutate(mod = c(mod[5:n()], mod[(n() - 3):n()]))

mod2 <- dat |>
  mutate(
    mod = mod + 7 * month + 7 * month * rnorm(n()),
    model = "model 2"
  )

mod3 <- dat |>
  mutate(
    mod = mod + 3 * month + 3 * month * rnorm(n()),
    model = "model 3"
  )

mod.dat <- bind_rows(mod1, mod2, mod3)

# basic Taylor plot
TaylorDiagram(mod.dat, obs = "obs", mod = "mod", group = "model")

# Taylor plot by season
TaylorDiagram(
  mod.dat,
  obs = "obs",
  mod = "mod",
  group = "model",
  type = "season"
)

# now show how to evaluate model improvement (or otherwise)
mod1a <- dat |>
  mutate(
    mod = mod + 2 * month + 2 * month * rnorm(n()),
    model = "model 1"
  )

mod2a <- mod2 |> mutate(mod = mod * 1.3)

mod3a <- dat |>
  mutate(
    mod = mod + 10 * month + 10 * month * rnorm(n()),
    model = "model 3"
  )

# now we have a data frame with 3 models, 1 set of observations
# and two sets of model predictions (mod and mod2)
mod.dat <- mod.dat |>
  mutate(mod2 = bind_rows(mod1a, mod2a, mod3a) |> pull(mod))

# do for all models
TaylorDiagram(mod.dat, obs = "obs", mod = c("mod", "mod2"), group = "model")

# all models, by season
```

```
TaylorDiagram(
  mod.dat,
  obs = "obs",
  mod = c("mod", "mod2"),
  group = "model",
  type = "season"
)

# consider two groups (model/month). In this case all months are shown by
# model but are only differentiated by model.
TaylorDiagram(mod.dat, obs = "obs", mod = "mod", group = c("model", "month"))

## End(Not run)
```

TheilSen *Tests for trends using Theil-Sen estimates*

Description

Theil-Sen slope estimates and tests for trend. The TheilSen function is flexible in the sense that it can be applied to data in many ways e.g. by day of the week, hour of day and wind direction. This flexibility makes it much easier to draw inferences from data e.g. why is there a strong downward trend in concentration from one wind sector and not another, or why trends on one day of the week or a certain time of day are unexpected.

Usage

```
TheilSen(
  mydata,
  pollutant = "nox",
  deseason = FALSE,
  type = "default",
  avg.time = "month",
  statistic = "mean",
  percentile = NA,
  data.thresh = 0,
  alpha = 0.05,
  dec.place = 2,
  lab.frac = 0.99,
  lab.cex = 0.8,
  x.relation = "same",
  y.relation = "same",
  data.col = "cornflowerblue",
  trend = list(lty = c(1, 5), lwd = c(2, 1), col = c("red", "red")),
  text.col = "darkgreen",
  slope.text = NULL,
  cols = NULL,
  auto.text = TRUE,
```

```
  autocor = FALSE,
  slope.percent = FALSE,
  date.breaks = 7,
  date.format = NULL,
  strip.position = "top",
  plot = TRUE,
  silent = FALSE,
  ...
)
```

Arguments

mydata	A data frame containing the field date and at least one other parameter for which a trend test is required; typically (but not necessarily) a pollutant.
pollutant	The parameter for which a trend test is required. Mandatory.
deseason	Should the data be de-deasonalized first? If TRUE the function stl is used (seasonal trend decomposition using loess). Note that if TRUE missing data are first imputed using a Kalman filter and Kalman smooth.
type	Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of: • A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season. • The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4). • The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.
avg.time	Can be "month" (the default), "season" or "year". Determines the time over which data should be averaged. Note that for "year", six or more years are required. For "season" the data are split up into spring: March, April, May etc. Note that December is considered as belonging to winter of the following year.
statistic	Statistic used for calculating monthly values. Default is "mean", but can also be "percentile". See timeAverage() for more details.
percentile	Single percentile value to use if statistic = "percentile" is chosen.
data.thresh	The data capture threshold to use (%) when aggregating the data using avg.time. A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as NA.

`alpha` For the confidence interval calculations of the slope. The default is 0.05. To show 99\ trend, choose alpha = 0.01 etc.

`dec.place` The number of decimal places to display the trend estimate at. The default is 2.

`lab.frac` Fraction along the y-axis that the trend information should be printed at, default 0.99.

`lab.cex` Size of text for trend information.

`x.relation`, `y.relation`
This determines how the x- and y-axis scales are plotted. `"same"` ensures all panels use the same scale and `"free"` will use panel-specific scales. The latter is a useful setting when plotting data with very different values.

`data.col` Colour name for the data

`trend` list containing information on the line width, line type and line colour for the main trend line and confidence intervals respectively.

`text.col` Colour name for the slope/uncertainty numeric estimates

`slope.text` The text shown for the slope (default is 'units/year').

`cols` Predefined colour scheme, currently only enabled for `"greyscale"`.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`autocor` Should autocorrelation be considered in the trend uncertainty estimates? The default is `FALSE`. Generally, accounting for autocorrelation increases the uncertainty of the trend estimate — sometimes by a large amount.

`slope.percent` Should the slope and the slope uncertainties be expressed as a percentage change per year? The default is `FALSE` and the slope is expressed as an average units/year change e.g. ppb. Percentage changes can often be confusing and should be clearly defined. Here the percentage change is expressed as 100 * (C.end/C.start - 1) / (end.year - start.year). Where C.start is the concentration at the start date and C.end is the concentration at the end date.

For `avg.time = "year"` (end.year - start.year) will be the total number of years - 1. For example, given a concentration in year 1 of 100 units and a percentage reduction of 5%/yr, after 5 years there will be 75 units but the actual time span will be 6 years i.e. year 1 is used as a reference year. Things are slightly different for monthly values e.g. `avg.time = "month"`, which will use the total number of months as a basis of the time span and is therefore able to deal with partial years. There can be slight differences in the %/yr trend estimate therefore, depending on whether monthly or annual values are considered.

`date.breaks` Number of major x-axis intervals to use. The function will try and choose a sensible number of dates/times as well as formatting the date/time appropriately to the range being considered. The user can override this behaviour by adjusting the value of `date.breaks` up or down.

`date.format` This option controls the date format on the x-axis. A sensible format is chosen by default, but the user can set `date.format` to override this. For format types see `strptime()`. For example, to format the date like "Jan-2012" set `date.format = "\%b-\%Y"`.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `type`s are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`silent` When `FALSE` the function will give updates on trend-fitting progress.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `cex`, `lwd`, and `pch` control various graphical parameters.
- `ylim` and `xlim` control axis limits.

Details

For data that are strongly seasonal, perhaps from a background site, or a pollutant such as ozone, it will be important to deseasonalise the data (using the option `deseason = TRUE`.Similarly, for data that increase, then decrease, or show sharp changes it may be better to use `smoothTrend()`.

A minimum of 6 points are required for trend estimates to be made.

Note! that since version 0.5-11 openair uses Theil-Sen to derive the p values also for the slope. This is to ensure there is consistency between the calculated p value and other trend parameters i.e. slope estimates and uncertainties. The p value and all uncertainties are calculated through bootstrap simulations.

Note that the symbols shown next to each trend estimate relate to how statistically significant the trend estimate is: p $<$ 0.001 = ***, p $<$ 0.01 = **, p $<$ 0.05 = * and p $<$ 0.1 = $+$.

Some of the code used in `TheilSen` is based on that from Rand Wilcox. This mostly relates to the Theil-Sen slope estimates and uncertainties. Further modifications have been made to take account of correlated data based on Kunsch (1989). The basic function has been adapted to take account of auto-correlated data using block bootstrap simulations if `autocor = TRUE` (Kunsch, 1989). We follow the suggestion of Kunsch (1989) of setting the block length to n(1/3) where n is the length of the time series.

The slope estimate and confidence intervals in the slope are plotted and numerical information presented.

Value

an openair object. The `data` component of the `TheilSen` output includes two subsets: `main.data`, the monthly data `res2` the trend statistics. For `output <- TheilSen(mydata, "nox")`, these can be extracted as `object$data$main.data` and `object$data$res2`, respectively. Note: In the case of the intercept, it is assumed the y-axis crosses the x-axis on 1/1/1970.

Author(s)

David Carslaw with some trend code from Rand Wilcox

References

Helsel, D., Hirsch, R., 2002. Statistical methods in water resources. US Geological Survey. Note that this is a very good resource for statistics as applied to environmental data.

Hirsch, R. M., Slack, J. R., Smith, R. A., 1982. Techniques of trend analysis for monthly water-quality data. Water Resources Research 18 (1), 107-121.

Kunsch, H. R., 1989. The jackknife and the bootstrap for general stationary observations. Annals of Statistics 17 (3), 1217-1241.

Sen, P. K., 1968. Estimates of regression coefficient based on Kendall's tau. Journal of the American Statistical Association 63(324).

Theil, H., 1950. A rank invariant method of linear and polynomial regression analysis, i, ii, iii. Proceedings of the Koninklijke Nederlandse Akademie Wetenschappen, Series A - Mathematical Sciences 53, 386-392, 521-525, 1397-1412.

...see also several of the Air Quality Expert Group (AQEG) reports for the use of similar tests applied to UK/European air quality data.

See Also

Other time series and trend functions: `calendarPlot()`, `smoothTrend()`, `timePlot()`, `timeProp()`, `timeVariation()`

Examples

```
# trend plot for nox
TheilSen(mydata, pollutant = "nox")

# trend plot for ozone with p=0.01 i.e. uncertainty in slope shown at
# 99 % confidence interval

## Not run:
TheilSen(mydata, pollutant = "o3", ylab = "o3 (ppb)", alpha = 0.01)

## End(Not run)

# trend plot by each of 8 wind sectors
## Not run:
TheilSen(mydata, pollutant = "o3", type = "wd", ylab = "o3 (ppb)")

## End(Not run)

# and for a subset of data (from year 2000 onwards)
## Not run:
TheilSen(selectByDate(mydata, year = 2000:2005), pollutant = "o3", ylab = "o3 (ppb)")

## End(Not run)
```

`timeAverage` *Function to calculate time averages for data frames*

Description

Function to flexibly aggregate or expand data frames by different time periods, calculating vector-averaged wind direction where appropriate. The averaged periods can also take account of data capture rates.

Usage

```
timeAverage(
  mydata,
  avg.time = "day",
  data.thresh = 0,
  statistic = "mean",
  type = "default",
  percentile = NA,
  start.date = NA,
  end.date = NA,
  interval = NA,
  vector.ws = FALSE,
  fill = FALSE,
  progress = TRUE,
  ...
)
```

Arguments

`mydata`	A data frame containing a `date` field . Can be class `POSIXct` or `Date`.
`avg.time`	This defines the time period to average to. Can be `"sec"`, `"min"`, `"hour"`, `"day"`, `"DSTday"`, `"week"`, `"month"`, `"quarter"` or `"year"`. For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be `avg.time = "2 month"`. In addition, `avg.time` can equal `"season"`, in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that `avg.time` can be *less* than the time interval of the original series, in which case the series is expanded to the new time interval. This is useful, for example, for calculating a 15-minute time series from an hourly one where an hourly value is repeated for each new 15-minute period. Note that when expanding data in this way it is necessary to ensure that the time interval of the original series is an exact multiple of `avg.time` e.g. hour to 10 minutes, day to hour. Also, the input time series must have consistent time gaps between successive intervals so that `timeAverage()` can work out how much 'padding' to apply. To pad-out data in this way choose `fill = TRUE`.

`data.thresh` The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as `NA`. See also `interval`, `start.date` and `end.date` to see whether it is advisable to set these other options.

`statistic` The statistic to apply when aggregating the data; default is the mean. Can be one of `"mean"`, `"max"`, `"min"`, `"median"`, `"frequency"`, `"sum"`, `"sd"`, `"percentile"`. Note that `"sd"` is the standard deviation, `"frequency"` is the number (frequency) of valid records in the period and `"data.cap"` is the percentage data capture. `"percentile"` is the percentile level (%) between 0-100, which can be set using the `"percentile"` option — see below. Not used if `avg.time = "default"`.

`type` `type` allows `timeAverage()` to be applied to cases where there are groups of data that need to be split and the function applied to each group. The most common example is data with multiple sites identified with a column representing site name e.g. `type = "site"`. More generally, `type` should be used where the date repeats for a particular grouping variable. However, if type is not supplied the data will still be averaged but the grouping variables (character or factor) will be dropped.

`percentile` The percentile level used when `statistic = "percentile"`. The default is 95%.

`start.date` A string giving a start date to use. This is sometimes useful if a time series starts between obvious intervals. For example, for a 1-minute time series that starts `2009-11-29 12:07:00` that needs to be averaged up to 15-minute means, the intervals would be `2009-11-29 12:07:00`, `2009-11-29 12:22:00`, etc. Often, however, it is better to round down to a more obvious start point, e.g., `2009-11-29 12:00:00` such that the sequence is then `2009-11-29 12:00:00`, `2009-11-29 12:15:00`, and so on. `start.date` is therefore used to force this type of sequence. Note that this option does not truncate a time series if it already starts earlier than `start.date`; see `selectByDate()` for that functionality.

`end.date` A string giving an end date to use. This is sometimes useful to make sure a time series extends to a known end point and is useful when `data.thresh > 0` but the input time series does not extend up to the final full interval. For example, if a time series ends sometime in October but annual means are required with a data capture of >75 % then it is necessary to extend the time series up until the end of the year. Input in the format yyyy-mm-dd HH:MM. Note that this option does not truncate a time series if it already ends later than `end.date`; see `selectByDate()` for that functionality.

`interval` The `timeAverage()` function tries to determine the interval of the original time series (e.g. hourly) by calculating the most common interval between time steps. The interval is needed for calculations where the `data.thresh` >0. For the vast majority of regular time series this works fine. However, for data with very poor data capture or irregular time series the automatic detection may not work. Also, for time series such as monthly time series where there is a variable difference in time between months users should specify the time interval explicitly e.g.

	interval = "month". Users can also supply a time interval to *force* on the time series. See avg.time for the format. This option can sometimes be useful with start.date and end.date to ensure full periods are considered e.g. a full year when avg.time = "year".
vector.ws	Should vector averaging be carried out on wind speed if available? The default is FALSE and scalar averages are calculated. Vector averaging of the wind speed is carried out on the u and v wind components. For example, consider the average of two hours where the wind direction and speed of the first hour is 0 degrees and 2m/s and 180 degrees and 2m/s for the second hour. The scalar average of the wind speed is simply the arithmetic average = 2m/s and the vector average is 0m/s. Vector-averaged wind speeds will always be lower than scalar-averaged values.
fill	When time series are expanded, i.e., when a time interval is less than the original time series, data are 'padded out' with NA. To 'pad-out' the additional data with the first row in each original time interval, choose fill = TRUE.
progress	Show a progress bar when many groups make up type? Defaults to TRUE.
...	Additional arguments for other functions calling timeAverage().

Details

This function calculates time averages for a data frame. It also treats wind direction correctly through vector-averaging. For example, the average of 350 degrees and 10 degrees is either 0 or 360 - not 180. The calculations therefore average the wind components.

When a data capture threshold is set through data.thresh it is necessary for timeAverage() to know what the original time interval of the input time series is. The function will try and calculate this interval based on the most common time gap (and will print the assumed time gap to the screen). This works fine most of the time but there are occasions where it may not e.g. when very few data exist in a data frame or the data are monthly (i.e. non-regular time interval between months). In this case the user can explicitly specify the interval through interval in the same format as avg.time e.g. interval = "month". It may also be useful to set start.date and end.date if the time series do not span the entire period of interest. For example, if a time series ended in October and annual means are required, setting end.date to the end of the year will ensure that the whole period is covered and that data.thresh is correctly calculated. The same also goes for a time series that starts later in the year where start.date should be set to the beginning of the year.

timeAverage() should be useful in many circumstances where it is necessary to work with different time average data. For example, hourly air pollution data and 15-minute meteorological data. To merge the two data sets timeAverage() can be used to make the meteorological data 1-hour means first. Alternatively, timeAverage() can be used to expand the hourly data to 15 minute data - see example below.

For the research community timeAverage() should be useful for dealing with outputs from instruments where there are a range of time periods used.

It is also very useful for plotting data using timePlot(). Often the data are too dense to see patterns and setting different averaging periods easily helps with interpretation.

Value

Returns a data frame with date in class POSIXct.

Author(s)

David Carslaw

See Also

timePlot() that plots time series data and uses timeAverage() to aggregate data where necessary.

calcPercentile() that wraps timeAverage() to allow multiple percentiles to be calculated at once.

Examples

```
# daily average values
daily <- timeAverage(mydata, avg.time = "day")

# daily average values ensuring at least 75 % data capture
# i.e., at least 18 valid hours
## Not run:
daily <- timeAverage(mydata, avg.time = "day", data.thresh = 75)

## End(Not run)

# 2-weekly averages
## Not run:
fortnight <- timeAverage(mydata, avg.time = "2 week")

## End(Not run)

# make a 15-minute time series from an hourly one
## Not run:
min15 <- timeAverage(mydata, avg.time = "15 min", fill = TRUE)

## End(Not run)

# average by grouping variable
## Not run:
dat <- importAURN(c("kc1", "my1"), year = 2011:2013)
timeAverage(dat, avg.time = "year", type = "site")

# can also retain site code
timeAverage(dat, avg.time = "year", type = c("site", "code"))

# or just average all the data, dropping site/code
timeAverage(dat, avg.time = "year")

## End(Not run)
```

timePlot *Plot time series, perhaps for multiple pollutants, grouped or in separate panels.*

Description

The `timePlot()` is the basic time series plotting function in `openair`. Its purpose is to make it quick and easy to plot time series for pollutants and other variables. The other purpose is to plot potentially many variables together in as compact a way as possible.

Usage

```
timePlot(
  mydata,
  pollutant = "nox",
  group = FALSE,
  stack = FALSE,
  normalise = NULL,
  avg.time = "default",
  data.thresh = 0,
  statistic = "mean",
  percentile = NA,
  date.pad = FALSE,
  type = "default",
  cols = "brewer1",
  log = FALSE,
  windflow = NULL,
  smooth = FALSE,
  smooth_k = NULL,
  ci = TRUE,
  x.relation = "same",
  y.relation = "same",
  ref.x = NULL,
  ref.y = NULL,
  key.columns = NULL,
  key.position = "bottom",
  strip.position = "top",
  name.pol = pollutant,
  date.breaks = 7,
  date.format = NULL,
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame of time series. Must include a date field and at least one variable to plot.
pollutant	Name of variable to plot. Two or more pollutants can be plotted, in which case a form like pollutant = c("nox", "co") should be used.
group	Controls how multiple lines/series are grouped. Three options are available: • FALSE (default): each pollutant is plotted in its own panel with its own scale. • TRUE: all pollutants are plotted together on the same panel and scale, coloured by pollutant name. • A character string giving the name of a column in mydata (e.g. group = "site" or group = "pollutant"): lines are coloured by the values in that column. With a single pollutant all groups appear in one panel; with multiple pollutants each pollutant gets its own panel and lines within each panel are coloured by the group column. This is particularly useful for long-format data where multiple species are stored in one column.
stack	If TRUE the time series will be stacked by year. This option can be useful if there are several years worth of data making it difficult to see much detail when plotted on a single plot.
normalise	Should variables be normalised? The default is is not to normalise the data. normalise can take two values, either "mean" or a string representing a date in UK format e.g. "1/1/1998" (in the format dd/mm/YYYY). If normalise = "mean" then each time series is divided by its mean value. If a date is chosen, then values at that date are set to 100 and the rest of the data scaled accordingly. Choosing a date (say at the beginning of a time series) is very useful for showing how trends diverge over time. Setting group = TRUE is often useful too to show all time series together in one panel.
avg.time	This defines the time period to average to. Can be "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year". For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be avg.time = "2 month". In addition, avg.time can equal "season", in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that avg.time can be *less* than the time interval of the original series, in which case the series is expanded to the new time interval. This is useful, for example, for calculating a 15-minute time series from an hourly one where an hourly value is repeated for each new 15-minute period. Note that when expanding data in this way it is necessary to ensure that the time interval of the original series is an exact multiple of avg.time e.g. hour to 10 minutes, day to hour. Also, the input time series must have consistent time gaps between successive intervals so that timeAverage() can work out how much 'padding' to apply. To pad-out data in this way choose fill = TRUE.
data.thresh	The data capture threshold to use (%). A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as NA. See also

interval, start.date and end.date to see whether it is advisable to set these other options.

statistic — The statistic to apply when aggregating the data; default is the mean. Can be one of "mean", "max", "min", "median", "frequency", "sum", "sd", "percentile". Note that "sd" is the standard deviation, "frequency" is the number (frequency) of valid records in the period and "data.cap" is the percentage data capture. "percentile" is the percentile level (%) between 0-100, which can be set using the "percentile" option — see below. Not used if avg.time = "default".

percentile — The percentile level in percent used when statistic = "percentile" and when aggregating the data with avg.time. More than one percentile level is allowed for type = "default" e.g. percentile = c(50, 95). Not used if avg.time = "default".

date.pad — Should missing data be padded-out? This is useful where a data frame consists of two or more "chunks" of data with time gaps between them. By setting date.pad = TRUE the time gaps between the chunks are shown properly, rather than with a line connecting each chunk. For irregular data, set to FALSE. Note, this should not be set for type other than default.

type — Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of:

- A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season.
- The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4).
- The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.

cols — Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

log — Should the y-axis appear on a log scale? The default is FALSE. If TRUE a well-formatted log10 scale is used. This can be useful for plotting data for several different pollutants that exist on very different scales. It is therefore useful to use log = TRUE together with group = TRUE.

windflow — If TRUE, the vector-averaged wind speed and direction will be plotted using arrows. Alternatively, can be a list of arguments to control the appearance of the arrows (colour, linewidth, alpha value, etc.). See windflowOpts() for details.

smooth — Should a smooth line be applied to the data? The default is FALSE.

smooth_k — An integer controlling the number of basis functions used in the GAM smooth. In a GAM, k sets the maximum degrees of freedom for the smooth term: larger values allow more flexibility and can capture finer structure in the data, while smaller values produce smoother, less wiggly fits. The default (NULL) lets ggplot2 choose automatically (typically k = 10). Increase k if the smooth appears too rigid; decrease it to avoid over-fitting.

ci — If a smooth fit line is applied, then ci determines whether the 95 percent confidence intervals are shown.

x.relation, y.relation — This determines how the x- and y-axis scales are plotted. "same" ensures all panels use the same scale and "free" will use panel-specific scales. The latter is a useful setting when plotting data with very different values.

ref.x — See ref.y for details. In this case the correct date format should be used for a vertical line e.g. ref.x = list(v = as.POSIXct("2000-06-15"), lty = 5).

ref.y — A list with details of the horizontal lines to be added representing reference line(s). For example, ref.y = list(h = 50, lty = 5) will add a dashed horizontal line at 50. Several lines can be plotted e.g. ref.y = list(h = c(50, 100), lty = c(1, 5), col = c("green", "blue")).

key.columns — Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting key.columns to be less than the number of categories.

key.position — Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

strip.position — Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

name.pol — This option can be used to give alternative names for the variables plotted. Instead of taking the column headings as names, the user can supply replacements. For example, if a column had the name "nox" and the user wanted a different description, then setting name.pol = "nox before change" can be used. If more than one pollutant is plotted then use c e.g. name.pol = c("nox here", "o3 there").

date.breaks — Number of major x-axis intervals to use. The function will try and choose a sensible number of dates/times as well as formatting the date/time appropriately to the range being considered. The user can override this behaviour by adjusting the value of date.breaks up or down.

date.format — This option controls the date format on the x-axis. A sensible format is chosen by default, but the user can set date.format to override this. For format types see strptime(). For example, to format the date like "Jan-2012" set date.format = "\%b-\%Y".

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `lwd`, `lty`, and `pch` control various graphical parameters.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each tile.
- `ylim` and `xlim` control axis limits.

Details

The function is flexible enough to plot more than one variable at once. If more than one variable is chosen plots it can either show all variables on the same plot (with different line types) *on the same scale*, or (if `group = FALSE`) each variable in its own panels with its own scale.

The general preference is not to plot two variables on the same graph with two different y-scales. It can be misleading to do so and difficult with more than two variables. If there is in interest in plotting several variables together that have very different scales, then it can be useful to normalise the data first, which can be down be setting the `normalise` option.

The user has fine control over the choice of colours, line width and line types used. This is useful for example, to emphasise a particular variable with a specific line type/colour/width.

`timePlot()` works very well with `selectByDate()`, which is used for selecting particular date ranges quickly and easily. See examples below.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

See Also

Other time series and trend functions: `TheilSen()`, `calendarPlot()`, `smoothTrend()`, `timeProp()`, `timeVariation()`

Examples

```
# basic use, single pollutant
timePlot(mydata, pollutant = "nox")

# two pollutants in separate panels
## Not run:
timePlot(mydata, pollutant = c("nox", "no2"))

# two pollutants in the same panel with the same scale
timePlot(mydata, pollutant = c("nox", "no2"), group = TRUE)

# group by a column (e.g. long-format data with a 'site' column)
d <- rbind(
  cbind(mydata[, c("date", "nox")], site = "London"),
  cbind(transform(mydata[, c("date", "nox")], nox = nox * 1.5), site = "Manchester")
)
timePlot(d, pollutant = "nox", group = "site")

# alternative by normalising concentrations and plotting on the same scale
timePlot(
  mydata,
  pollutant = c("nox", "co", "pm10", "so2"),
  group = TRUE,
  avg.time = "year",
  normalise = "1/1/1998",
  lwd = 3,
  lty = 1
)

# examples of selecting by date

# plot for nox in 1999
timePlot(selectByDate(mydata, year = 1999), pollutant = "nox")

# select specific date range for two pollutants
timePlot(
  selectByDate(mydata, start = "6/8/2003", end = "13/8/2003"),
  pollutant = c("no2", "o3")
)

# choose different line styles etc
timePlot(mydata, pollutant = c("nox", "no2"), lty = 1)

# choose different line styles etc
timePlot(
  selectByDate(mydata, year = 2004, month = 6),
  pollutant = c("nox", "no2"),
  lwd = c(1, 2),
  col = "black"
)

# different averaging times
```

```
# daily mean O3
timePlot(mydata, pollutant = "o3", avg.time = "day")

# daily mean O3 ensuring each day has data capture of at least 75%
timePlot(mydata, pollutant = "o3", avg.time = "day", data.thresh = 75)

# 2-week average of O3 concentrations
timePlot(mydata, pollutant = "o3", avg.time = "2 week")

## End(Not run)
```

`timeProp` *Time series plot with categories shown as a stacked bar chart*

Description

This function shows time series plots as stacked bar charts. The different categories in the bar chart are made up from a character or factor variable in a data frame. The function is primarily developed to support the plotting of cluster analysis output from `polarCluster()` and `trajCluster()` that consider local and regional (back trajectory) cluster analysis respectively. However, the function has more general use for understanding time series data.

Usage

```
timeProp(
  mydata,
  pollutant = "nox",
  proportion = "wd",
  avg.time = "day",
  type = "default",
  cols = "Set1",
  normalise = FALSE,
  x.relation = "same",
  y.relation = "same",
  key.columns = 1,
  key.position = "right",
  key.title = proportion,
  strip.position = "top",
  date.breaks = 7,
  date.format = NULL,
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame containing the fields date, pollutant and a splitting variable proportion
pollutant	Name of the pollutant to plot contained in mydata.
proportion	The splitting variable that makes up the bars in the bar chart, defaulting to "wd". Could be "cluster" if the output from polarCluster() or trajCluster() is being analysed. If proportion is a numeric variable it is split into 4 quantiles (by default) by cutData(). If proportion is a factor or character variable then the categories are used directly.
avg.time	This defines the time period to average to. Can be "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year". For much increased flexibility a number can precede these options followed by a space. For example, an average of 2 months would be avg.time = "2 month". In addition, avg.time can equal "season", in which case 3-month seasonal values are calculated with spring defined as March, April, May and so on. Note that avg.time when used in timeProp should be greater than the time gap in the original data. For example, avg.time = "day" for hourly data is OK, but avg.time = "hour" for daily data is not.
type	Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of: • A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season. • The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4). • The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.
cols	Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.
normalise	If normalise = TRUE then each time interval is scaled to 100. This is helpful to show the relative (percentage) contribution of the proportions.
x.relation, y.relation	This determines how the x- and y-axis scales are plotted. "same" ensures all panels use the same scale and "free" will use panel-specific scales. The latter is a useful setting when plotting data with very different values.

key.columns — Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting key.columns to be less than the number of categories.

key.position — Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

key.title — Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.

strip.position — Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

date.breaks — Number of major x-axis intervals to use. The function will try and choose a sensible number of dates/times as well as formatting the date/time appropriately to the range being considered. The user can override this behaviour by adjusting the value of date.breaks up or down.

date.format — This option controls the date format on the x-axis. A sensible format is chosen by default, but the user can set date.format to override this. For format types see strptime(). For example, to format the date like "Jan-2012" set date.format = "\%b-\%Y".

auto.text — Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().

plot — When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

key — Deprecated; please use key.position. If FALSE, sets key.position to "none".

... — Addition options are passed on to cutData() for type handling. Some additional arguments are also available:

- xlab, ylab and main override the x-axis label, y-axis label, and plot title.
- layout sets the layout of facets - e.g., layout(2, 5) will have 2 columns and 5 rows.
- fontsize overrides the overall font size of the plot.
- border sets the border colour of each bar.

Details

In order to plot time series in this way, some sort of time aggregation is needed, which is controlled by the option avg.time.

The plot shows the value of pollutant on the y-axis (averaged according to avg.time). The time intervals are made up of bars split according to proportion. The bars therefore show how the total value of pollutant is made up for any time interval.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

See Also

Other time series and trend functions: `TheilSen()`, `calendarPlot()`, `smoothTrend()`, `timePlot()`, `timeVariation()`

Other cluster analysis functions: `polarCluster()`, `trajCluster()`

Examples

```
# monthly plot of SO2 showing the contribution by wind sector
timeProp(mydata, pollutant = "so2", avg.time = "month", proportion = "wd")
```

---

timeVariation *Temporal variation plots with flexible panel control*

Description

Plots temporal variation for different variables, typically pollutant concentrations, across user-defined time scales. Multiple panels can be shown, such as hour of the day, day of the week, week of the year, month of the year, annual mean, or any other time-based grouping the user specifies. By default, this function plots the diurnal, day of the week and monthly variation for different variables, typically pollutant concentrations. Four separate plots are produced. This is a convenient alternative to using `variationPlot()` and assembling the plots manually.

Usage

```
timeVariation(
  mydata,
  pollutant = "nox",
  panels = c("hour.weekday", "hour", "month", "weekday"),
  local.tz = NULL,
  normalise = FALSE,
  xlab = NULL,
  name.pol = NULL,
  type = "default",
  group = NULL,
  difference = FALSE,
  statistic = "mean",
  conf.int = NULL,
  B = 100,
```

```
  ci = TRUE,
  cols = "hue",
  ref.y = NULL,
  key = NULL,
  key.columns = NULL,
  key.position = "top",
  strip.position = "top",
  panel.gap = 1.5,
  auto.text = TRUE,
  alpha = 0.4,
  plot = TRUE,
  ...
)
```

Arguments

mydata	A data frame of time series. Must include a date field and at least one variable to plot.
pollutant	Name of variable to plot. Two or more pollutants can be plotted, in which case a form like pollutant = c("nox", "co") should be used.
panels	A vector of character values which can be passed to cutData(); used to define each panel in the plot. The first panel will take up the entire first row, and any remaining panels will make up the bottom row. If a single panel is given, it will take up the entire plotting area. Combining two type strings delimited with a full stop (e.g., "hour.weekday") will use the first as the x-axis variable the second as a facet.
local.tz	Used for identifying whether a date has daylight savings time (DST) applied or not. Examples include local.tz = "Europe/London", local.tz = "America/New_York", i.e., time zones that assume DST. https://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones shows time zones that should be valid for most systems. It is important that the original data are in GMT (UTC) or a fixed offset from GMT.
normalise	Should variables be normalised? The default is FALSE. If TRUE then the variable(s) are divided by their mean values. This helps to compare the shape of the diurnal trends for variables on very different scales.
xlab	x-axis label; one for each panel. Defaults to the x-axis variable defined in panels. Must be the same length as panels.
name.pol	This option can be used to give alternative names for the variables plotted. Instead of taking the column headings as names, the user can supply replacements. For example, if a column had the name "nox" and the user wanted a different description, then setting name.pol = "nox before change" can be used. If more than one pollutant is plotted then use c e.g. name.pol = c("nox here", "o3 there").
type	type determines how the data are split i.e. conditioned, and then plotted. The default is will produce a single plot using the entire data. Type can be one of the built-in types as detailed in cutData(), e.g., "season", "year", "weekday"

and so on. For example, type = "season" will produce four plots — one for each season.

It is also possible to choose type as another variable in the data frame. If that variable is numeric, then the data will be split into four quantiles (if possible) and labelled accordingly. If type is an existing character or factor variable, then those categories/levels will be used directly. This offers great flexibility for understanding the variation of different variables and how they depend on one another.

Only one type is allowed in timeVariation(), and it is applied to each panel. For additional splits, use the "x.type" syntax in the panels argument (e.g, panels = c("hour.weekday")).

group — This sets the grouping variable to be used. For example, if a data frame had a column site setting group = "site" will plot all sites together in each panel. Passed to cutData().

difference — If two pollutants are chosen then setting difference = TRUE will also plot the difference in means between the two variables as pollutant[2] - pollutant[1]. Bootstrap 95\ difference in means are also calculated. A horizontal dashed line is shown at y = 0. The difference can also be calculated if there is a column that identifies two groups, e.g., having used splitByDate(). In this case it is possible to call the function with the option group = "split.by" and difference = TRUE.

statistic — Can be "mean" (default) or "median". If the statistic is "mean" then the mean line and the 95% confidence interval in the mean are plotted by default. If the statistic is "median" then the median line is plotted together with the 5/95 and 25/75th quantiles are plotted. Users can control the confidence intervals with conf.int.

conf.int — The confidence intervals to be plotted. If statistic = "mean" then the confidence intervals in the mean are plotted. If statistic = "median" then the conf.int and 1 - conf.int *quantiles* are plotted. Any number of conf.ints can be provided.

B — Number of bootstrap replicates to use. Can be useful to reduce this value when there are a large number of observations available to increase the speed of the calculations without affecting the 95% confidence interval calculations by much.

ci — Should confidence intervals be shown? The default is TRUE. Setting this to FALSE can be useful if multiple pollutants are chosen where over-lapping confidence intervals can over complicate plots.

cols — Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.

ref.y — A list with details of the horizontal lines to be added representing reference line(s). For example, ref.y = list(h = 50, lty = 5) will add a dashed horizontal line at 50. Several lines can be plotted e.g. ref.y = list(h = c(50, 100), lty = c(1, 5), col = c("green", "blue")).

`key` By default `timeVariation()` produces four plots on one page. While it is useful to see these plots together, it is sometimes necessary just to use one for a report. If `key` is `TRUE`, a key is added to all plots allowing the extraction of a single plot *with* key. If `key` is `FALSE`, no key is shown for any plot.

`key.columns` Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`panel.gap` The gap between panels in any split panel (e.g., the default `"hour.weekday"` panel).

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`alpha` The alpha transparency used for plotting confidence intervals. `0` is fully transparent and 1 is opaque. The default is `0.4`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `lwd` and `lty` control various graphical parameters.
- `fontsize` overrides the overall font size of the plot.
- `ylim` controls axis limits.

Details

The variation of pollutant concentrations by time can reveal many interesting features that relate to source types and meteorology. For traffic sources, there are often important differences in the way vehicles vary by type - e.g., fewer heavy vehicles at weekends.

Users can supply their own `ylim`, e.g. `ylim = c(0, 200)`, which will be used for all plots. Alternatively, `ylim` can be a list equal to the length of `panels` to control y-limits for each individual panel, e.g. `ylim = list(c(-100,500), c(200, 300), c(-400,400), c(50,70))`.

Note also that the `timeVariation()` function works well on a subset of data and in conjunction with other plots. For example, a `polarPlot()` may highlight an interesting feature for a particular

wind speed/direction range. By filtering for those conditions `timeVariation()` can help determine whether the temporal variation of that feature differs from other features — and help with source identification.

Value

an openair object. The components of `timeVariation()` are named after `panels`. `main.plot` is a patchwork assembly.

Author(s)

David Carslaw

Jack Davison

See Also

Other time series and trend functions: `TheilSen()`, `calendarPlot()`, `smoothTrend()`, `timePlot()`, `timeProp()`

Examples

```
# basic use
timeVariation(mydata, pollutant = "nox")

# for a subset of conditions
## Not run:
timeVariation(subset(mydata, ws > 3 & wd > 100 & wd < 270),
  pollutant = "pm10", ylab = "pm10 (ug/m3)"
)

# multiple pollutants with concentrations normalised
timeVariation(mydata, pollutant = c("nox", "co"), normalise = TRUE)

# show BST/GMT variation (see ?cutData for more details)
# the NOx plot shows the profiles are very similar when expressed in
# local time, showing that the profile is dominated by a local source
# that varies by local time and not by GMT i.e. road vehicle emissions

timeVariation(mydata, pollutant = "nox", type = "dst", local.tz = "Europe/London")

# In this case it is better to group the results for clarity:
timeVariation(mydata, pollutant = "nox", group = "dst", local.tz = "Europe/London")

# By contrast, a variable such as wind speed shows a clear shift when
#  expressed in local time. These two plots can help show whether the
#  variation is dominated by man-made influences or natural processes

timeVariation(mydata, pollutant = "ws", group = "dst", local.tz = "Europe/London")

# It is also possible to plot several variables and set type. For
# example, consider the NOx and NO2 split by levels of O3:
```

```
timeVariation(mydata, pollutant = c("nox", "no2"), type = "o3", normalise = TRUE)

# difference in concentrations
timeVariation(mydata, poll = c("pm25", "pm10"), difference = TRUE)

# It is also useful to consider how concentrations vary by
# considering two different periods e.g. in intervention
# analysis. In the following plot NO2 has clearly increased but much
# less so at weekends - perhaps suggesting vehicles other than cars
# are important because flows of cars are approximately invariant by
# day of the week

mydata <- splitByDate(mydata, dates = "1/1/2003", labels = c("before Jan. 2003", "After Jan. 2003"))
timeVariation(mydata, pollutant = "no2", group = "split.by", difference = TRUE)

# sub plots can be extracted from the openair object
myplot <- timeVariation(mydata, pollutant = "no2")
myplot$plot$hour.weekday

# individual plots
myplot$plot$hour.weekday
myplot$plot$hour
myplot$plot$day
myplot$plot$month

# numerical results (mean, lower/upper uncertainties)
myplot$data$hour.weekday
myplot$data$hour
myplot$data$day
myplot$data$month

# plot quantiles and median
timeVariation(
  mydata,
  statistic = "median",
  poll = "pm10",
  cols = "firebrick"
)

# with different intervals
timeVariation(
  mydata,
  statistic = "median",
  poll = "pm10",
  conf.int = c(0.75, 0.99),
  cols = "firebrick"
)

# with different (arbitrary) panels
# note 'hemisphere' is passed to cutData() for season
timeVariation(
  mydata,
  pollutant = "no2",
```

```
  panels = c("weekday.season", "year", "wd"),
  hemisphere = "southern"
)

## End(Not run)
```

`trajCluster` *Calculate clusters for back trajectories*

Description

This function carries out cluster analysis of HYSPLIT back trajectories. The function is specifically designed to work with the trajectories imported using the `openair` `importTraj()` function, which provides pre-calculated back trajectories at specific receptor locations.

Usage

```
trajCluster(
  traj,
  method = "Euclid",
  n.cluster = 5,
  type = "default",
  split.after = FALSE,
  by.type = FALSE,
  crs = 4326,
  cols = "Set1",
  plot = TRUE,
  ...
)
```

Arguments

`traj`	An openair trajectory data frame resulting from the use of `importTraj()`.
`method`	Method used to calculate the distance matrix for the back trajectories. There are two methods available: "Euclid" and "Angle".
`n.cluster`	Number of clusters to calculate.
`type`	Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of: • A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season. • The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).

- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`split.after` For `type` other than "default" e.g. "season", the trajectories can either be calculated for each level of `type` independently or extracted after the cluster calculations have been applied to the whole data set.

`by.type` The percentage of the total number of trajectories is given for all data by default. Setting `by.type = TRUE` will make each panel add up to 100.

`crs` The coordinate reference system to use for plotting. Defaults to `4326`, which is the WGS84 geographic coordinate system, the standard, unprojected latitude/longitude system used in GPS, Google Earth, and GIS mapping. Other `crs` values are available - for example, `27700` will use the the OSGB36/British National Grid.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`...` Passed to `trajPlot()`.

Details

Two main methods are available to cluster the back trajectories using two different calculations of the distance matrix. The default is to use the standard Euclidian distance between each pair of trajectories. Also available is an angle-based distance matrix based on Sirois and Bottenheim (1995). The latter method is useful when the interest is the direction of the trajectories in clustering.

The distance matrix calculations are made in C++ for speed. For data sets of up to 1 year both methods should be relatively fast, although the `method = "Angle"` does tend to take much longer to calculate. Further details of these methods are given in the openair manual.

Value

an openair object. The `data` component contains both `traj` (the original data appended with its cluster) and `results` (the average trajectory path per cluster, shown in the `trajCluster()` plot.)

Author(s)

David Carslaw

Jack Davison

References

Sirois, A. and Bottenheim, J.W., 1995. Use of backward trajectories to interpret the 5-year record of PAN and O3 ambient air concentrations at Kejimkujik National Park, Nova Scotia. Journal of Geophysical Research, 100: 2867-2881.

See Also

Other trajectory analysis functions: `importTraj()`, `trajLevel()`, `trajPlot()`

Other cluster analysis functions: `polarCluster()`, `timeProp()`

Examples

```
## Not run:
## import trajectories
traj <- importTraj(site = "london", year = 2009)
## calculate clusters
clust <- trajCluster(traj, n.cluster = 5)
head(clust$data) ## note new variable 'cluster'
## use different distance matrix calculation, and calculate by season
traj <- trajCluster(traj, method = "Angle", type = "season", n.cluster = 4)

## End(Not run)
```

---

trajLevel *Trajectory level plots with conditioning*

Description

This function plots gridded back trajectories. This function requires that data are imported using the `importTraj()` function.

Usage

```
trajLevel(
  mydata,
  lon = "lon",
  lat = "lat",
  pollutant = "height",
  type = "default",
  smooth = FALSE,
  statistic = "frequency",
  percentile = 90,
  lon.inc = 1,
  lat.inc = lon.inc,
  min.bin = 1,
  .combine = NULL,
  sigma = 1.5,
```

```
  cols = "default",
  crs = 4326,
  map = TRUE,
  map.fill = TRUE,
  map.cols = "grey40",
  map.border = "black",
  map.alpha = 0.3,
  map.lwd = 1,
  map.lty = 1,
  grid.col = "deepskyblue",
  grid.nx = 9,
  grid.ny = grid.nx,
  origin = TRUE,
  key.title = NULL,
  key.position = "right",
  key.columns = NULL,
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

`mydata`	Data frame, the result of importing a trajectory file using `importTraj()`.
`lon`, `lat`	Columns containing the decimal longitude and latitude.
`pollutant`	Pollutant (or any numeric column) to be plotted, if any. Alternatively, use `group`.
`type`	Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of: • A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season. • The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4). • The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.
`smooth`	Should the trajectory surface be smoothed?

`statistic` One of:

- `"frequency"` (the default) shows trajectory frequencies.
- `"hexbin"`, which is similar to `"frequency"` but shows a hexagonal grid of counts.
- `"difference"` - in this case trajectories where the associated concentration is greater than `percentile` are compared with the the full set of trajectories to understand the differences in frequencies of the origin of air masses. The comparison is made by comparing the percentage change in gridded frequencies. For example, such a plot could show that the top 10\ to the east.
- `"pscf"` for a Potential Source Contribution Function map. This statistic method interacts with `percentile`.
- `"cwt"` for concentration weighted trajectories.
- `"sqtba"` to undertake Simplified Quantitative Transport Bias Analysis. This statistic method interacts with `.combine` and `sigma`.

`percentile` The percentile concentration of `pollutant` against which the all trajectories are compared.

`lon.inc`, `lat.inc`

The longitude and latitude intervals to be used for binning data. If `statistic = "hexbin"`, the minimum value out of of `lon.inc` and `lat.inc` is passed to the `binwidth` argument of `ggplot2::geom_hex()`.

`min.bin` The minimum number of unique points in a grid cell. Counts below `min.bin` are set as missing.

`.combine` When statistic is "SQTBA" it is possible to combine lots of receptor locations to derive a single map. `.combine` identifies the column that differentiates different sites (commonly a column named `"site"`). Note that individual site maps are normalised first by dividing by their mean value.

`sigma` For the SQTBA approach `sigma` determines the amount of back trajectory spread based on the Gaussian plume equation. Values in the literature suggest 5.4 km after one hour. However, testing suggests lower values reveal source regions more effectively while not introducing too much noise.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`crs` The coordinate reference system to use for plotting. Defaults to `4326`, which is the WGS84 geographic coordinate system, the standard, unprojected latitude/longitude system used in GPS, Google Earth, and GIS mapping. Other `crs` values are available - for example, `27700` will use the the OSGB36/British National Grid.

`map` Should a base map be drawn? If `TRUE` the world base map provided by `ggplot2::map_data()` will be used.

`map.fill` Should the base map be a filled polygon? Default is to fill countries.

map.cols — If map.fill = TRUE map.cols controls the fill colour. Examples include map.fill = "grey40" and map.fill = openColours("default", 10). The latter colours the countries and can help differentiate them.

map.border — The colour to use for the map outlines/borders. Defaults to "black".

map.alpha — The transparency level of the filled map which takes values from 0 (full transparency) to 1 (full opacity). Setting it below 1 can help view trajectories, trajectory surfaces etc. *and* a filled base map.

map.lwd — The map line width, a positive number, defaulting to 1.

map.lty — The map line type. Line types can either be specified as an integer (0 = blank, 1 = solid (default), 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses 'invisible lines' (i.e., does not draw them).

grid.col — The colour of the map grid to be used. To remove the grid set grid.col = "transparent".

grid.nx, grid.ny — The approximate number of ticks to draw on the map grid. grid.nx defaults to 9, and grid.ny defaults to whatever value is passed to grid.nx. Setting both values to 0 will remove the grid entirely. The number of ticks is approximate as this value is passed to scales::breaks_pretty() to determine nice-looking, round breakpoints.

origin — If true a filled circle dot is shown to mark the receptor point.

key.title — Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.

key.position — Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.

key.columns — Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting key.columns to be less than the number of categories.

strip.position — Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

auto.text — Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().

plot — When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

key — Deprecated; please use key.position. If FALSE, sets key.position to "none".

... — Addition options are passed on to cutData() for type handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each tile.

Details

An alternative way of showing the trajectories compared with plotting trajectory lines is to bin the points into latitude/longitude intervals. For these purposes `trajLevel()` should be used. There are several trajectory statistics that can be plotted as gridded surfaces. First, `statistic` can be set to "frequency" to show the number of back trajectory points in a grid square. Grid squares are by default at 1 degree intervals, controlled by `lat.inc` and `lon.inc`. Such plots are useful for showing the frequency of air mass locations. Note that it is also possible to set `statistic = "hexbin"` for plotting frequencies (not concentrations), which will produce a plot by hexagonal binning.

If `statistic = "difference"` the trajectories associated with a concentration greater than `percentile` are compared with the the full set of trajectories to understand the differences in frequencies of the origin of air masses of the highest concentration trajectories compared with the trajectories on average. The comparison is made by comparing the percentage change in gridded frequencies. For example, such a plot could show that the top 10\ the east.

If `statistic = "pscf"` then the Potential Source Contribution Function is plotted. The PSCF calculates the probability that a source is located at latitude i and longitude j (Pekney et al., 2006).The basis of PSCF is that if a source is located at (i,j), an air parcel back trajectory passing through that location indicates that material from the source can be collected and transported along the trajectory to the receptor site. PSCF solves

$$PSCF = m_{ij}/n_{ij}$$

where n_{ij} is the number of times that the trajectories passed through the cell (i,j) and m_{ij} is the number of times that a source concentration was high when the trajectories passed through the cell (i,j). The criterion for determining m_{ij} is controlled by `percentile`, which by default is 90. Note also that cells with few data have a weighting factor applied to reduce their effect.

A limitation of the PSCF method is that grid cells can have the same PSCF value when sample concentrations are either only slightly higher or much higher than the criterion. As a result, it can be difficult to distinguish moderate sources from strong ones. Seibert et al. (1994) computed concentration fields to identify source areas of pollutants. The Concentration Weighted Trajectory (CWT) approach considers the concentration of a species together with its residence time in a grid cell. The CWT approach has been shown to yield similar results to the PSCF approach. The openair manual has more details and examples of these approaches.

A further useful refinement is to smooth the resulting surface, which is possible by setting `smooth = TRUE`.

Value

an openair object

Author(s)

David Carslaw

Jack Davison

References

Pekney, N. J., Davidson, C. I., Zhou, L., & Hopke, P. K. (2006). Application of PSCF and CPF to PMF-Modeled Sources of PM 2.5 in Pittsburgh. Aerosol Science and Technology, 40(10), 952-961.

Seibert, P., Kromp-Kolb, H., Baltensperger, U., Jost, D., 1994. Trajectory analysis of high-alpine air pollution data. NATO Challenges of Modern Society 18, 595-595.

Xie, Y., & Berkowitz, C. M. (2007). The use of conditional probability functions and potential source contribution functions to identify source regions and advection pathways of hydrocarbon emissions in Houston, Texas. Atmospheric Environment, 41(28), 5831-5847.

See Also

Other trajectory analysis functions: `importTraj()`, `trajCluster()`, `trajPlot()`

Examples

```
# show a simple case with no pollutant i.e. just the trajectories
# let's check to see where the trajectories were coming from when
# Heathrow Airport was closed due to the Icelandic volcanic eruption
# 15--21 April 2010.
# import trajectories for London and plot
## Not run:
lond <- importTraj("london", 2010)

## End(Not run)
# more examples to follow linking with concentration measurements...

# import some measurements from KC1 - London
## Not run:
kc1 <- importAURN("kc1", year = 2010)
# now merge with trajectory data by 'date'
lond <- merge(lond, kc1, by = "date")

# trajectory plot, no smoothing - and limit lat/lon area of interest
# use PSCF
trajLevel(subset(lond, lat > 40 & lat < 70 & lon > -20 & lon < 20),
  pollutant = "pm10", statistic = "pscf"
)

# can smooth surface, suing CWT approach:
trajLevel(subset(lond, lat > 40 & lat < 70 & lon > -20 & lon < 20),
  pollutant = "pm2.5", statistic = "cwt", smooth = TRUE
)

# plot by season:
trajLevel(subset(lond, lat > 40 & lat < 70 & lon > -20 & lon < 20),
  pollutant = "pm2.5",
  statistic = "pscf", type = "season"
)
```

```
## End(Not run)
```

`trajPlot` *Trajectory line plots with conditioning*

Description

This function plots back trajectories. This function requires that data are imported using the `importTraj()` function, or matches that structure.

Usage

```
trajPlot(
  mydata,
  lon = "lon",
  lat = "lat",
  pollutant = NULL,
  type = "default",
  map = TRUE,
  group = NULL,
  cols = "default",
  crs = 4326,
  map.fill = TRUE,
  map.cols = "grey40",
  map.border = "black",
  map.alpha = 0.4,
  map.lwd = 1,
  map.lty = 1,
  grid.col = "deepskyblue",
  grid.nx = 9,
  grid.ny = grid.nx,
  npoints = 12,
  origin = TRUE,
  key.title = group,
  key.position = "right",
  key.columns = 1,
  strip.position = "top",
  auto.text = TRUE,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

`mydata`	Data frame, the result of importing a trajectory file using `importTraj()`.

`lon`, `lat` Columns containing the decimal longitude and latitude.

`pollutant` Pollutant (or any numeric column) to be plotted, if any. Alternatively, use `group`.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`map` Should a base map be drawn? If `TRUE` the world base map provided by `ggplot2::map_data()` will be used.

`group` A condition to colour the plot by, passed to `cutData()`. An alternative to `pollutant`, and used preferentially to `pollutant` if both are set.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`crs` The coordinate reference system to use for plotting. Defaults to `4326`, which is the WGS84 geographic coordinate system, the standard, unprojected latitude/longitude system used in GPS, Google Earth, and GIS mapping. Other `crs` values are available - for example, `27700` will use the the OSGB36/British National Grid.

`map.fill` Should the base map be a filled polygon? Default is to fill countries.

`map.cols` If `map.fill = TRUE` `map.cols` controls the fill colour. Examples include `map.fill = "grey40"` and `map.fill = openColours("default", 10)`. The latter colours the countries and can help differentiate them.

`map.border` The colour to use for the map outlines/borders. Defaults to `"black"`.

`map.alpha` The transparency level of the filled map which takes values from 0 (full transparency) to 1 (full opacity). Setting it below 1 can help view trajectories, trajectory surfaces etc. *and* a filled base map.

`map.lwd` The map line width, a positive number, defaulting to `1`.

`map.lty` The map line type. Line types can either be specified as an integer (`0` = blank, `1` = solid (default), `2` = dashed, `3` = dotted, `4` = dotdash, `5` = longdash, `6` =

twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses 'invisible lines' (i.e., does not draw them).

`grid.col` The colour of the map grid to be used. To remove the grid set `grid.col = "transparent"`.

`grid.nx, grid.ny`
The approximate number of ticks to draw on the map grid. `grid.nx` defaults to `9`, and `grid.ny` defaults to whatever value is passed to `grid.nx`. Setting both values to `0` will remove the grid entirely. The number of ticks is approximate as this value is passed to `scales::breaks_pretty()` to determine nice-looking, round breakpoints.

`npoints` A dot is placed every `npoints` along each full trajectory. For hourly back trajectories points are plotted every `npoint` hours. This helps to understand where the air masses were at particular times and get a feel for the speed of the air (points closer together correspond to slower moving air masses). If `npoints = NA` then no points are added.

`origin` If true a filled circle dot is shown to mark the receptor point.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`key.columns` Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each bar.

Details

Several types of trajectory plot are available:

- `trajPlot()` by default will plot each lat/lon location showing the origin of each trajectory, if no `pollutant` is supplied.
- If a pollutant is given, by merging the trajectory data with concentration data, the trajectories are colour-coded by the concentration of `pollutant`. With a long time series there can be lots of overplotting making it difficult to gauge the overall concentration pattern. In these cases setting `alpha` to a low value e.g. 0.1 can help.

The user can also show points instead of lines by `plot.type = "p"`.

Note that `trajPlot()` will plot only the full length trajectories. This should be remembered when selecting only part of a year to plot.

Author(s)

David Carslaw

Jack Davison

See Also

Other trajectory analysis functions: `importTraj()`, `trajCluster()`, `trajLevel()`

Examples

```
## Not run:
# show a simple case with no pollutant i.e. just the trajectories
# let's check to see where the trajectories were coming from when
# Heathrow Airport was closed due to the Icelandic volcanic eruption
# 15--21 April 2010.
# import trajectories for London and plot

lond <- importTraj("london", 2010)

# well, HYSPLIT seems to think there certainly were conditions where trajectories
# orginated from Iceland...
trajPlot(selectByDate(lond, start = "15/4/2010", end = "21/4/2010"))

# plot by day, need a column that makes a date
lond$day <- as.Date(lond$date)
trajPlot(
  selectByDate(lond, start = "15/4/2010", end = "21/4/2010"),
  type = "day"
)

# or show each day grouped by colour, with some other options set
trajPlot(
  selectByDate(lond, start = "15/4/2010", end = "21/4/2010"),
  group = "day",
  cols = "turbo",
```

```
  key.position = "right",
  key.columns = 1,
  lwd = 2,
  cex = 4
)

## End(Not run)
```

`trendLevel` *Plot heat maps of atmospheric composition data*

Description

The `trendLevel()` function provides a way of rapidly showing a large amount of data in a condensed form. In one plot, the variation in the concentration of one pollutant can to shown as a function of between two and four categorical properties. The default arguments plot hour of day on the x-axis and month of year on the y-axis. However, `x`, `y` and `type` and summarising statistics can all be modified to provide a range of other similar plots, all being passed to `cutData()` for discretisation. The average wind speed and direction in each bin can also be plotted using the `windflow` argument.

Usage

```
trendLevel(
  mydata,
  pollutant = "nox",
  x = "month",
  y = "hour",
  type = "default",
  rotate.axis = c(90, 0),
  n.levels = c(10, 10, 4),
  windflow = NULL,
  limits = NULL,
  min.bin = 1,
  cols = "default",
  auto.text = TRUE,
  key.title = paste("use.stat.name", pollutant, sep = " "),
  key.position = "right",
  strip.position = "top",
  labels = NULL,
  breaks = NULL,
 statistic = c("mean", "max", "min", "median", "frequency", "sum", "sd", "percentile"),
  percentile = 95,
  stat.args = NULL,
  stat.safe.mode = TRUE,
  drop.unused.types = TRUE,
  col.na = "white",
```

```
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	The openair data frame to use to generate the trendLevel() plot.
pollutant	The name of the data series in mydata to sample to produce the trendLevel() plot.
x, y, type	The name of the data series to use as the trendLevel() x-axis, y-axis or conditioning variable, passed to cutData(). These are used before applying statistic. trendLevel() does not allow duplication in x, y and type options.
rotate.axis	The rotation to be applied to trendLevel x and y axes. The default, c(90, 0), rotates the x axis by 90 degrees but does not rotate the y axis. If only one value is supplied, this is applied to both axes; if more than two values are supplied, only the first two are used.
n.levels	The number of levels to split x, y and type data into if numeric. The default, c(10, 10, 4), cuts numeric x and y data into ten levels and numeric type data into four levels. This option is ignored for date conditioning and factors. If less than three values are supplied, three values are determined by recursion; if more than three values are supplied, only the first three are used.
windflow	If TRUE, the vector-averaged wind speed and direction will be plotted using arrows. Alternatively, can be a list of arguments to control the appearance of the arrows (colour, linewidth, alpha value, etc.). See windflowOpts() for details.
limits	The colour scale range to use when generating the trendLevel() plot.
min.bin	The minimum number of records required in a bin to show a value. Bins with fewer than min.bin records are set to NA. The default is 1, i.e., all bins with no records are set to NA. Setting min.bin to a value greater than 1 can be useful to exclude bins with very few records that might produce unreliable statistic values.
cols	Colours to use for plotting. Can be a pre-set palette (e.g., "turbo", "viridis", "tol", "Dark2", etc.) or a user-defined vector of R colours (e.g., c("yellow", "green", "blue", "black") - see colours() for a full list) or hex-codes (e.g., c("#30123B", "#9CF649", "#7A0403")). See openColours() for more details.
auto.text	Either TRUE (default) or FALSE. If TRUE titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to quickText().
key.title	Used to set the title of the legend. The legend title is passed to quickText() if auto.text = TRUE.
key.position	Location where the legend is to be placed. Allowed arguments include "top", "right", "bottom", "left" and "none", the last of which removes the legend entirely.
strip.position	Location where the facet 'strips' are located when using type. When one type is provided, can be one of "left", "right", "bottom" or "top". When two

types are provided, this argument defines whether the strips are "switched" and can take either "x", "y", or "both". For example, "x" will switch the 'top' strip locations to the bottom of the plot.

breaks, labels
: If a categorical colour scale is required then breaks should be specified. These should be provided as a numeric vector, e.g., breaks = c(0, 50, 100, 1000). Users should set the maximum value of breaks to exceed the maximum data value to ensure it is within the maximum final range, e.g., 100–1000 in this case. Labels will automatically be generated, but can be customised by passing a character vector to labels, e.g., labels = c("good", "bad", "very bad"). In this example, 0 - 50 will be "good" and so on. Note there is one less label than break.

statistic
: The statistic to apply when aggregating the data; default is the mean. Can be one of "mean", "max", "min", "median", "frequency", "sum", "sd", "percentile". Note that "sd" is the standard deviation, "frequency" is the number (frequency) of valid records in the period and "data.cap" is the percentage data capture. "percentile" is the percentile level (%) between 0-100, which can be set using the "percentile" option. Functions can also be sent directly via statistic; see 'Details' for more information.

percentile
: The percentile level used when statistic = "percentile". The default is 95%.

stat.args
: Additional options to be used with statistic if this is a function. The extra options should be supplied as a list of named parameters; see 'Details' for more information.

stat.safe.mode
: An addition protection applied when using functions directly with statistic that most users can ignore. This option returns NA instead of running statistic on binned sub samples that are empty. Many common functions terminate with an error message when applied to an empty dataset. So, this option provides a mechanism to work with such functions. For a very few cases, e.g., for a function that counted missing entries, it might need to be set to FALSE; see 'Details' for more information.

drop.unused.types
: Hide unused/empty type conditioning cases. Some conditioning options may generate empty cases for some data sets, e.g. a hour of the day when no measurements were taken. Empty x and y cases generate 'holes' in individual plots. However, empty type cases would produce blank panels if plotted. Therefore, the default, TRUE, excludes these empty panels from the plot. The alternative FALSE plots all type panels.

col.na
: Colour to be used to show missing data.

plot
: When openair plots are created they are automatically printed to the active graphics device. plot = FALSE deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

key
: Deprecated; please use key.position. If FALSE, sets key.position to "none".

...
: Addition options are passed on to cutData() for type handling. Some additional arguments are also available:

  - xlab, ylab and main override the x-axis label, y-axis label, and plot title.

- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.
- `border` sets the border colour of each tile.

Details

`trendLevel()` allows the use of third party summarising functions via the `statistic` option. Any additional function arguments not included within a function called using `statistic` should be supplied as a list of named parameters and sent using `stat.args`. For example, the encoded option `statistic = "mean"` is equivalent to `statistic = mean, stat.args = list(na.rm = TRUE)` or the R command `mean(x, na.rm = TRUE)`. Many R functions and user's own code could be applied in a similar fashion, subject to the following restrictions: the first argument sent to the function must be the data series to be analysed; the name 'x' cannot be used for any of the extra options supplied in `stat.args`; and the function should return the required answer as a numeric or `NA`. Note: If the supplied function returns more than one answer, currently only the first of these is retained and used by `trendLevel()`. All other returned information will be ignored without warning. If the function terminates with an error when it is sent an empty data series, the option `stat.safe.mode` should not be set to `FALSE` or `trendLevel()` may fail. Note: The `stat.safe.mode = TRUE` option returns an NA without warning for empty data series.

Value

an openair object.

Author(s)

Karl Ropkins

David Carslaw

Jack Davison

Examples

```
# basic use
# default statistic = "mean"
trendLevel(mydata, pollutant = "nox")

# applying same as 'own' statistic
my.mean <- function(x) mean(x, na.rm = TRUE)
trendLevel(mydata, pollutant = "nox", statistic = my.mean)

# alternative for 'third party' statistic
# trendLevel(mydata, pollutant = "nox", statistic = mean,
#           stat.args = list(na.rm = TRUE))

## Not run:
# example with categorical scale
trendLevel(mydata,
  pollutant = "no2",
  border = "white", statistic = "max",
```

```
  breaks = c(0, 50, 100, 500),
  labels = c("low", "medium", "high"),
  cols = c("forestgreen", "yellow", "red")
)

## End(Not run)
```

`variationPlot` *Variation Plot*

Description

The `variationPlot()` function is designed to explore how the distribution of a pollutant (or other variable) changes by another variable (`x`). For example, it can be used to explore how the distribution of `nox` varies by `season` or by `weekday`. This plot can be extensively conditioned using the `type` and `group` arguments, both of which are passed to `cutData()`. An appropriate plot type will be chosen based on the type of `x` - e.g., ordered variables will be joined by a line.

Usage

```
variationPlot(
  mydata,
  pollutant = "nox",
  x = "hour",
  statistic = "mean",
  type = "default",
  group = "default",
  normalise = FALSE,
  difference = FALSE,
  conf.int = NULL,
  B = 100,
  local.tz = NULL,
  ci = TRUE,
  cols = "hue",
  alpha = 0.4,
  strip.position = "top",
  key.position = "top",
  key.columns = NULL,
  name.pol = NULL,
  auto.text = TRUE,
  plot = TRUE,
  ...
)
```

Arguments

`mydata`	A data frame of time series. Must include a `date` field and at least one variable to plot.

`pollutant` Name of variable to plot. Two or more pollutants can be plotted, in which case a form like `pollutant = c("nox", "co")` should be used.

`x` A character value to be passed to `cutData()`; used to define the category by which `pollutant` will be varied and plotted.

`statistic` Can be `"mean"` (default) or `"median"`. If the statistic is `"mean"` then the mean line and the 95% confidence interval in the mean are plotted by default. If the statistic is `"median"` then the median line is plotted together with the 5/95 and 25/75th quantiles are plotted. Users can control the confidence intervals with `conf.int`.

`type` Character string(s) defining how data should be split/conditioned before plotting. `"default"` produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one `type` is given, or a 2D matrix of panels if two `types` are given. `type` is always passed to `cutData()`, and can therefore be any of:

- A built-in type defined in `cutData()` (e.g., `"season"`, `"year"`, `"weekday"`, etc.). For example, `type = "season"` will split the plot into four panels, one for each season.
- The name of a numeric column in `mydata`, which will be split into `n.levels` quantiles (defaulting to 4).
- The name of a character or factor column in `mydata`, which will be used as-is. Commonly this could be a variable like `"site"` to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not).

Most `openair` plotting functions can take two `type` arguments. If two are given, the first is used for the columns and the second for the rows.

`group` This sets the grouping variable to be used. For example, if a data frame had a column `site` setting `group = "site"` will plot all sites together in each panel. Passed to `cutData()`.

`normalise` Should variables be normalised? The default is `FALSE`. If `TRUE` then the variable(s) are divided by their mean values. This helps to compare the shape of the diurnal trends for variables on very different scales.

`difference` If two pollutants are chosen then setting `difference = TRUE` will also plot the difference in means between the two variables as `pollutant[2] - pollutant[1]`. Bootstrap 95\ difference in means are also calculated. A horizontal dashed line is shown at y = 0. The difference can also be calculated if there is a column that identifies two groups, e.g., having used `splitByDate()`. In this case it is possible to call the function with the option `group = "split.by"` and `difference = TRUE`.

`conf.int` The confidence intervals to be plotted. If `statistic = "mean"` then the confidence intervals in the mean are plotted. If `statistic = "median"` then the `conf.int` and `1 - conf.int` *quantiles* are plotted. Any number of `conf.ints` can be provided.

`B` Number of bootstrap replicates to use. Can be useful to reduce this value when there are a large number of observations available to increase the speed of the calculations without affecting the 95% confidence interval calculations by much.

`local.tz` Used for identifying whether a date has daylight savings time (DST) applied or not. Examples include `local.tz = "Europe/London"`, `local.tz = "America/New_York"`, i.e., time zones that assume DST. https://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones shows time zones that should be valid for most systems. It is important that the original data are in GMT (UTC) or a fixed offset from GMT.

`ci` Should confidence intervals be shown? The default is `TRUE`. Setting this to `FALSE` can be useful if multiple pollutants are chosen where over-lapping confidence intervals can over complicate plots.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`alpha` The alpha transparency used for plotting confidence intervals. `0` is fully transparent and 1 is opaque. The default is `0.4`.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`key.columns` Number of columns to be used in a categorical legend. With many categories a single column can make to key too wide. The user can thus choose to use several columns by setting `key.columns` to be less than the number of categories.

`name.pol` This option can be used to give alternative names for the variables plotted. Instead of taking the column headings as names, the user can supply replacements. For example, if a column had the name "nox" and the user wanted a different description, then setting `name.pol = "nox before change"` can be used. If more than one pollutant is plotted then use `c` e.g. `name.pol = c("nox here", "o3 there")`.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.

- `lwd` and `lty` control various graphical parameters.
- `fontsize` overrides the overall font size of the plot.
- `ylim` controls axis limits.

Details

When `statistic = "mean"`, the plot shows the 95% confidence intervals in the mean. The 95% confidence intervals are calculated through bootstrap simulations, which will provide more robust estimates of the confidence intervals (particularly when there are relatively few data).

Users can supply their own `ylim`, e.g. `ylim = c(0, 200)`.

The `difference` option calculates the difference in means between two pollutants, along with bootstrap estimates of the 95\ in the difference. This works in two ways: either two pollutants are supplied in separate columns (e.g. `pollutant = c("no2", "o3")`), or there are two unique values of `group`. The difference is calculated as the second pollutant minus the first and is labelled accordingly. This feature is particularly useful for model evaluation and identifying where models diverge from observations across time scales.

Depending on the choice of statistic, a subheading is added. Users can control the text in the subheading through the use of `sub` e.g. `sub = ""` will remove any subheading.

Value

an openair object.

Author(s)

Jack Davison

David Carslaw

See Also

`timeVariation()`, which conveniently assembles many time-related variation plots into a single plot

Examples

```
# example using the 'mydata' dataset
variationPlot(
  mydata,
  pollutant = c("nox", "o3"),
  x = "hour",
  type = "season",
  normalise = TRUE
)
```

`WhittakerSmooth` *Calculate Whittaker-Eilers Smoothing, Interpolation and Baseline Determination*

Description

This function applies the Whittaker-Eilers smoothing and interpolation method to a specified pollutant in a data frame. The method is based on penalised least squares and is designed to handle time series data with missing values, providing a smoothed estimate of the pollutant concentrations over time. The function allows for flexible control over the amount of smoothing through the `lambda` parameter and can be applied to multiple pollutants simultaneously.

Usage

```
WhittakerSmooth(
  mydata,
  pollutant = "o3",
  lambda = 24L,
  d = 2,
  type = "default",
  new.name = NULL,
  date.pad = FALSE,
  p = NULL,
  ...
)
```

Arguments

`mydata`	A data frame containing a `date` field. `mydata` must contain a `date` field in `Date` or `POSIXct` format.
`pollutant`	The name of a pollutant, e.g., `pollutant = "o3"`. More than one pollutant can be supplied as a vector, e.g., `pollutant = c("o3", "nox")`.
`lambda`	The value of `lambda` to use in the smoothing. This controls the amount of smoothing, with higher values leading to smoother results. If `lambda = NA` Generalised Cross Validation (GCV) is used to select the optimal value of `lambda` for each pollutant. This can be time consuming, so a fixed value of `lambda` is recommended for large datasets or multiple pollutants. Note that the value of `lambda` needs to increase exponentially to smooth long time series data of several years e.g. `lambda = 10e9`.
`d`	The order used to penalise the roughness of the data. By default this is set to 2, which penalises the second derivative of the data. Setting `d = 1` will penalise the first derivative, which can be useful for smoothing data with sharp peaks or troughs. Setting `d = 1` will effectively linearly interpolate across missing data.
`type`	Used for splitting the data further. Passed to `cutData()`.
`new.name`	The name given to the new column(s). If not supplied it will create a name based on the name of the pollutant.

date.pad	Should missing dates be padded? Default is FALSE.
p	The asymmetry weight parameter used exclusively for baseline estimation (Asymmetric Least Squares). It defines how the algorithm treats points that fall above the fitted line versus points that fall below it. It takes a value between 0 and 1. When p is very small, the algorithm assigns a massive penalty to the curve if it rises above the data points, but almost no penalty if it drops below them. This forces the curve to "hug" the bottom of the signal, effectively ignoring the positive peaks. Typical Values: 0.01 to 0.05.
...	Additional parameters passed to cutData(). For use with type.

Details

In addition to smoothing, the function can also perform baseline estimation using Asymmetric Least Squares (ALS) when the p parameter is provided. This allows for the separation of the underlying baseline from the observed data, which can be particularly useful for identifying trends or correcting for background levels in pollutant concentrations.

The function is designed to work with regularly spaced time series.

Value

A tibble with new columns for the smoothed pollutant values.

Author(s)

David Carslaw

References

Paul H. C. Eilers, A Perfect Smoother, Analytical Chemistry 2003 75 (14), 3631-3636, DOI: 10.1021/ac034173t

Examples

```
# Smoothing with lambda = 24
mydata <- WhittakerSmooth(mydata, pollutant = "o3", lambda = 24)
```

---

windflowOpts *Define* windflow *options for* openair *plots*

Description

This function provides a convenient way to set default options for windflow layers in openair plots, which typically show the mean wind speed and direction as compass arrows. It returns a list of options that can be passed to layer_windflow().

Usage

```
windflowOpts(
  limits = c(NA, NA),
  range = c(0.1, 1),
  arrow.angle = 15,
  arrow.length = ggplot2::unit(0.5, "lines"),
  arrow.ends = "last",
  arrow.type = "closed",
  lineend = "butt",
  alpha = 1,
  colour = "black",
  linetype = 1,
  linewidth = 0.5,
  color = NULL,
  windflow = TRUE
)
```

Arguments

limits	Numeric vector of length 2 specifying the limits for wind speed. By default, it is set to c(NA, NA), which means the limits will be determined automatically based on the data.
range	Numeric vector of length 2 specifying the range of possible sizes of the windflow arrows. The default is broadly appropriate throughout openair, but if plots are beings saved at different resolutions it may be appropriate to to tweak this.
arrow.angle, arrow.length, arrow.ends, arrow.type	Passed to the respective arguments of grid::arrow(); used to control various arrow styling options.
lineend	The style of line endings. Options include "butt", "round", and "square". Default is "butt".
alpha	Numeric value between 0 and 1 specifying the transparency of the lines. Default is 1 (fully opaque).
colour, color	Colour of the lines. Default is "black". colour and color are interchangeable, but colour is used preferentially if both are given.
linetype	Line type. Can be an integer (e.g., 1 for solid, 2 for dashed) or a string (e.g., "solid", "dashed"). Default is 1 (solid).
linewidth	Numeric value specifying the width of the lines. Default is 0.5.
windflow	Logical value indicating whether to include the windflow layer. Default is TRUE. Used internally by openair functions.
arrow	A grid::arrow() object specifying the appearance of the arrows.

Value

A list of options that can be passed to the windflow argument of functions like trendLevel().

Examples

```
# `windflow` can be `TRUE` to use defaults
trendLevel(mydata, type = "default", cols = "greyscale", windflow = TRUE)

# use the `windflowOpts()` function to customise arrows
trendLevel(
  mydata,
  type = "default",
  cols = "greyscale",
  windflow = windflowOpts(
    colour = "white",
    alpha = 0.8,
    linewidth = 0.25,
    linetype = 2
  )
)
```

`windRose` *Traditional wind rose plot*

Description

The traditional wind rose plot that plots wind speed and wind direction by different intervals. The pollution rose applies the same plot structure but substitutes other measurements, most commonly a pollutant time series, for wind speed.

Usage

```
windRose(
  mydata,
  ws = "ws",
  wd = "wd",
  ws2 = NA,
  wd2 = NA,
  ws.int = 2,
  angle = 30,
  type = "default",
  calm.thresh = 0,
  bias.corr = TRUE,
  cols = "default",
  grid.line = NULL,
  width = 0.9,
  seg = 0.9,
  auto.text = TRUE,
  breaks = 4,
  offset = 10,
  normalise = FALSE,
  max.freq = NULL,
```

```
  paddle = TRUE,
  key.title = "(m/s)",
  key.position = "bottom",
  strip.position = "top",
  dig.lab = 5,
  include.lowest = FALSE,
  statistic = "prop.count",
  pollutant = NULL,
  annotate = TRUE,
  angle.scale = 315,
  border = NA,
  plot = TRUE,
  key = NULL,
  ...
)
```

Arguments

mydata	A data frame containing fields ws and wd
ws	Name of the column representing wind speed.
wd	Name of the column representing wind direction.
ws2, wd2	The user can supply a second set of wind speed and wind direction values with which the first can be compared. See pollutionRose() for more details.
ws.int	The Wind speed interval. Default is 2 m/s but for low met masts with low mean wind speeds a value of 1 or 0.5 m/s may be better.
angle	Default angle of "spokes" is 30. Other potentially useful angles are 45 and 10. Note that the width of the wind speed interval may need adjusting using width.
type	Character string(s) defining how data should be split/conditioned before plotting. "default" produces a single panel using the entire dataset. Any other options will split the plot into different panels - a roughly square grid of panels if one type is given, or a 2D matrix of panels if two types are given. type is always passed to cutData(), and can therefore be any of: • A built-in type defined in cutData() (e.g., "season", "year", "weekday", etc.). For example, type = "season" will split the plot into four panels, one for each season. • The name of a numeric column in mydata, which will be split into n.levels quantiles (defaulting to 4). • The name of a character or factor column in mydata, which will be used as-is. Commonly this could be a variable like "site" to ensure data from different monitoring sites are handled and presented separately. It could equally be any arbitrary column created by the user (e.g., whether a nearby possible pollutant source is active or not). Most openair plotting functions can take two type arguments. If two are given, the first is used for the columns and the second for the rows.
calm.thresh	By default, conditions are considered to be calm when the wind speed is zero. The user can set a different threshold for calms be setting calm.thresh to a

higher value. For example, `calm.thresh = 0.5` will identify wind speeds **below** 0.5 as calm.

`bias.corr` When `angle` does not divide exactly into 360 a bias is introduced in the frequencies when the wind direction is already supplied rounded to the nearest 10 degrees, as is often the case. For example, if `angle = 22.5`, N, E, S, W will include 3 wind sectors and all other angles will be two. A bias correction can made to correct for this problem. A simple method according to Applequist (2012) is used to adjust the frequencies.

`cols` Colours to use for plotting. Can be a pre-set palette (e.g., `"turbo"`, `"viridis"`, `"tol"`, `"Dark2"`, etc.) or a user-defined vector of R colours (e.g., `c("yellow", "green", "blue", "black")` - see `colours()` for a full list) or hex-codes (e.g., `c("#30123B", "#9CF649", "#7A0403")`). See `openColours()` for more details.

`grid.line` Grid line interval to use. If `NULL`, as in default, this is assigned based on the available data range. However, it can also be forced to a specific value, e.g. `grid.line = 10`. `grid.line` can also be a list to control the interval, line type and colour. For example `grid.line = list(value = 10, lty = 5, col = "purple")`.

`width` For paddle = TRUE, the adjustment factor for width of wind speed intervals. For example, width = 1.5 will make the paddle width 1.5 times wider.

`seg` `seg` determines with width of the segments. For example, `seg = 0.5` will produce segments 0.5 * `angle`.

`auto.text` Either `TRUE` (default) or `FALSE`. If `TRUE` titles and axis labels will automatically try and format pollutant names and units properly, e.g., by subscripting the "2" in "NO2". Passed to `quickText()`.

`breaks` Most commonly, the number of break points for wind speed. With the `ws.int` default of 2 m/s, the `breaks` default, 4, generates the break points 2, 4, 6, 8 m/s. However, `breaks` can also be used to set specific break points. For example, the argument `breaks = c(0, 1, 10, 100)` breaks the data into segments <1, 1-10, 10-100, >100.

`offset` `offset` controls the size of the 'hole' in the middle and is expressed on a scale of `0` to `100`, where `0` is no hole and `100` is a hole that takes up the entire plotting area.

`normalise` If `TRUE` each wind direction segment is normalised to equal one. This is useful for showing how the concentrations (or other parameters) contribute to each wind sector when the proportion of time the wind is from that direction is low. A line showing the probability that the wind directions is from a particular wind sector is also shown.

`max.freq` Controls the scaling used by setting the maximum value for the radial limits. This is useful to ensure several plots use the same radial limits.

`paddle` Either `TRUE` or `FALSE`. If `TRUE` plots rose using 'paddle' style spokes. If `FALSE` plots rose using 'wedge' style spokes.

`key.title` Used to set the title of the legend. The legend title is passed to `quickText()` if `auto.text = TRUE`.

`key.position` Location where the legend is to be placed. Allowed arguments include `"top"`, `"right"`, `"bottom"`, `"left"` and `"none"`, the last of which removes the legend entirely.

`strip.position` Location where the facet 'strips' are located when using `type`. When one `type` is provided, can be one of `"left"`, `"right"`, `"bottom"` or `"top"`. When two `types` are provided, this argument defines whether the strips are "switched" and can take either `"x"`, `"y"`, or `"both"`. For example, `"x"` will switch the 'top' strip locations to the bottom of the plot.

`dig.lab` The number of significant figures at which scientific number formatting is used in break point and key labelling. Default 5.

`include.lowest` Logical. If `FALSE` (the default), the first interval will be left exclusive and right inclusive. If `TRUE`, the first interval will be left and right inclusive. Passed to the `include.lowest` argument of `cut()`.

`statistic` The `statistic` to be applied to each data bin in the plot. Options currently include "prop.count", "prop.mean" and "abs.count". The default "prop.count" sizes bins according to the proportion of the frequency of measurements. Similarly, "prop.mean" sizes bins according to their relative contribution to the mean. "abs.count" provides the absolute count of measurements in each bin.

`pollutant` Alternative data series to be sampled instead of wind speed. The `windRose()` default NULL is equivalent to `pollutant = "ws"`. Use in `pollutionRose()`.

`annotate` If `TRUE` then the percentage calm and mean values are printed in each panel together with a description of the statistic below the plot. If `FALSE` then only the statistic will be printed.

`angle.scale` In radial plots (e.g., `polarPlot()`), the radial scale is drawn directly on the plot itself. While suitable defaults have been chosen, sometimes the placement of the scale may interfere with an interesting feature. `angle.scale` can take any value between `0` and `360` to place the scale at a different angle, or `FALSE` to move it to the side of the plots.

`border` Border colour for shaded areas. Default is no border.

`plot` When `openair` plots are created they are automatically printed to the active graphics device. `plot = FALSE` deactivates this behaviour. This may be useful when the plot *data* is of more interest, or the plot is required to appear later (e.g., later in a Quarto document, or to be saved to a file).

`key` Deprecated; please use `key.position`. If `FALSE`, sets `key.position` to `"none"`.

`...` Addition options are passed on to `cutData()` for `type` handling. Some additional arguments are also available:

- `xlab`, `ylab` and `main` override the x-axis label, y-axis label, and plot title.
- `layout` sets the layout of facets - e.g., `layout(2, 5)` will have 2 columns and 5 rows.
- `fontsize` overrides the overall font size of the plot.

Details

For `windRose` data are summarised by direction, typically by 45 or 30 (or 10) degrees and by different wind speed categories. Typically, wind speeds are represented by different width "paddles". The plots show the proportion (here represented as a percentage) of time that the wind is from a certain angle and wind speed range.

By default windRose will plot a windRose in using "paddle" style segments and placing the scale key below the plot.

The argument pollutant uses the same plotting structure but substitutes another data series, defined by pollutant, for wind speed. It is recommended to use pollutionRose() for plotting pollutant concentrations.

The option statistic = "prop.mean" provides a measure of the relative contribution of each bin to the panel mean, and is intended for use with pollutionRose.

Value

an openair object. Summarised proportions can be extracted directly using the $data operator, e.g. object$data for output <- windRose(mydata). This returns a data frame with three set columns: cond, conditioning based on type; wd, the wind direction; and calm, the statistic for the proportion of data unattributed to any specific wind direction because it was collected under calm conditions; and then several (one for each range binned for the plot) columns giving proportions of measurements associated with each ws or pollutant range plotted as a discrete panel.

Author(s)

David Carslaw

Karl Ropkins

Jack Davison

References

Applequist, S, 2012: Wind Rose Bias Correction. J. Appl. Meteor. Climatol., 51, 1305-1309.

Droppo, J.G. and B.A. Napier (2008) Wind Direction Bias in Generating Wind Roses and Conducting Sector-Based Air Dispersion Modeling, Journal of the Air & Waste Management Association, 58:7, 913-918.

See Also

Other polar directional analysis functions: percentileRose(), polarAnnulus(), polarCluster(), polarDiff(), polarFreq(), polarPlot(), pollutionRose()

Examples

```
# basic plot
windRose(mydata)

# one windRose for each year
windRose(mydata, type = "year")

# windRose in 10 degree intervals with gridlines and width adjusted
## Not run:
windRose(mydata, angle = 10, width = 0.2, grid.line = 1)

## End(Not run)
```

Index