Package 'openair'

June 18, 2020

Type Package

Title Tools for the Analysis of Air Pollution Data

Version 2.7-4

Date 2020-06-17

ByteCompile true

Depends R (>= 3.2.0),

Imports grid, rlang, dplyr (>= 1.0), purrr, tidyr, readr, mgcv, lattice, latticeExtra, lubridate, cluster, mapproj, hexbin, Rcpp, grDevices, graphics, methods, stats, MASS, utils

Suggests KernSmooth, maps, mapdata, quantreg

LinkingTo Rcpp

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

Description Tools to analyse, interpret and understand air pollution data. Data are typically hourly time series and both monitoring data and dispersion model output can be analysed. Many functions can also be applied to other data, including meteorological and traffic data.

License GPL (>= 2)

URL http://davidcarslaw.github.io/openair/

BugReports https://github.com/davidcarslaw/openair/issues

LazyLoad yes

LazyData yes

Encoding UTF-8

RoxygenNote 7.1.0

NeedsCompilation yes

Author David Carslaw [aut, cre], Karl Ropkins [aut]

Repository CRAN

Date/Publication 2020-06-18 17:40:02 UTC

R topics documented:

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

Description

Calculate a range of air pollution-relevant statistics by year.

Usage

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

Arguments

`mydata`	A data frame containing a `date` field of hourly data.
`pollutant`	The name of a pollutant e.g. `pollutant = c("o3","pm10")`.
`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.
`data.thresh`	The data capture threshold in if data capture over the period of interest is less than this value. `data.thresh` is used for example in the calculation of daily mean values from hourly data. If there are less than `data.thresh` percentage of measurements available in a period, `NA` is returned.
`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-site combination.
`...`	Other arguments, currently unused.

Details

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).

The following statistics are calculated:

- **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.
- **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.
- **hours.gt.200** — number of hours NO2 is more than 200 ug/m3.
- **days.gt.50** — 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 95% confidence interval in the mean*

Description

Bin a variable and calculate mean an uncertainties in mean

Usage

```
binData(mydata, bin = "nox", uncer = "no2", n = 40, interval = NA, breaks = NA)
```

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`.
`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.

Details

This function summarises data by intervals and calculates the mean and bootstrap 95% confidence intervals in the mean of a chosen variable in a data frame. Any other numeric variables are summarised by their mean intervals.

There are three options for binning. The default is to bon `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.

Value

Retruns a summarised data frame with new columns for the mean and upper / lower 95% confidence intervals in the mean.

Examples

```
# how does nox vary by intervals of wind speed?
results <- binData(mydata, bin = "ws", uncer = "nox")

# easy to plot this using ggplot2
## Not run:
library(ggplot2)
ggplot(results, aes(ws, mean, ymin = min, ymax = max)) +
geom_pointrange()
```

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

---

bootMeanDF *Bootsrap confidence intervals in the mean*

Description

A utility function to calculation the uncertainty intervals in the mean of a vector. The function removes any missing data before the calculation.

Usage

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

Arguments

x	A vector from which the mean and bootstrap confidence intervals in the mean are to be calculated
conf.int	The confidence interval; default = 0.95.
B	The number of bootstrap simulations

Value

Returns a data frame with the mean, lower uncertainty, upper uncertainty and number of values used in the calculation

Examples

```
test <- rnorm(20, mean = 10)
bootMeanDF(test)
```

---

calcFno2 *Estimate NO2/NOX emission ratios from monitoring data*

Description

Given hourly NOX and NO2 from a roadside site and hourly NOX, NO2 and O3 from a background site the function will estimate the emissions ratio of NO2/NOX — the level of primary NO2

Usage

```
calcFno2(input, tau = 60, user.fno2, main = "", xlab = "year", ...)
```

Arguments

input	A data frame with the following fields. nox andno2 (roadside NOX and NO2 concentrations), back_nox, back_no2 and back_o3 (hourly background concentrations of each pollutant). In addition temp (temperature in degrees Celsius) and cl (cloud cover in Oktas). Note that if temp and cl are not available, typical means values of 11 deg. C and cloud = 3.5 will be used.
tau	Mixing time scale. It is unlikely the user will need to adjust this. See details below.
user.fno2	User-supplied f-NO2 fraction e.g. 0.1 is a NO2/NOX ratio of 10 series and is useful for testing "what if" questions.
main	Title of plot if required.
xlab	x-axis label.
...	Other graphical parameters send to scatterPlot.

Details

The principal purpose of this function is to estimate the level of primary (or direct) NO2 from road vehicles. When hourly data of NOX, NO2 and O3 are available, the total oxidant method of Clapp and Jenkin (2001) can be used. If roadside O3 measurements are available see linearRelation for details of how to estimate the primary NO2 fraction.

In the absence of roadside O3 measurements, it is rather more problematic to calculate the fraction of primary NO2. Carslaw and Beevers (2005c) developed an approach based on linearRelation the analysis of roadside and background measurements. The increment in roadside NO2 concentrations is primarily determined by direct emissions of NO2 and the availability of One to react with NO to form NO2. The method aims to quantify the amount of NO2 formed through these two processes by seeking the optimum level of primary NO2 that gives the least error.

Test data is provided at http://www.openair-project.org.

Value

As well as generating the plot itself, calcFno2 also returns an object of class “openair”. The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-calcFno2(...), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

Author(s)

David Carslaw

References

Clapp, L.J., Jenkin, M.E., 2001. Analysis of the relationship between ambient levels of O3, NO2 and NO as a function of NOX in the UK. Atmospheric Environment 35 (36), 6391-6405.

Carslaw, D.C. and N Carslaw (2007). Detecting and characterising small changes in urban nitrogen dioxide concentrations. Atmospheric Environment. Vol. 41, 4723-4733.

Carslaw, D.C., Beevers, S.D. and M.C. Bell (2007). Risks of exceeding the hourly EU limit value for nitrogen dioxide resulting from increased road transport emissions of primary nitrogen dioxide. Atmospheric Environment 41 2073-2082.

Carslaw, D.C. (2005a). Evidence of an increasing NO2/NOX emissions ratio from road traffic emissions. Atmospheric Environment, 39(26) 4793-4802.

Carslaw, D.C. and Beevers, S.D. (2005b). Development of an urban inventory for road transport emissions of NO2 and comparison with estimates derived from ambient measurements. Atmospheric Environment, (39): 2049-2059.

Carslaw, D.C. and Beevers, S.D. (2005c). Estimations of road vehicle primary NO2 exhaust emission fractions using monitoring data in London. Atmospheric Environment, 39(1): 167-177.

Carslaw, D. C. and S. D. Beevers (2004). Investigating the Potential Importance of Primary NO2 Emissions in a Street Canyon. Atmospheric Environment 38(22): 3585-3594.

Carslaw, D. C. and S. D. Beevers (2004). New Directions: Should road vehicle emissions legislation consider primary NO2? Atmospheric Environment 38(8): 1233-1234.

See Also

`linearRelation` if you have roadside ozone measurements.

Examples

```
## Users should see the full openair manual for examples of how
## to use this function.
```

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

Description

Calculates multiple percentile values from a time series, with flexible time aggregation.

Usage

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

Arguments

mydata	A data frame of data with a date field in the format Date or POSIXct. Must have one variable to apply calculations to.
pollutant	Name of variable to process. Mandatory.
avg.time	Averaging period to use. See timeAverage for details.
percentile	A vector of percentile values. For example percentile = 50 for median values, percentile = c(5,50,95 for multiple percentile values.
data.thresh	Data threshold to apply when aggregating data. See timeAverage for details.
start	Start date to use - see timeAverage for details.

Details

This is a utility function to calculate percentiles and is used in, for example, timePlot. Given a data frame with a date field and one other numeric variable, percentiles are calculated.

Value

Returns a data frame with new columns for each percentile level. New columns are given names like percentile.95 e.g. when percentile = 95 is chosen. See examples below.

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 convential 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 = 2003,
  month = 1:12,
  type = "default",
  annotate = "date",
  statistic = "mean",
  cols = "heat",
  limits = c(0, 100),
  lim = NULL,
  col.lim = c("grey30", "black"),
  col.arrow = "black",
  font.lim = c(1, 2),
  cex.lim = c(0.6, 1),
  digits = 0,
  data.thresh = 0,
  labels = NA,
  breaks = NA,
  w.shift = 0,
  remove.empty = TRUE,
  main = NULL,
  key.header = "",
  key.footer = "",
  key.position = "right",
  key = TRUE,
  auto.text = TRUE,
  ...
)
```

Arguments

`mydata`	A data frame minimally containing `date` and at least one other numeric variable. The date should be in either `Date` format or class `POSIXct`.
`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 all data potentially spanning several years will be plotted.

`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` Not yet implemented.

`annotate` This option controls what appears on each day of the calendar. Can be: "date" — shows day of the month; "wd" — shows vector-averaged wind direction, or "ws" — shows vector-averaged wind direction scaled by wind speed. Finally it can be "value" which shows the daily mean value.

`statistic` Statistic passed to `timeAverage`.

`cols` Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and `RColorBrewer` colours — see the `openair` `openColours` function for more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`

`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 maximimum 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.arrow` The colour of the annotated wind direction / wind speed arrows.

`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`.

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

`data.thresh` Data capture threshold passed to `timeAverage`. For example, `data.thresh = 75` means that at least 75% of the data must be available in a day for the value to be calculate, else the data is removed.

`labels` If a categorical scale is required then these labels will be used. Note there is one less label than break. For example, `labels = c("good","bad","very bad")`. `breaks` must also be supplied if labels are given.

`breaks` If a categorical scale is required then these breaks will be used. For example, `breaks = c(0,50,100,1000)`. In this case "good" corresponds to values

	berween 0 and 50 and so on. 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.
`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.
`remove.empty`	Should months with no data present be removed? Default is `TRUE`.
`main`	The plot title; default is pollutant and year.
`key.header`	Adds additional text/labels to the scale key. For example, passing `calendarPlot(mydata,key.header = "header",key.footer = "footer")` adds addition text above and below the scale key. These arguments are passed to `drawOpenKey` via `quickText`, applying the `auto.text` argument, to handle formatting.
`key.footer`	see `key.header`.
`key.position`	Location where the scale key is to plotted. Allowed arguments currently include `"top"`, `"right"`, `"bottom"` and `"left"`.
`key`	Fine control of the scale key via `drawOpenKey`. See `drawOpenKey` for further 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.
`...`	Other graphical parameters are passed onto the `lattice` function `lattice:levelplot`, with common axis and title labelling options (such as `xlab`, `ylab`, `main`) being passed to via `quickText` to handle routine formatting.

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

As well as generating the plot itself, `calendarPlot` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-calendarPlot(mydata,"nox")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw

See Also

`timePlot`, `timeVariation`

Examples

```
# load example data from package
data(mydata)

# 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")


## 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",
  xlab = "predicted value",
  ylab = "statistic",
  col = brewer.pal(5, "YlOrRd"),
  col.var = "Set1",
  var.names = NULL,
  auto.text = 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")`. Note that including other variables could reduce the number of data available to plot due to teh need of having non-missing data for all variables.
`var.mod`	Other variable predictions for which statistics should be calculated. Can be more than length one e.g. `var.obs = c("nox.obs","ws.obs")`.

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.
bins	Number of bins used in conditionalQuantile.
statistic	Statistic(s) to be plotted. Can be "MB", "NMB", "r", "COE", "MGE", "NMGE", "RMSE" and "FAC2", as described in modStats. When these statistics are chosen, they are calculated from var.mod and var.mod. statistic can also be a value that can be supplied to cutData. For example, statistic = "season" will show how model performance varies by season across the distribution of predictions which might highlight that at high concentrations of NOx the model tends to underestimate concentrations and that these periods mostly occur in winter. statistic can also be another variable in the data frame — see cutData for more information. A special case is statistic = "cluster" if clusters have been calculated using trajCluster.
xlab	label for the x-axis, by default "predicted value".
ylab	label for the y-axis, by default "observed value".
col	Colours to be used for plotting the uncertainty bands and median line. Must be of length 5 or more.
col.var	Colours for the additional variables to be compared. See openColours for more details.
var.names	Variable names to be shown on plot for plotting var.obs and var.mod.
auto.text	Either TRUE (default) or FALSE. If TRUE titles and axis labels etc. will automatically try and format pollutant names and units properly e.g. by subscripting the '2' in NO2.
...	Other graphical parameters passed onto conditionalQuantile and cutData. For example, conditionalQuantile passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to xyplot via quickText to handle routine formatting.

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% confidence 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. For example, an under-prediction in wind speed could result in higher surface NOx concentrations and lower ozone concentrations. Similarly if wind speed predictions were good and NOx was over predicted it might suggest an over-estimate of NOx emissions. One or more additional variables can be plotted.

A special case is statistic = "cluster". In this case a data frame is provided that contains the cluster calculated by trajCluster and importTraj. Alternatively users could supply their own pre-calculated clusters. These calculations can be very useful in showing whether certain back trajectory clusters are associated with poor (or good) model performance. Note that in the case of statistic = "cluster" there will be fewer data points used in the analysis compared with the ordinary statistics above because the trajectories are available for every three hours. Also 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.

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

See Wilks (2005) for more details of conditional quantile plots.

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

See `conditionalQuantile` for information on conditional quantiles, `modStats` for model evaluation statistics and the package `verification` for comprehensive functions for forecast verification.

Examples

```
## Examples to follow, or will be shown in the openair manual
```

`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),
  xlab = "predicted value",
  ylab = "observed value",
  col = brewer.pal(5, "YlOrRd"),
  key.columns = 2,
  key.position = "bottom",
  auto.text = 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.
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. Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second 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.
xlab	label for the x-axis, by default "predicted value".
ylab	label for the y-axis, by default "observed value".
col	Colours to be used for plotting the uncertainty bands and median line. Must be of length 5 or more.
key.columns	Number of columns to be used in the key.
key.position	Location of the key e.g. "top", "bottom", "right", "left". See lattice xyplot for more details.
auto.text	Either TRUE (default) or FALSE. If TRUE titles and axis labels etc. will automatically try and format pollutant names and units properly e.g. by subscripting the '2' in NO2.
...	Other graphical parameters passed onto cutData and lattice:xyplot. For example, conditionalQuantile passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to xyplot via quickText to handle routine formatting.

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 line).

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

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

See `modStats` for model evaluation statistics and the package `verification` for comprehensive functions for forecast verification.

Examples

```
# load example data from package
data(mydata)

## 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")
```

---

corPlot *corrgram plot with conditioning*

Description

Function to to draw and visualise correlation matrices using lattice. 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",
  dendrogram = FALSE,
  lower = FALSE,
  cols = "default",
  r.thresh = 0.8,
  text.col = c("black", "black"),
  auto.text = TRUE,
```

```
  ...
)
```

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	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.
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".
dendrogram	Should a dendrogram be plotted? When TRUE a dendrogram is shown on the right of the plot. Note that this will only work for type = "default".
lower	Should only the lower triangle be plotted?
cols	Colours to be used for plotting. Options include "default", "increment", "heat", "spectral", "hue", "greyscale" and user defined (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.
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.
...	Other graphical parameters passed onto lattice:levelplot, with common axis and title labelling options (such as xlab, ylab, main) being passed via quickText to handle routine formatting.

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

As well as generating the plot itself, `corPlot` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-corPlot(mydata)`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis. Note the denogram when `cluster = TRUE` can aslo be returned and plotted. See examples.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw — but mostly based on code contained in Sarkar (2007)

References

Sarkar, D. (2007). Lattice Multivariate Data Visualization with R. New York: Springer.

Friendly, M. (2002). Corrgrams : Exploratory displays for correlation matrices. American Statistician, 2002(4), 1-16. doi:10.1198/000313002533

See Also

`taylor.diagram` from the `plotrix` package from which some of the annotation code was used.

Examples

```
# load openair data if not loaded already
data(mydata)
## basic corrgram plot
corPlot(mydata)
## plot by season ... and so on
corPlot(mydata, type = "season")
```

```
## recover dendogram when cluster = TRUE and plot it
res <-corPlot(mydata)
plot(res$clust)
## Not run:
## 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. Users would generally not be expected to call this function directly. Widely used by many openair functions usually through the option type.

Usage

```
cutData(
  x,
  type = "default",
  hemisphere = "northern",
  n.levels = 4,
  start.day = 1,
  is.axis = FALSE,
  local.tz = NULL,
  latitude = 51,
  longitude = -0.5,
  ...
)
```

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.

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.
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. http://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. See import and the openair manual for information on how to import data and ensure no DST is applied.
latitude	The decimal latitude used in type = "daylight".
longitude	The decimal longitude. Note that locations west of Greenwich are negative.
...	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" 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 defintions 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.

"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 "dst" 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".

"ws" splits the data by 8 quantiles of wind speed and requires a column `ws`.

"site" splits the data by site and therefore requires a column `site`.

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 a data frame with a column `cond` that is defined by `type`.

Author(s)

David Carslaw (cutData) and Karl Ropkins (cutDaylight)

Examples

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

`drawOpenKey` *Scale key handling for openair*

Description

General function for producing scale keys for other openair functions. The function is a crude modification of the draw.colorkey function developed by Deepayan Sarkar as part of the lattice package, and allows additional key labelling to added, and provides some additional control of the appearance and scaling.

Usage

```
drawOpenKey(key, draw = FALSE, vp = NULL)
```

Arguments

key	List defining the scale key structure to be produced. Most options are identical to original `draw.colorkey` function. Original `draw.colorkey` options: `space` location of the scale key ("left", "right", "top" or "bottom"). Defaults to "right". `col` vector of colours, used in scale key. `at` numeric vector specifying where the colors change. Must be of length 1 more than the col vector. `labels` a character vector for labelling the at values, or more commonly, a list describing characteristics of the labels. This list may include components `labels`, `at`, `cex`, `col`, `rot`, `font`, `fontface` and `fontfamily`. `tick.number` approximate number of ticks. `width` width of the key. `height` height of key. Note: `width` and `height` refer to the key dimensions. `height` is the length of the key along the plot axis it is positioned against, and `width` is the length perpendicular to that. Additional options include: `header` a character vector of extra text to be added above the key, or a list describing some characteristics of the `header`. This list may include components `header`, the character vector of header labels, `tweaks`, a list of local controls, e.g. 'gap' and 'balance' for spacing relative to scale and footer, respectively, `auto.text`, `TRUE/FALSE` option to apply `quickText`, and `slot`, a numeric vector setting the size of the text boxes `header` text is placed in. `footer` as in `header` but for labels below the scale key. Notes: `header` and `footer` formatting can not be set locally, but instead are matched to those set in `labels`. `drawOpenKey` allows for up to six additional labels (three above and three below scale key). Any additional text is ignored. `tweak,auto.text,slot` as in `header` and `footer` but sets all options uniformly. This also overwrites anything in `header` and/or `footer`. `fit` the fit method to be applied to the header, scale key and footer when placing the scale key left or right of the plot. Options include: 'all', 'soft' and 'scale'. The default 'all' fits header, key and footer into `height` range. The alternative 'scale' fits only the key within `height`. (This means that keys keep the same proportions relative to the main plot regardless of positioning but that header and footer may exceed plot dimensions if `height` and/or `slots` are too large. `plot.style` a character vector of key plotting style instructions: Options currently include: 'paddle', 'ticks' and 'border'. 'paddle' applies the incremental paddle layout used by `winRose`. 'ticks' places ticks between the labels scale key. 'border' places a border about the scale key. Any combination of these may be used but if none set, scale key defaults to `c("ticks","border")` for most plotting operations or `c("paddle")` for `windRose`.

draw — Option to return the key object or plot it directly. The default, FALSE, should always be used within openair calls.

vp — View port to be used when plotting key. The default, NULL, should always be used within openair calls.

(Note: drawOpenKey is a crude modification of lattice::draw.colorkey, that provides labelling options for openair plot scale keys. Some aspects of the function are in development and may to subject to change. Therefore, it is recommended that you use parent openair function controls, e.g. key.position, key.header, key.footer options, where possible. drawOpenKey may obviously be used in other plots but it is recommended that draw.colorkey itself be used wherever this type of additional scale labelling is not required.)

Details

The drawOpenKey function produces scale keys for other openair functions.

Most drawOpenKey options are identical to those of lattice::draw.colorkey. For example, scale key size and position are controlled via height, width and space. Likewise, the axis labelling can be set in and formatted by labels. See draw.colorkey for further details.

Additional scale labelling may be added above and below the scale using header and footer options within key. As in other openair functions, automatic text formatting can be enabled via auto.key.

(Note: Currently, the formatting of header and footer text are fixed to the same style as labels (the scale axis) and cannot be defined locally.)

The relationship between header, footer and the scale key itself can be controlled using fit options. These can be set in key$fit to apply uniform control or individually in key$header$fit and/or key$footer$fit to control locally.

The appearance of the scale can be controlled using plot.style.

Value

The function is a modification of lattice::draw.colorkey and returns a scale key using a similar mechanism to that used in in the original function as developed by Deepayan Sarkar.

Note

We gratefully acknoweldge the considerable help and advice of Deepayan Sarkar.

Author(s)

draw.colorkey is part of the lattice package, developed by Deepayan Sarkar.

Additional modifications by Karl Ropkins.

References

Deepayan Sarkar (2010). lattice: Lattice Graphics. R package version 0.18-5. http://r-forge.r-project.org/projects/lattice/

See Also

Functions using drawOpenKey currently include windRose, pollutionRose.

For details of the original function, see draw.colorkey

Examples

```
##########
#example 1
##########

#paddle style scale key used by windRose

windRose(mydata,)

#adding text and changing style and position via key

#note:
#some simple key control also possible directly
#For example, below does same as
#windRose(mydata, key.position="right")

windRose(mydata,
   key =list(space="right")
)

#however:
#more detailed control possible working with
#key and drawOpenKey. For example,

windRose(mydata,
   key = list(header="Title", footer="wind speed",
              plot.style = c("ticks", "border"),
              fit = "all", height = 1,
              space = "top")
)
```

---

import *Generic data import for openair*

Description

This function is mostly used to simplify the importing of csv and text file in openair. In particular it helps to get the date or date/time into the correct format. The file can contain either a date or date/time in a single column or a date in one column and time in another.

Usage

```
import(
  file = file.choose(),
  file.type = "csv",
  sep = ",",
  header.at = 1,
  data.at = 2,
  date = "date",
  date.format = "%d/%m/%Y %H:%M",
  time = NULL,
  time.format = NULL,
  tzone = "GMT",
  na.strings = c("", "NA"),
  quote = "\"",
  ws = NULL,
  wd = NULL,
  correct.time = NULL,
  ...
)
```

Arguments

file	The name of the file to be imported. Default, file = file.choose(), opens browser. Alternatively, the use of read.table (in utils) also allows this to be a character vector of a file path, connection or url.
file.type	The file format, defaults to common 'csv' (comma delimited) format, but also allows 'txt' (tab delimited).
sep	Allows user to specify a delimiter if not ',' (csv) or TAB (txt). For example ';' is sometimes used to delineate separate columns.
header.at	The file row holding header information or NULL if no header to be used.
data.at	The file row to start reading data from. When generating the data frame, the function will ignore all information before this row, and attempt to include all data from this row onwards.
date	Name of the field containing the date. This can be a date e.g. 10/12/2012 or a date-time format e.g. 10/12/2012 01:00.
date.format	The format of the date. This is given in 'R' format according to strptime. For example, a date format such as 1/11/2000 12:00 (day/month/year hour:minutes) is given the format "%d/%m/%Y %H:%M". See examples below and strptime for more details.
time	The name of the column containing a time — if there is one. This is used when a time is given in a separate column and date contains no information about time.
time.format	If there is a column for time then the time format must be supplied. Common examples include "%H:%M" (like 07:00) or an integer giving the hour, in which case the format is "%H". Again, see examples below.

tzone	The time zone for the data. In order to avoid the complexities of DST (daylight savings time), openair assumes the data are in GMT (UTC) or a constant offset from GMT. Users can set a positive or negative offset in hours from GMT. For example, to set the time zone of the data to the time zone in New York (EST, 5 hours behind GMT) set tzone = "Etc/GMT+5". To set the time zone of the data to Central European Time (CET, 1 hour ahead of GMT) set tzone = "Etc/GMT-1". *Note that the positive and negative offsets are opposite to what most users expect.*
na.strings	Strings of any terms that are to be interpreted as missing (NA). For example, this might be "-999", or "n/a" and can be of several items.
quote	String of characters (or character equivalents) the imported file may use to represent a character field.
ws	Name of wind speed field if present if different from "ws" e.g. ws = "WSPD".
wd	Name of wind direction field if present if different from "wd" e.g. wd = "WDIR".
correct.time	Numerical correction (in seconds) for imported date. Default NULL turns this option off. This can be useful if the hour is represented as 1 to 24 (rather than 0 to 23 assumed by R). In which case correct.time = -3600 will correct the hour.
...	Other arguments passed to read.table.

Details

The function uses strptime to parse dates and times. Users should consider the examples for use of these formats.

The function can either deal with combined date-time formats e.g. 10/12/1999 23:00 or with two separate columns that deal with date and time. Often there is a column for the date and another for hour. For the latter, the option time.format = "%H" should be supplied. Note that R considers hours 0 to 23. However, if hours 1 to 24 are detected import will correct the hours accordingly.

import will also ensure wind speed and wind direction are correctly labelled (i.e. "ws", "wd") if ws or wd are given.

Note that it is assumed that the input data are in GMT (UTC) format and in particular there is no consideration of daylight saving time i.e. where in the input data set an hour is missing in spring and duplicated in autumn.

Examples of use are given in the openair manual.

Value

A data frame formatted for openair use.

Author(s)

David Carslaw

See Also

Dedicated import functions available for selected file types, e.g. : importAURN, importAURNCsv, importKCL, importADMS, etc.

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 `read.csv` (in `utils`) 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,
  ...
)
```

Arguments

`file`	The ADMS file to be imported. Default, `file.choose()` opens browser. Use of `read.csv` (in `utils`) 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 repaced 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`.

`...` Additional arguments, passed to `read.csv` as part of import operation.

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 simplfication 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

Generic import function import, for possible alternative import methods. Other dedicated import functions available for other file types, including importKCL, importAURN, etc.

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
```

importAQE *Air Quality England Network data import for openair*

Description

Functions for importing hourly mean air pollution data from a range of UK networks including the Automatic Urban and Rural Network. Files are imported from a remote server operated by Ricardo that provides air quality data files as R data objects.

Usage

```
importAQE(
  site = "yk13",
  year = 2018,
  pollutant = "all",
  meta = FALSE,
  ratified = FALSE,
  to_narrow = FALSE
)

importAURN(
  site = "my1",
  year = 2009,
  pollutant = "all",
  hc = FALSE,
  meta = FALSE,
  ratified = FALSE,
  to_narrow = FALSE,
  verbose = FALSE
)

importNI(
  site = "bel0",
  year = 2018,
  pollutant = "all",
  meta = FALSE,
  ratified = FALSE,
  to_narrow = FALSE
)

importSAQN(
  site = "gla4",
  year = 2009,
  pollutant = "all",
  meta = FALSE,
  ratified = FALSE,
  to_narrow = FALSE
```

```
)

importWAQN(
  site = "card",
  year = 2018,
  pollutant = "all",
  meta = FALSE,
  ratified = FALSE,
  to_narrow = FALSE
)
```

Arguments

site	Site code of the site to import e.g. "my1" is Marylebone Road. Several sites can be imported with site = c("my1","nott") — to import Marylebone Road and Nottingham for example.
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.
pollutant	Pollutants to import. If omitted will import all pollutants ffrom a site. To import only NOx and NO2 for example use pollutant = c("nox","no2").
meta	Should meta data be returned? If TRUE the site type, latitude and longitude are returned.
ratified	If TRUE columns are returned indicating when each species was ratified i.e. quality-checked.
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.
hc	A few sites have hydrocarbon measurements available and setting hc = TRUE will ensure hydrocarbon data are imported. The default is however not to as most users will not be interested in using hydrocarbon data and the resulting data frames are considerably larger. This option is only available for importAURN.
verbose	Should the function give messages when downloading files? Default is FALSE.

Details

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.

For an up to date list of available sites that can be imported, see importMeta.

The site codes and pollutant names can be upper or lower case.

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.

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

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.

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 few 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.

Value

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

Functions

- importAQE: Import data from the Air Quality England
- importNI: Import data from the Northern Ireland Air Quality Network
- importSAQN: Import data from the Scottish Air Quality Network
- importWAQN: Import data from the Welsh Air Quality Network

Author(s)

David Carslaw and Trevor Davies

See Also

importKCL, importADMS

Examples

```
## import all pollutants from Marylebone Rd from 1990:2009
## Not run: mary <- importAURN(site = "my1", year = 2000:2009)

## import nox, no2, o3 from Marylebone Road and Nottingham Centre for 2000
## Not run: thedata <- importAURN(site = c("my1", "nott"), year = 2000,
pollutant = c("nox", "no2", "o3"))
## End(Not run)

# Other functions work in the same way e.g. to import Cardiff Centre data:

## Not run: cardiff <- importWAQN(site = "card", year = 2020)
```

`importAURNCsv` *AURN csv file data import for openair*

Description

Function for importing common 1 hour average (hourly) UK Automatic Urban and Rural Network (AURN) Air Quality Archive data files previously downloaded in ".csv" format for use with the openair package. The function uses `read.table` (in `utils`) and `rbind` (in `reshape`).

Usage

```
importAURNCsv(
  file = file.choose(),
  header.at = 5,
  data.at = 7,
  na.strings = c("No data", "", "NA"),
  date.name = "Date",
  date.break = "-",
  time.name = "time",
  misc.info = c(1, 2, 3, 4),
  is.site = 4,
  bad.24 = TRUE,
  correct.time = -3600,
  output = "final",
  data.order = c("value", "status", "unit"),
  simplify.names = TRUE,
  ...
)
```

Arguments

`file`	The name of the AURN file to be imported. Default, `file.choose` opens browser. Use of `read.table` (in `utils`) also allows this to be a readable text-mode connection or url (although these options are currently not fully tested).

header.at	The file row holding header information. This is used to set names for the resulting imported data frame, but may be subject to further modifications depending on following argument settings.
data.at	The first file row holding actual data. When generating the data frame, the function will ignore all information before this row, and attempt to include all data from this row onwards.
na.strings	Strings of any terms that are to be interpreted as NA values within the file.
date.name	Header name of column holding date information. Combined with time information as single date column in the generated data frame.
date.break	The break character separating days, months and years in date information. For example, "-" in "01-01-2009".
time.name	Header name of column holding time information. Combined with date information as single date column in the generated data frame.
misc.info	Row numbers of any additional information that may be required from the original file. Each line retained as a character vector in the generated data frame comment.
is.site	Header name of column holding site information. Setting to NULL turns this option off.
bad.24	Reset AURN 24 time stamp. AURN time series are logged as 00:00:01 to 24:00:00 as opposed to the more conventional 00:00:00 to 23:59:59. bad.24 = TRUE resets the time stamp which is not allowed in some time series classes or functions.
correct.time	Numerical correction (in seconds) for imported date. AURN data is logged retrospectively. For 1 hour average data, correct.time = -3600 resets this to the start of the sampling period.
output	Output style. Default "final" using import().
data.order	A vector of names defining the order of data types. AURN files typically include three data types, actual data and associated data quality and measurement unit reports. Here, these are defined as "value", "status" and "unit", respectively.
simplify.names	A logical (default TRUE) prompting the function to try to simply data frame names using common chemical shorthand. FALSE retains names from original file, although these may be modified if they contain unallowed characters or non-unique names.
...	Other parameters. Passed onto and handled by import().

Details

The importAURN() function was developed for use with air quality monitoring site data files downloaded in standard hourly (or 1 hour average) format using the Air Quality Archive email service. Argument defaults are set to common values to simplify both the import operation and use with openair.

Similar file structures can also be imported using this function with argument modification.

Value

The function 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.POSIXct`). Time adjustments may also be made, subject to "bad.24" and "correct.time" argument settings. Using default settings, the argument `correct.time = -3600` resets the time stamp to the start of the measurement period. If name simplification was requested (`simplify.names = TRUE`), common chemical names will be simplified. For example, "carbon monoxide" will be reset to "co". Currently, this option only applies to inorganics and particulates, not organics. Non-value information will be rationalised according to data.order. For example, for the default, `data.order = c("value","status","unit")`, the status and unit columns following the "co" column will be automatically renamed "unit.co" and "status.co", respectively. An additional "site" column will be generated. Multiple "site" files are allowed. Additional information (as defined in "misc.info") and data adjustments (as defined by "bad.24" and "correct.time") are retained in the data frame comment.

Author(s)

Karl Ropkins

See Also

Generic import function `import`, or direct (on-line) data import function `importAURN`. Other dedicated import functions available for other file types, e.g.: `importKCL`, `importADMS`, etc.

Examples

```
##########
#example 1
##########
#data obtained from email service:
#http://www.airquality.co.uk/archive/data_selector.php
#or
#http://www.airquality.co.uk/archive/data_and_statistics.php?action=step_pre_1
#example file "AirQualityDataHourly.csv" Brighton Roadside and Brighton Preston Park 2008.

#import data as mydata
## mydata <- importAURN.csv("AirQualityDataHourly.csv")

#read additional information retained by importAURN
## comment(mydata)

#analysis data by site
## boxplot(nox ~ site, data = mydata)

##########
#example 2
##########
#example using data from url
```

```
#import data as otherdata
## otherdata <- importAURN.csv(
##  "http://www.airquality.co.uk/archive/data_files/site_data/HG1_2007.csv")

#use openair function
## summarise(otherdata)

##########
#example 3
##########
#example of importing other similar data formats

#import 15 min average so2 data from Bexley using url
## so2.15min.data <- importAURN.csv(
##  "http://www.airquality.co.uk/archive/data_files/15min_site_data/BEX_2008.csv",
##  correct.time = -900)

#note: correct.time amended for 15 min offset/correction.

#additional comments
## comment(so2.15min.data)

#analysis
## diurnal.error(so2.15min.data, pollutant="so2")

#wrapper for above operation
##(e.g. if you have to do this -or similar- a lot of time)
## my.import.wrapper <- function(file, correct.time = -900, ...)
##  { importAURN.csv(file = file, correct.time = correct.time, ...) }

#same as above
## so2.15min.data.again <- my.import.wrapper(
##  "http://www.airquality.co.uk/archive/data_files/15min_site_data/BEX_2008.csv")

#analysis
## timeVariation(so2.15min.data.again, pollutant="so2")
```

`importEurope` *Import air quality data from European database*

Description

This function is a simplified version of the `saqgetr` package (see https://github.com/skgrange/saqgetr) for accessing European air quality data. The function only returns valid hourly data and is meant as a fast and convenient way of accessing the most common type of hourly air quality data. The function works in the same way as other `openair` functions that import air quality data that generally need a site code and year to be supplied.

Usage

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

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.

Details

The function can however return key site meta data.

The saqgetr package is much more comprehensive and provides data at other time averages e.g. daily data.

Value

A tibble of data.

Examples

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

importKCL *Import data from King's College London networks*

Description

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

Usage

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

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 or years to import. To import a sequence of years from 1990 to 2000 use year = 1990:2000. To import several specfic years use year = c(1990,1995,2000) for example.
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").
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. Access to reliable and free meteorological data is problematic.
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.
extra	Not currently 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.

Details

The `importKCL` function has been written to make it easy to import data from the King's College London air pollution networks. KCL have provided .RData files (R workspaces) of all individual sites and years for the KCL 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 `importKCL` 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 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 `importKCL`), 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.

While the function is being developed, the following site codes should help with selection. We will also make available other meta data such as site type and location to make it easier to select sites based on other information. Note that these codes need to be refined because only the common species are available for export currently i.e. NOx, NO2, O3, CO, SO2, PM10, PM2.5.

- A30 | Kingston - Kingston Bypass A3 | Roadside
- AD1 | Shoreham-by-Sea | Kerbside
- AR1 | Chichester - Lodsworth | Rural
- AR2 | Wealden - Isfield | Rural
- AS1 | Bath Aethalometer | Urban Background
- BA1 | Basildon - Gloucester Park | Roadside
- BB1 | Broxbourne (Roadside) | Roadside
- BE0 | Belfast - Carbon | Urban Background
- BE1 | Belfast Centre AURN | Urban Background
- BE3 | Belfast Centre Aethalometer | Urban Background
- BE7 | Belfast Centre FDMS trial | Urban Background
- BE8 | Belfast - Nitrate | Urban Background
- BE9 | Belfast - Partisol SO4 | Urban Background
- BF1 | Bedford Stewartby (Rural) | Industrial
- BF3 | Bedford - Kempston | Industrial
- BF4 | Bedford - Prebend Street | Roadside
- BF5 | Bedford - Lurke Street | Roadside
- BG1 | Barking and Dagenham - Rush Green | Suburban
- BG2 | Barking and Dagenham - Scrattons Farm | Suburban
- BG3 | Barking and Dagenham - North Street | Kerbside
- BH0 | Brighton Preston Park AURN | Urban Background
- BH1 | Brighton Roadside | Roadside
- BH2 | Brighton and Hove - Hove Town Hall | Roadside
- BH3 | Brighton and Hove - Foredown Tower | Urban Background

- BH5 | Brighton Mobile (Preston Fire Station) | Roadside
- BH6 | Brighton Mobile (Lewes Road) | Roadside
- BH7 | Brighton Mobile (Gloucester Road) | Roadside
- BH8 | Brighton and Hove - Stanmer Park | Rural
- BH9 | Brighton Mobile Beaconsfield Road | Roadside
- BI1 | Birmingham Tyburn CPC | Urban Background
- BL0 | Camden - Bloomsbury | Urban Background
- BL1 | Bloomsbury AURN SMPS | Urban Background
- BM1 | Ballymena - Ballykeel | Suburban
- BM2 | Ballymena - North Road | Roadside
- BN1 | Barnet - Tally Ho Corner | Kerbside
- BN2 | Barnet - Finchley | Urban Background
- BN3 | Barnet - Strawberry Vale | Urban Background
- BO1 | Ballymoney 1 | Suburban
- BP0 | Westminster - Bridge Place | Urban Background
- BQ5 | Bexley - Manor Road West Gravimetric | Industrial
- BQ6 | Bexley - Manor Road East Gravimetric | Industrial
- BQ7 | Belvedere West | Urban Background
- BQ8 | Belvedere West FDMS | Urban Background
- BT1 | Brent - Kingsbury | Suburban
- BT2 | Brent - Ikea Car Park | Roadside
- BT3 | Brent - Harlesden | Roadside
- BT4 | Brent - Ikea | Roadside
- BT5 | Brent - Neasden Lane | Industrial
- BT6 | Brent - John Keble Primary School | Roadside
- BT7 | Brent - St Marys Primary School | Urban Background
- BW1 | Brentwood - Brentwood Town Hall | Urban Background
- BX0 | Bexley - Belvedere FDMS | Suburban
- BX1 | Bexley - Slade Green | Suburban
- BX2 | Bexley - Belvedere | Suburban
- BX3 | Bexley - Thamesmead | Suburban
- BX4 | Bexley - Erith | Industrial
- BX5 | Bexley - Bedonwell | Suburban
- BX6 | Bexley - Thames Road North FDMS | Roadside
- BX7 | Bexley - Thames Road North | Roadside
- BX8 | Bexley - Thames Road South | Roadside
- BX9 | Bexley - Slade Green FDMS | Suburban

- BY1 | Bromley - Rent Office | Urban Background
- BY4 | Bromley - Tweedy Rd | Roadside
- BY5 | Bromley - Biggin Hill | Suburban
- BY7 | Bromley - Harwood Avenue | Roadside
- CA1 | Crawley Background | Urban Background
- CA2 | Crawley - Gatwick Airport | Urban Background
- CB1 | Chelmsford - Fire Station | Roadside
- CB2 | Chelmsford - Springfield Road | Roadside
- CB3 | Chelmsford - Chignal St James | Urban Background
- CB4 | Chelmsford - Baddow Road | Roadside
- CC1 | Colchester - Lucy Lane South | Roadside
- CC2 | Colchester - Brook Street | Roadside
- CC3 | Colchester - Mersea Road | Roadside
- CD1 | Camden - Swiss Cottage | Kerbside
- CD3 | Camden - Shaftesbury Avenue | Roadside
- CD4 | Camden - St Martins College (NOX 1) | Urban Background
- CD5 | Camden - St Martins College (NOX 2) | Urban Background
- CD7 | Camden - Swiss Cottage Partisol | Kerbside
- CD9 | Camden - Euston Road | Roadside
- CF1 | Cardiff Aethalometer | Urban Background
- CH1 | Cheltenham | Urban Background
- CI1 | Chichester - A27 Chichester Bypass | Roadside
- CI4 | Chichester - Orchard Street | Roadside
- CK1 | Cookstown | Suburban
- CP1 | Castle Point - Canvey Island | Urban Background
- CR2 | Croydon - Purley Way | Roadside
- CR3 | Croydon - Thornton Heath | Suburban
- CR4 | Croydon - George Street | Roadside
- CR5 | Croydon - Norbury | Kerbside
- CR6 | Croydon - Euston Road | Suburban
- CT1 | City of London - Senator House | Urban Background
- CT2 | City of London - Farringdon Street | Kerbside
- CT3 | City of London - Sir John Cass School | Urban Background
- CT4 | City of London - Beech Street | Roadside
- CT6 | City of London - Walbrook Wharf | Roadside
- CT8 | City of London - Upper Thames Street | Roadside
- CY1 | Crystal Palace - Crystal Palace Parade | Roadside

- DC1 | Dacorum 1 Hemel Hempstead (Background) | Urban Background
- DC2 | Dacorum 2 Hemel Hempstead (Background) | Urban Background
- DC3 | High Street Northchurch | Roadside
- DE1 | Derry City - Brandywell | Urban Background
- DE2 | Derry City - Dales Corner | Roadside
- DM1 | Dunmurry Aethalometer | Urban Background
- EA0 | Ealing - Acton Town Hall FDMS | Roadside
- EA1 | Ealing - Ealing Town Hall | Urban Background
- EA2 | Ealing - Acton Town Hall | Roadside
- EA3 | Ealing 3 - A40 East Acton | Roadside
- EA4 | Ealing Mobile - Hamilton Road | Roadside
- EA5 | Ealing Mobile - Southall | Roadside
- EA6 | Ealing - Hanger Lane Gyratory | Roadside
- EA7 | Ealing - Southall | Urban Background
- EA8 | Ealing - Horn Lane | Industrial
- EA9 | Ealing - Court Way | Roadside
- EB1 | Eastbourne - Devonshire Park | Urban Background
- EB3 | Eastbourne - Holly Place | Urban Background
- EH1 | E Herts Throcking (Rural) | Rural
- EH2 | East Herts Sawbridgeworth (Background) | Urban Background
- EH3 | East Herts Sawbridgeworth (Roadside) | Roadside
- EH4 | East Herts Ware | Roadside
- EH5 | East Herts Bishops Stortford | Roadside
- EI0 | Ealing - Greenford | Urban Background
- EI1 | Ealing - Western Avenue | Roadside
- EL1 | Elmbridge - Bell Farm Hersham | Urban Background
- EL2 | Elmbridge - Esher High Street | Roadside
- EL3 | Elmbridge - Hampton Court Parade | Roadside
- EL4 | Elmbridge - Walton High Street | Kerbside
- EN1 | Enfield - Bushhill Park | Suburban
- EN2 | Enfield - Church Street | Roadside
- EN3 | Enfield - Salisbury School | Urban Background
- EN4 | Enfield - Derby Road | Roadside
- EN5 | Enfield - Bowes Primary School | Roadside
- FB1 | Rushmoor - Medway Drive | Roadside
- GB0 | Greenwich and Bexley - Falconwood FDMS | Roadside
- GB6 | Greenwich and Bexley - Falconwood | Roadside

- GL1 | Glasgow Centre | Suburban
- GL4 | Glasgow Centre Aethalometer | Suburban
- GN0 | Greenwich - A206 Burrage Grove | Roadside
- GN2 | Greenwich - Millennium Village | Industrial
- GN3 | Greenwich - Plumstead High Street | Roadside
- GN4 | Greenwich - Fiveways Sidcup Rd A20 | Roadside
- GR4 | Greenwich - Eltham | Suburban
- GR5 | Greenwich - Trafalgar Road | Roadside
- GR7 | Greenwich - Blackheath | Roadside
- GR8 | Greenwich - Woolwich Flyover | Roadside
- GR9 | Greenwich - Westhorne Avenue | Roadside
- HA0 | Harwell - Carbon | Rural
- HA1 | Harwell Rural AURN | Rural
- HA2 | Harwell Rural PARTISOL | Rural
- HA4 | Harwell Rural SMPS | Rural
- HA9 | Harwell - Partisol SO4 | Urban Background
- HF1 | Hammersmith and Fulham - Broadway | Roadside
- HF2 | Hammersmith and Fulham - Brook Green | Urban Background
- HF3 | Hammersmith and Fulham - Scrubs Lane | Kerbside
- HG1 | Haringey - Haringey Town Hall | Roadside
- HG2 | Haringey - Priory Park | Urban Background
- HG3 | Haringey - Bounds Green | Roadside
- HI0 | Hillingdon - Sipson Road | Suburban
- HI1 | Hillingdon - South Ruislip | Roadside
- HI2 | Hillingdon - Hillingdon Hospital | Roadside
- HI3 | Hillingdon - Oxford Avenue | Roadside
- HK4 | Hackney - Clapton | Urban Background
- HK6 | Hackney - Old Street | Roadside
- HL1 | Halifax Aethalometer | Urban Background
- HM1 | Hertsmere Borehamwood 1 (Background) | Urban Background
- HM4 | Hertsmere - Borehamwood | Urban Background
- HO1 | Horsham Background | Urban Background
- HO2 | Horsham - Park Way | Roadside
- HO4 | Horsham - Storrington | Roadside
- HO5 | Horsham - Cowfold | Roadside
- HR1 | Harrow - Stanmore | Urban Background
- HR2 | Harrow - Pinner Road | Roadside

- HS1 | Hounslow - Brentford | Roadside
- HS2 | Hounslow - Cranford | Suburban
- HS3 | Hounslow - Brentford | Roadside
- HS4 | Hounslow - Chiswick High Road | Roadside
- HS5 | Hounslow - Brentford | Roadside
- HS6 | Hounslow - Heston Road | Roadside
- HS7 | Hounslow - Hatton Cross | Urban Background
- HS9 | Hounslow - Feltham | Roadside
- HT1 | Hastings - Bulverhythe | Roadside
- HT2 | Hastings - Fresh Fields | Roadside
- HV1 | Havering - Rainham | Roadside
- HV2 | Havering - Harold Hill | Suburban
- HV3 | Havering - Romford | Roadside
- HX0 | Birmingham Tyburn Aethalometer | Urban Background
- IC6 | City of London - Walbrook Wharf Indoor | Roadside
- IG4 | Greenwich - Eltham Ecology Centre Indoor | Urban Background
- IS1 | Islington - Upper Street | Urban Background
- IS2 | Islington - Holloway Road | Roadside
- IS4 | Islington - Foxham Gardens | Urban Background
- IS5 | Islington - Duncan Terrace | Roadside
- IS6 | Islington - Arsenal | Urban Background
- IT2 | Tower Hamlets - Mile End Road | Roadside
- KB1 | South Kirkby Aethalometer | Urban Background
- KC0 | North Kensington - Carbon | Urban Background
- KC1 | Kensington and Chelsea - North Ken | Urban Background
- KC2 | Kensington and Chelsea - Cromwell Road | Roadside
- KC3 | Kensington and Chelsea - Knightsbridge | Roadside
- KC4 | Kensington and Chelsea - Kings Road | Roadside
- KC5 | Kensington and Chelsea - Earls Court Rd | Kerbside
- KC7 | Kensington and Chelsea - North Ken FDMS | Urban Background
- KC9 | North Kensington - Partisol SO4 | Urban Background
- KT1 | Kingston - Chessington | Suburban
- KT2 | Kingston - Town Centre | Roadside
- LA1 | Luton Airport | Urban Background
- LB1 | Lambeth - Christchurch Road | Roadside
- LB2 | Lambeth - Vauxhall Cross | Roadside
- LB3 | Lambeth - Loughborough Junct | Urban Background

- LB4 | Lambeth - Brixton Road | Kerbside
- LB5 | Lambeth - Bondway Interchange | Roadside
- LB6 | Lambeth - Streatham Green | Urban Background
- LH0 | Hillingdon - Harlington | Urban Background
- LH2 | Heathrow Airport | Urban Background
- LL1 | Lullington Heath Rural AURN | Rural
- LN1 | Luton - Challney Community College | Urban Background
- LS1 | Lewes - Telscombe Cliffs | Roadside
- LS2 | Lewes - Commercial Square | Roadside
- LS4 | Newhaven - Denton School | Urban Background
- LW1 | Lewisham - Catford | Urban Background
- LW2 | Lewisham - New Cross | Roadside
- LW3 | Lewisham - Mercury Way | Industrial
- MA1 | Manchester Piccadilly CPC | Urban Background
- MA2 | Manchester Piccadilly | Urban Background
- MD1 | Mid Beds Biggleswade (Roadside) | Roadside
- MD2 | Mid Beds Silsoe (Rural) | Rural
- MD3 | Central Beds - Sandy | Roadside
- MD4 | Central Beds - Marston Vale | Rural
- ME1 | Merton - Morden Civic Centre | Roadside
- MP1 | Marchwood Power - Marchwood | Industrial
- MP2 | Marchwood Power - Millbrook Rd Soton | Industrial
- MR3 | Marylebone Road Aethalometer | Kerbside
- MV1 | Mole Valley - Leatherhead | Rural
- MV2 | Mole Valley - Lower Ashtead | Suburban
- MV3 | Mole Valley - Dorking | Urban Background
- MW1 | Windsor and Maidenhead - Frascati Way | Roadside
- MW2 | Windsor and Maidenhead - Clarence Road | Roadside
- MW3 | Windsor and Maidenhead - Ascot | Rural
- MY0 | Marylebone Road - Carbon | Kerbside
- MY1 | Westminster - Marylebone Road | Kerbside
- MY7 | Westminster - Marylebone Road FDMS | Kerbside
- NA5 | Newtownabbey- Mallusk | Urban Background
- NA6 | Newtownabbey- Shore Road | Roadside
- NE2 | Port Talbot TEOM and CPC | Urban Background
- NF1 | New Forest - Holbury | Industrial
- NF2 | New Forest - Fawley | Industrial

- NF3 | New Forest - Ringwood | Urban Background
- NF4 | New Forest - Totton | Roadside
- NF5 | New Forest - Lyndhurst | Roadside
- NH1 | North Herts Mobile - Baldock 1 | Roadside
- NH2 | North Herts Mobile - Baldock 2 | Roadside
- NH3 | North Herts Mobile - Royston | Urban Background
- NH4 | North Herts - Breechwood Green | Urban Background
- NH5 | North Herts - Baldock Roadside | Roadside
- NH6 | North Herts - Hitchin Library | Roadside
- NK1 | North Kensington - CPC | Urban Background
- NK3 | North Kensington Aethalometer | Urban Background
- NK6 | North Kensington - URG | Urban Background
- NM1 | Newham - Tant Avenue | Urban Background
- NM2 | Newham - Cam Road | Roadside
- NM3 | Newham - Wren Close | Urban Background
- NW1 | Norwich Centre Aethalometer | Urban Background
- OX0 | Oxford Centre Roadside AURN | Urban Background
- OX1 | South Oxfordshire - Henley | Roadside
- OX2 | South Oxfordshire - Wallingford | Roadside
- OX3 | South Oxfordshire - Watlington | Roadside
- OX4 | Oxford St Ebbes AURN | Urban Background
- PO1 | Portsmouth Background AURN | Urban Background
- PT6 | Port Talbot Dyffryn School | Industrial
- RB1 | Redbridge - Perth Terrace | Urban Background
- RB2 | Redbridge - Ilford Broadway | Kerbside
- RB3 | Redbridge - Fullwell Cross | Kerbside
- RB4 | Redbridge - Gardner Close | Roadside
- RB5 | Redbridge - South Woodford | Roadside
- RD0 | Reading AURN - New Town | Urban Background
- RD1 | Reading - Caversham Road | Roadside
- RD2 | Reading - Kings Road | Roadside
- RD3 | Reading - Oxford Road | Roadside
- RG1 | Reigate and Banstead - Horley | Suburban
- RG2 | Reigate and Banstead - Horley South | Suburban
- RG3 | Reigate and Banstead - Poles Lane | Rural
- RG4 | Reigate and Banstead - Reigate High St | Kerbside
- RHA | Richmond - Lower Mortlake Road | Roadside

- RHB | Richmond - Lower Mortlake Road | Roadside
- RI1 | Richmond - Castelnau | Roadside
- RI2 | Richmond - Barnes Wetlands | Suburban
- RI5 | Richmond Mobile - St Margarets | Kerbside
- RI6 | Richmond Mobile - St Margarets | Kerbside
- RI7 | Richmond Mobile - Richmond Park | Suburban
- RI8 | Richmond Mobile - Richmond Park | Suburban
- RIA | Richmond Mobile - George Street | Kerbside
- RIB | Richmond Mobile - George Street | Kerbside
- RIC | Richmond Mobile - Kew Rd | Kerbside
- RID | Richmond Mobile - Kew Rd | Kerbside
- RIE | Richmond Mobile - Richmond Rd Twickenham | Roadside
- RIF | Richmond Mobile - Richmond Rd Twickenham | Roadside
- RIG | Richmond Mobile - Upper Teddington Rd | Roadside
- RIH | Richmond Mobile - Upper Teddington Rd | Roadside
- RII | Richmond Mobile - Somerset Rd Teddington | Urban Background
- RIJ | Richmond Mobile - Somerset Rd Teddington | Urban Background
- RIK | Richmond Mobile - St. Margarets Grove | Urban Background
- RIL | Richmond Mobile - St. Margarets Grove | Urban Background
- RIM | Richmond Mobile - Petersham Rd Ham | Roadside
- RIN | Richmond Mobile - Petersham Rd Ham | Roadside
- RIO | Richmond Mobile - Stanley Rd Twickenham | Roadside
- RIP | Richmond Mobile - Stanley Rd Twickenham | Roadside
- RIQ | Richmond Mobile - Richmond Rd Twickenham | Roadside
- RIR | Richmond Mobile - Richmond Rd Twickenham | Roadside
- RIS | Richmond Mobile - Lincoln Ave Twickenham | Roadside
- RIU | Richmond Mobile - Mortlake Rd Kew | Roadside
- RIW | Richmond - Upper Teddington Road | Roadside
- RIY | Richmond - Hampton Court Road | Kerbside
- RO1 | Rochford - Rayleigh High Street | Roadside
- RY1 | Rother - Rye Harbour | Rural
- RY2 | Rother - De La Warr Road | Roadside
- SA1 | St Albans - Fleetville | Urban Background
- SB1 | South Beds - Dunstable | Urban Background
- SC1 | Sevenoaks 1 | Suburban
- SD1 | Southend-on-Sea AURN | Urban Background
- SE1 | Stevenage - Lytton Way | Roadside

- SH1 | Southampton Background AURN | Urban Background
- SH2 | Southampton - Redbridge | Roadside
- SH3 | Southampton - Onslow Road | Roadside
- SH4 | Southampton - Bitterne | Urban Background
- SK1 | Southwark - Larcom Street | Urban Background
- SK2 | Southwark - Old Kent Road | Roadside
- SK5 | Southwark - A2 Old Kent Road | Roadside
- SL1 | Sunderland Aethalometer | Urban Background
- ST1 | Sutton - Robin Hood School | Roadside
- ST2 | Sutton - North Cheam | Urban Background
- ST3 | Sutton - Carshalton | Suburban
- ST4 | Sutton - Wallington | Kerbside
- ST5 | Sutton - Beddington Lane | Industrial
- ST6 | Sutton - Worcester Park | Kerbside
- ST7 | Sutton - Therapia Lane | Industrial
- SU1 | Sussex Mobile10 Stockbridge | Kerbside
- SU2 | Sussex Mobile11 Jct Whitley Rd | Kerbside
- SU3 | Sussex Mobile12 Cowfold | Kerbside
- SU4 | Sussex Mobile 13 Newhaven | Roadside
- SU5 | Sussex Mobile 14 Crawley | Roadside
- SU6 | Sussex Mobile15 Chichester County Hall | Urban Background
- SU7 | Sussex Mobile 16 Warnham | Rural
- SU8 | Sussex Mobile 17 Newhaven Paradise Park | Roadside
- SX1 | Sussex Mobile 1 | Urban Background
- SX2 | Sussex Mobile 2 North Berstead | Roadside
- SX3 | Sussex Mobile 3 | Roadside
- SX4 | Sussex Mobile 4 Adur | Roadside
- SX5 | Sussex Mobile 5 Fresh Fields Rd Hastings | Roadside
- SX6 | Sussex Mobile 6 Orchard St Chichester | Roadside
- SX7 | Sussex Mobile 7 New Road Newhaven | Roadside
- SX8 | Sussex Mobile 8 Arundel | Kerbside
- SX9 | Sussex Mobile 9 Newhaven Kerbside | Kerbside
- TD0 | Richmond - National Physical Laboratory | Suburban
- TE0 | Tendring St Osyth AURN | Rural
- TE1 | Tendring - Town Hall | Roadside
- TH1 | Tower Hamlets - Poplar | Urban Background
- TH2 | Tower Hamlets - Mile End Road | Roadside

- TH3 | Tower Hamlets - Bethnal Green | Urban Background
- TH4 | Tower Hamlets - Blackwall | Roadside
- TK1 | Thurrock - London Road (Grays) | Urban Background
- TK2 | Thurrock - Purfleet | Roadside
- TK3 | Thurrock - Stanford-le-Hope | Roadside
- TK8 | Thurrock - London Road (Purfleet) | Roadside
- TR1 | Three Rivers - Rickmansworth | Urban Background
- UT1 | Uttlesford - Saffron Walden Fire Station | Roadside
- UT2 | Uttlesford - Takeley | Urban Background
- UT3 | Uttlesford - Broxted Farm | Rural
- VS1 | Westminster - Victoria Street | Kerbside
- WA1 | Wandsworth - Garratt Lane | Roadside
- WA2 | Wandsworth - Town Hall | Urban Background
- WA3 | Wandsworth - Roehampton | Rural
- WA4 | Wandsworth - High Street | Roadside
- WA6 | Wandsworth - Tooting | Roadside
- WA7 | Wandsworth - Putney High Street | Kerbside
- WA8 | Wandsworth - Putney High Street Facade | Roadside
- WA9 | Wandsworth - Putney | Urban Background
- WE0 | Kensington and Chelsea - Pembroke Road | Urban Background
- WF1 | Watford (Roadside) | Roadside
- WF2 | Watford - Watford Town Hall | Roadside
- WH1 | Welwyn Hatfield - Council Offices | Urban Background
- WL1 | Waltham Forest - Dawlish Road | Urban Background
- WL2 | Waltham Forest - Mobile | Roadside
- WL3 | Waltham Forest - Chingford | Roadside
- WL4 | Waltham Forest - Crooked Billet | Kerbside
- WL5 | Waltham Forest - Leyton | Roadside
- WM0 | Westminster - Horseferry Road | Urban Background
- WM3 | Westminster - Hyde Park Partisol | Roadside
- WM4 | Westminster - Charing Cross Library | Roadside
- WM5 | Westminster - Covent Garden | Urban Background
- WM6 | Westminster - Oxford St | Kerbside
- WR1 | Bradford Town Hall Aethalometer | Urban Background
- WT1 | Worthing - Grove Lodge | Kerbside
- XB1 | Bletchley | Rural
- XS1 | Shukri Outdoor | Industrial

- XS2 | Shukri Indoor | Industrial
- XS3 | Osiris mobile | Urban Background
- YH1 | Harrogate Roadside | Roadside
- ZA1 | Ashford Rural - Pluckley | Rural
- ZA2 | Ashford Roadside | Roadside
- ZA3 | Ashford Background | Urban Background
- ZA4 | Ashford M20 Background | Urban Background
- ZC1 | Chatham Roadside - A2 | Roadside
- ZD1 | Dover Roadside - Town Hall | Roadside
- ZD2 | Dover Roadside - Townwall Street | Roadside
- ZD3 | Dover Background - Langdon Cliff | Urban Background
- ZD4 | Dover Background - East Cliff | Urban Background
- ZD5 | Dover Coast Guard Met | Urban Background
- ZD6 | Dover Docks | Industrial
- ZF1 | Folkestone Suburban - Cheriton | Suburban
- ZG1 | Gravesham Backgrnd - Northfleet | Urban Background
- ZG2 | Gravesham Roadside - A2 | Roadside
- ZG3 | Gravesham Ind Bgd - Northfleet | Urban Background
- ZH1 | Thanet Rural - Minster | Rural
- ZH2 | Thanet Background - Margate | Urban Background
- ZH3 | Thanet Airport - Manston | Urban Background
- ZH4 | Thanet Roadside - Ramsgate | Roadside
- ZL1 | Luton Background | Urban Background
- ZM1 | Maidstone Meteorological | Urban Background
- ZM2 | Maidstone Roadside - Fairmeadow | Kerbside
- ZM3 | Maidstone Rural - Detling | Rural
- ZR1 | Dartford Roadside - St Clements | Kerbside
- ZR2 | Dartford Roadside 2 - Town Centre | Roadside
- ZR3 | Dartford Roadside 3 - Bean Interchange | Roadside
- ZS1 | Stoke Rural AURN | Rural
- ZT1 | Tonbridge Roadside - Town Centre | Roadside
- ZT2 | Tunbridge Wells Background - Town Hall | Urban Background
- ZT3 | Tunbridge Wells Rural - Southborough | Rural
- ZT4 | Tunbridge Wells Roadside - St Johns | Roadside
- ZT5 | Tonbridge Roadside 2 - High St | Roadside
- ZV1 | Sevenoaks - Greatness Park | Urban Background
- ZV2 | Sevenoaks - Bat and Ball | Roadside

- ZW1 | Swale Roadside - Ospringe A2 | Roadside
- ZW2 | Swale Background - Sheerness | Urban Background
- ZW3 | Swale Roadside 2 - Ospringe Street | Roadside
- ZY1 | Canterbury Backgrnd - Chaucer TS | Urban Background
- ZY2 | Canterbury Roadside - St Dunstans | Roadside
- ZY4 | Canterbury St Peters Place | Roadside

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

importAURN, importADMS, importSAQN

Examples

```
## import all pollutants from Marylebone Rd from 1990:2009
## Not run: mary <- importKCL(site = "my1", year = 2000:2009)

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

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

---

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

Description

Function to import meta data for air quality monitoring sites

Usage

```
importMeta(source = "aurn", all = FALSE)
```

Arguments

source	The data source for the meta data. Can be "aurn", "saqn" (or "saqd"), "aqe", "ni", "kcl" or "europe"; upper or lower case.
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.

Details

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

- "aurn", The UK Automatic Urban and Rural Network.
- "saqn", The Scottish Air Quality Network.
- "waqn", The Welsh Air Quality Network.
- "ni", The Northern Ireland Air Quality Network.
- "aqe", The Air Quality England Network.
- "kcl", King's College London networks.
- "europe", Import hourly European data (Airbase/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. For most networks, this detailed information includes per-pollutant summaries, opening and closing dates of sites etc.

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

importAURN, importKCL and importSAQN for importing air quality data from each network.

Examples

```
## basic data

## Not run:
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)

## 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)
```

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 specfic 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/'`.

Details

This function imports pre-calculated back trajectories using the HYSPLIT trajectory model (Hybrid Single Particle Lagrangian Integrated Trajectory Model http://ready.arl.noaa.gov/HYSPLIT.php). 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 (see http://ready.arl.noaa.gov/HYSPLIT_agreement.php). 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

`trajPlot`, `importAURN`, `importKCL`,`importADMS`, `importSAQN`

Examples

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

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

kernelExceed *Kernel density plot for daily mean exceedance statistics*

Description

This function is used to explore the conditions leading to exeedances of air quality limits. Currently the focus is on understanding the conditions under which daily limit values for PM10 are in excess of a specified threshold. Kernel density estimates are calculated and plotted to highlight those conditions.

Usage

```
kernelExceed(
  polar,
  x = "wd",
  y = "ws",
  pollutant = "pm10",
  type = "default",
  by = c("day", "dayhour", "all"),
  limit = 50,
  data.thresh = 0,
  more.than = TRUE,
  cols = "default",
  nbin = 256,
  auto.text = TRUE,
  ...
)
```

Arguments

polar	A data frame minimally containing date and at least three other numeric variables, typically ws, wd and a pollutant.
x	x-axis variable. Mandatory.
y	y-axis variable. Mandatory
pollutant	Mandatory. A pollutant name corresponding to a variable in a data frame should be supplied e.g. pollutant = "nox"
type	The type of analysis to be done. The default is will produce a single plot using the entire data. Other types include "hour" (for hour of the day), "weekday" (for day of the week) and "month" (for month of the year), "year" for a polarPlot for each year. It is also possible to choose type as another variable in the data frame. For example, type = "o3" will plot four kernel exceedance plots for different levels of ozone, split into four quantiles (approximately equal numbers of counts in each of the four splits). This offers great flexibility for understanding the variation of different variables dependent on another. See function cutData for further details.

by	by determines how data above the limit are selected. by = "day" will select *all* data (typically hours) on days where the daily mean value is above limit. by = "dayhour" will select only those data above limit on days where the daily mean value is above limit. by = "hour" will select all data above limit.
limit	The threshold above which the pollutant concentration will be considered.
data.thresh	The data capture threshold to use (the data using timeAverage to daily means. 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.
more.than	If TRUE data will be selected that are greater than limit. If FALSE data will be selected that less than limit.
cols	Colours to be used for plotting. Options include "default", "increment", "heat", "spectral", "hue", "brewer1" and user defined (see manual for more details). The same line colour can be set for all pollutant e.g. cols = "black".
nbin	number of bins to be used for the kernel density estimate.
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.
...	Other graphical parameters passed onto lattice:levelplot and cutData. For example, kernelExceed passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to levelplot via quickText to handle routine formatting.

Details

The kernelExceed functions is for exploring the conditions under which exceedances of air pollution limits occur. Currently it is focused on the daily mean (European) Limit Value for PM10 of 50~ug/m3 not to be exceeded on more than 35 days. However, the function is sufficiently flexible to consider other limits e.g. could be used to explore days where the daily mean are greater than 100~ug/m3.

By default the function will plot the kernel density estimate of wind speed and wind directions for all days where the concentration of pollutant is greater than limit. Understanding the conditions where exceedances occur can help with source identification.

The function offers different ways of selecting the data on days where the pollutant are greater than limit through setting by. By default it will select all data on days where pollutant is greater than limit. With the default setting of by it will select all data on those days where pollutant is greater than limit, even if individual data (e.g. hours) are less than limit. Setting by = "dayhour" will additionally ensure that all data on the those dates are also greater than limit. Finally, by = "all" will select all values of pollutant above limit, regardless of when they occur.

The usefulness of the function is greatly enhanced through using type, which conditions the data according to the level of another variable. For example, type = "season" will show the kernel density estimate by spring, summer, autumn and winter and type = "so2" will attempt to show the

kernel density estimates by quantiles of SO2 concentration. By considering different values of `type` it is possible to develop a good understanding of the conditions under which exceedances occur.

To aid interpretation the plot will also show the *estimated* number of days or hours where exeedances occur. For `type = "default"` the number of days should exactly correspond to the actual number of exceedance days. However, with different values of `type` the number of days is an estimate. It is an estimate because conditioning breaks up individual days and the estimate is based on the proportion of data calculated for each level of `type`.

Value

To be completed.

Note

This function automatically chooses the bandwidth for the kernel density estimate. We generally find that most data sets are not overly sensitive to the choice of bandwidth. One important reason for this insensitivity is likley to be the characteristics of air pollution itself. Due to atmospheric dispersion processes, pollutant plumes generally mix rapidly in the atmosphere. This means that pollutant concentrations are 'smeared-out' and extra fidelity brought about by narrower bandwidths do not recover any more detail.

Author(s)

David Carslaw

See Also

`polarAnnulus`, `polarFreq`, `polarPlot`

Examples

```
# Note! the manual contains other examples that are more illuminating
# basic plot
kernelExceed(mydata, pollutant = "pm10")

# condition by NOx concentrations
## Not run: kernelExceed(mydata, pollutant = "pm10", type = "nox")
```

`linearRelation` *Linear relations between pollutants*

Description

This function considers linear relationships between two pollutants. The relationships are calculated on different times bases using a linear model. The slope and 95 in slope relationships by time unit are plotted in many ways. The function is particularly useful when considering whether relationships are consistent with emissions inventories.

Usage

```
linearRelation(
  mydata,
  x = "nox",
  y = "no2",
  period = "month",
  condition = FALSE,
  n = 20,
  rsq.thresh = 0,
  ylab = paste0("slope from ", y, " = m.", x, " + c"),
  auto.text = TRUE,
  cols = "grey30",
  date.breaks = 5,
  ...
)
```

Arguments

`mydata`	A data frame minimally containing `date` and two pollutants.
`x`	First pollutant that when plotted would appear on the x-axis of a relationship e.g. `x = "nox"`.
`y`	Second pollutant that when plotted would appear on the y-axis of a relationship e.g. `y = "pm10"`.
`period`	A range of different time periods can be analysed. The default is `month` but can be `year` and `week`. For increased flexibility an integer can be used e.g. for 3-month values `period = "3 month"`. Other cases include `"hour"` will show the diurnal relationship between `x` and `y` and "weekday" the day of the week relationship between `x` and `y`. "day.hour" will plot the relationship by weekday and hour of the day.
`condition`	For `period = "hour"`, `period = "day"` and `period = "day.hour"`, setting `condition = TRUE` will plot the relationships split by year. This is useful for seeing how the relationships may be changing over time.
`n`	The minimum number of points to be sent to the linear model. Because there may only be a few points e.g. hours where two pollutants are available over one

week, n can be set to ensure that at least n points are sent to the linear model. If a period has hours < n that period will be ignored.

`rsq.thresh` The minimum correlation coefficient (R2) allowed. If the relationship between x and y is not very good for a particular period, setting `rsq.thresh` can help to remove those periods where the relationship is not strong. Any R2 values below `rsq.thresh` will not be plotted.

`ylab` y-axis title, specified by the user.

`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.

`cols` Colour for the points and uncertainty intervals.

`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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of `date.breaks` up or down.

`...` Other graphical parameters. A useful one to remove the strip with the date range on at the top of the plot is to set `strip = FALSE`.

Details

The relationships between pollutants can yield some very useful information about source emissions and how they change. A scatterPlot between two pollutants is the usual way to investigate the relationship. A linear regression is useful to test the strength of the relationship. However, considerably more information can be gleaned by considering different time periods, such as how the relationship between two pollutants vary over time, by day of the week, diurnally and so on. The `linearRelation` function does just that - it fits a linear relationship between two pollutants over a wide range of time periods determined by `period`.

`linearRelation` function is particularly useful if background concentrations are first removed from roadside concentrations, as the increment will relate more directly with changes in emissions. In this respect, using `linearRelation` can provide valuable information on how emissions may have changed over time, by hour of the day etc. Using the function in this way will require users to do some basic manipulation with their data first.

If a data frame is supplied that contains `nox`, `no2` and `o3`, the `y` can be chosen as `y = "ox"`. In function will therefore consider total oxidant slope (sum of NO2 + O3), which can provide valuable information on likely vehicle primary NO emissions. Note, however, that most roadside sites do not have ozone measurements and `calcFno2` is the alternative.

Value

As well as generating the plot itself, `linearRelation` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-linearRelation(mydata,"nox","no2")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw

See Also

calcFno2

Examples

```
# monthly relationship between NOx and SO2 - note rapid fall in
# ratio at the beginning of the series
linearRelation(mydata, x = "nox", y = "so2")
# monthly relationship between NOx and SO2 - note rapid fall in
# ratio at the beginning of the series
## Not run: linearRelation(mydata, x = "nox", y = "ox")

# diurnal oxidant slope by year # clear change in magnitude
# starting 2003, but the diurnal profile has also changed: the
# morning and evening peak hours are more important, presumably
# due to change in certain vehicle types
## Not run: linearRelation(mydata, x = "nox", y = "ox", period = "hour", condition = TRUE)

# PM2.5/PM10 ratio, but only plot where monthly R2 >= 0.8
## Not run: linearRelation(mydata, x = "pm10", y = "pm25", rsq.thresh = 0.8)
```

---

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 respresents modelled values.
obs	Name of a variable in mydata that respresents 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.
...	Other aruments to be passed to cutData e.g. hemisphere = "southern"

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 aurument method e.g. method = "spearman"
- 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")`.

Details of the formulas are given in the openair manual.

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.

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 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. The data set is a tibble.

Format

Data frame with 65533 observations (rows) on the following 10 variables:

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

list("ws") Wind speed, in m/s, as numeric vector.

list("wd") Wind direction, in degrees from North, as a numeric vector.

list("nox") Oxides of nitrogen concentration, in ppb, as a numeric vector.

list("no2") Nitrogen dioxide concentration, in ppb, as a numeric vector.

list("o3") Ozone concentration, in ppb, as a numeric vector.

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

list("so2") Sulfur dioxide concentration, in ppb, as a numeric vector.

list("co") Carbon monoxide concentration, in ppm, as a numeric vector.

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

Details

`mydata` is supplied with the `openair` package as an example dataset for use with documented examples.

Note

`openair` functions generally require data frames with a field "date" that can be in either `POSIXct` or `Date` format but should be GMT time zone. This can be hourly data or higher resolution data.

Source

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

The same data is also provide in `'.csv'` format via the openair project web site http://www.openair-project.org.

Examples

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

---

openair *Tools for the analysis of air pollution data*

Description

This is a UK Natural Environment Research Council (NERC) funded knowledge exchange project that aims to make available innovative analysis tools for air pollution data; with additional support from Defra. The tools have generally been developed to analyse data of hourly resolution (or at least a regular time series) both for air pollution monitoring and dispersion modelling. The availability of meteorological data at the same time resolution greatly enhances the capabilities of these tools.

Details

The project code is also developed using R-Forge (http://r-forge.r-project.org/projects/openair/).

openair contains collection of functions to analyse air pollution data. Typically it is expected that data are hourly means, although most functions consider other time periods. The principal aim to make available analysis techniques that most users of air quality data and model output would not normally have access to. The functions consist of those developed by the authors and a growing number from other researchers.

The package also provides access to a wide range of data sources including the UK Automatic Urban and Rural Network (AURN), networks run by King's College London (e.g. the LAQN) and the Scottish Air Quality Network (SAQN).

The package has a number of requirements for input data and these are discussed in the manual (available on the openair website at http://www.openair-project.org). The key requirements are that a date or date-time field must have the name 'date' (and can be Date or POSIXct format), that wind speed is represented as 'ws' and that wind direction is 'wd'.

Most functions work in a very straightforward way, but offer many options for finer control and perhaps more in-depth analysis.

The openair package depends on several other packages written by other people to function properly.

To ensure that these other packages are available, they need to be installed, and this requires a connection to the internet. Other packages required come with the R base system. If there are problems with the automatic download of these packages, see http://www.openair-project.org for more details.

NOTE: openair assumes that data are not expressed in local time where 'Daylight Saving Time' is used. All functions check that this is the case and issue a warning if TRUE. It is recommended

that data are expressed in UTC/GMT (or a fixed offset from) to avoid potential problems with R and openair functions. The openair manual provides advice on these issues (available on the website).

To check to see if openair has been correctly installed, try some of the examples below.

Author(s)

David Carslaw with initial support from Karl Ropkins

References

Most reference details are given under the specific functions. The principal reference is below but users may also wish to cite the manual (details for doing this are contained in the manual itself).

Carslaw, D.C. and K. Ropkins, (2012) openair — an R package for air quality data analysis. Environmental Modelling & Software. Volume 27-28, 52-61.

See Also

See http://www.openair-project.org for up to date information on the project.

Examples

```
# load example data from package
data(mydata)

# summarise the data in a compact way
## Not run: summaryPlot(mydata)

# traditional wind rose
windRose(mydata)

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

---

openColours *openair colours*

Description

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

Usage

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

Arguments

scheme	The pre-defined schemes are "increment", "default", "brewer1", "heat", "jet", "hue", "greyscale", or a vector of R colour names e.g. c("green", "blue"). It is also possible to supply colour schemes from the RColorBrewer package. This package defines three types of colour schemes: sequential, diverging or qualitative. See http://colorbrewer2.org for more details concerning the orginal work on which this is based. Simplified versions of the viridis colours are also available. These include "viridis", "plasma", "magma", "inferno" and "cividis". Sequential colours are useful for ordered data where there is a need to show a difference between low and high values with colours going from light to dark. The pre-defined colours that can be supplied are: "Blues", "BuGn", "BuPu", "GnBu", "Greens", "Greys", "Oranges", "OrRd", "PuBu", "PuBuGn", "PuRd", "Purples", "RdPu", "Reds", "YlGn", "YlGnBu", "YlOrBr", "YlOrRd". Diverging palettes put equal emphasis on mid-range critical values and extremes at both ends of the data range. Pre-defined values are: "BrBG", "PiYG", "PRGn", "PuOr", "RdBu", "RdGy", "RdYlBu", "RdYlGn", "Spectral". Qualitative palettes are useful for differentiating between categorical data types. The pre-defined schemes are "Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3". A colorblind safe pallette "cbPalette" is available based on the work of: http://jfly.iam.u-tokyo.ac.jp/color/ Note that because of the way these 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.
n	number of colours required.

Details

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.

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" 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.

Value

Returns colour values - see examples below.

Author(s)

David Carslaw

References

http://colorbrewer2.org

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 mutiple 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,
```

```
  auto.text = TRUE,
  key.header = NULL,
  key.footer = "percentile",
  key.position = "bottom",
  key = TRUE,
  ...
)
```

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 the wind direction field.
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. Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second 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 be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for more details. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue")
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	The pollutant scale is by default shown at a 45 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set angle.scale to another value (between 0 and 360 degrees) to mitigate such problems. For example angle.scale = 315 will draw the scale heading in a NW direction.
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.
key.header	Adds additional text/labels to the scale key. For example, passing options key.header = "header",key.footer = "footer" adds addition text above and below the scale key. These arguments are passed to drawOpenKey via quickText, applying the auto.text argument, to handle formatting.
key.footer	key.header.
key.position	Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".
key	Fine control of the scale key via drawOpenKey. See drawOpenKey for further details.
...	Other graphical parameters are passed onto cutData and lattice:xyplot. For example, percentileRose passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common graphical arguments, such as xlim and ylim for plotting ranges and lwd for line thickness when using fill = FALSE, are passed on xyplot, although some local modifications may be applied by openair. For example, axis and title labelling options (such as xlab, ylab and main) are passed to xyplot via quickText to handle routine formatting.

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

As well as generating the plot itself, percentileRose also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-percentileRose(mydata,"nox"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

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.

See Also

See Also as windRose, pollutionRose, polarFreq, polarPlot

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")

# percentile concentrations by season and day/nighttime..
percentileRose(mydata, type = c("season", "daylight"), 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 = c(0, 100),
  cols = "default",
  width = "normal",
  min.bin = 1,
  exclude.missing = TRUE,
  date.pad = FALSE,
  force.positive = TRUE,
  k = c(20, 10),
  normalise = FALSE,
  key.header = "",
  key.footer = pollutant,
  key.position = "right",
  key = TRUE,
  auto.text = TRUE,
  ...
)
```

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 — 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.

Type can be up length two e.g. type = c("season","site") will produce a 2x2 plot split by season and site. The use of two types is mostly meant for situations where there are several sites. Note, when two types are provided the first forms the columns and the second the rows.

Also note that for the polarAnnulus function some type/period combinations are forbidden or make little sense. For example, type = "season" and period = "trend" (which would result in a plot with too many gaps in it for sensible smoothing), or type = "weekday" and period = "weekday".

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` Limits for colour scale.

`cols` Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and user defined. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`

`width` The width of the annulus; can be "normal" (the default), "thin" or "fat".

`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 postive. This is useful for several reasons. First, with lots of missing data more interpolation is needed and this can result in artifacts because the predictions are too far from the original data. Second, if it is known beforehand that the data are all postive, 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.

key.header	Adds additional text/labels to the scale key. For example, passing the options key.header = "header",key.footer = "footer1" adds addition text above and below the scale key. These arguments are passed to drawOpenKey via quickText, applying the auto.text argument, to handle formatting.
key.footer	see key.header.
key.position	Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".
key	Fine control of the scale key via drawOpenKey. See drawOpenKey for further 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.
...	Other graphical parameters passed onto lattice:levelplot and cutData. For example, polarAnnulus passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to levelplot via quickText to handle routine formatting.

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

As well as generating the plot itself, polarAnnulus also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-polarAnnulus(mydata,"nox"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

Author(s)

David Carslaw

See Also

polarPlot, polarFreq, pollutionRose and percentileRose

Examples

```
# load example data from package
data(mydata)

# 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")

# 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,
  cols = "Paired",
  angle.scale = 315,
```

```
  units = x,
  auto.text = TRUE,
  ...
)
```

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 lattice panel plot will be output showing the clusters identified for each one of n.clusters.
cols	Colours to be used for plotting. Useful options for categorical data are available from RColorBrewer colours — see the openair openColours function for more details. Useful schemes include "Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3" — but see ?brewer.pal for the maximum useful colours in each. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue").
angle.scale	The wind speed scale is by default shown at a 315 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set angle.scale to another value (between 0 and 360 degrees) to mitigate such problems. For example angle.scale = 45 will draw the scale heading in a NE direction.
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.
...	Other graphical parameters passed onto polarPlot, lattice:levelplot and cutData. Common axis and title labelling options (such as xlab, ylab, main) are passed via quickText to handle routine formatting.

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.

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

As well as generating the plot itself, polarCluster also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the original data frame with a new field cluster identifying the cluster; 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.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

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

polarPlot

Examples

```
## Not run:
# load example data from package
data(mydata)

## 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)
```

---

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 = "",
  statistic = "frequency",
  ws.int = 1,
```

```
  wd.nint = 36,
  grid.line = 5,
  breaks = seq(0, 5000, 500),
  cols = "default",
  trans = TRUE,
  type = "default",
  min.bin = 1,
  ws.upper = NA,
  offset = 10,
  border.col = "transparent",
  key.header = statistic,
  key.footer = pollutant,
  key.position = "right",
  key = TRUE,
  auto.text = TRUE,
  ...
)
```

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 "frequency", "mean", "median", "max" (maximum), "stdev" (standard deviation) or "weighted.mean". The option "frequency" (the default) is the simplest and plots the frequency of wind speed/direction in different bins. The scale therefore shows the counts in each bin. The option "mean" will plot the mean concentration of a pollutant (see next point) in wind speed/direction bins, and so on. Finally, "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. More information is given in the examples. Note that for options other than "frequency", it is necessary to also provide the name of a pollutant. See function cutData for further details.
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	The user can provide their own scale. breaks expects a sequence of numbers that define the range of the scale. The sequence could represent one with equal spacing e.g. breaks = seq(0,100,10) - a scale from 0-10 in intervals of 10, or a more flexible sequence e.g. breaks = c(0,1,5,7,10), which may be useful for some situations.
cols	Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for

more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`

`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` `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.

Type can be up length two e.g. `type = c("season","weekday")` will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second 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 `polarPlot` 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 as a percentage of the maximum wind speed. Setting a higher `offset` e.g. 50 is useful for `statistic = "weighted.mean"` when `ws.int` is greater than the maximum wind speed. See example below.

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

`key.header, key.footer`
Adds additional text/labels to the scale key. For example, passing options `key.header = "header",key.footer = "footer"` adds addition text above and below the scale key. These arguments are passed to `drawOpenKey` via `quickText`, applying the `auto.text` argument, to handle formatting.

`key.position` Location where the scale key is to plotted. Allowed arguments currently include `"top"`, `"right"`, `"bottom"` and `"left"`.

`key` Fine control of the scale key via `drawOpenKey`. See `drawOpenKey` for further 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.

`...` Other graphical parameters passed onto `lattice:xyplot` and `cutData`. For example, `polarFreq` passes the option `hemisphere = "southern"` on to `cutData` to provide southern (rather than default northern) hemisphere handling of `type = "season"`. Similarly, common axis and title labelling options (such as `xlab`, `ylab`, `main`) are passed to `xyplot` via `quickText` to handle routine formatting.

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

As well as generating the plot itself, `polarFreq` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-polarFreq(mydata,"nox")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw

References

~put references to the literature/web site here ~

See Also

See Also as `windRose`, `polarPlot`

Examples

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

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


# 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)

# 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 = "jet")
## 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))

# 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))

# 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",
  resolution = "fine",
  limits = NA,
  exclude.missing = TRUE,
  uncertainty = FALSE,
  percentile = NA,
  cols = "default",
  weights = c(0.25, 0.5, 0.75),
  min.bin = 1,
  mis.col = "grey",
  alpha = 1,
  upper = NA,
  angle.scale = 315,
  units = x,
  force.positive = TRUE,
  k = 100,
  normalise = FALSE,
  key.header = "",
  key.footer = pollutant,
  key.position = "right",
  key = TRUE,
  auto.text = TRUE,
  ws_spread = 0.5,
  wd_spread = 4,
  kernel = "gaussian",
  tau = 0.5,
  ...
)
```

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 — 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.

Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second 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 statisitics and `"quantile.slope"`, which uses quantile regression to estimate the slope for a particular quantile level (see also `tau` for setting the quantile level).

`resolution` Two plot resolutions can be set: "normal" and "fine" (the default), for a smoother plot. It should be noted that plots with a "fine" resolution can take longer to render.

`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 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 be used for plotting. Options include "default", "increment", "heat", "jet" and `RColorBrewer` colours — see the `openair` `openColours` function for more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`. `cols` can also take the values `"viridis"`, `"magma"`, `"inferno"`, or `"plasma"` which are the viridis colour maps ported from Python's Matplotlib library.

`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"`.

`alpha` The alpha transparency to use for the plotting surface (a value between 0 and 1 with zero being fully transparent and 1 fully opaque). Setting a value below 1 can be useful when plotting surfaces on a map using the package `openairmapss`.

`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` The wind speed scale is by default shown at a 315 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set `angle.scale` to another value (between 0 and 360 degrees) to mitigate such problems. For example `angle.scale = 45` will draw the scale heading in a NE direction.

`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 artifacts 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.header` Adds additional text/labels to the scale key. For example, passing the options `key.header = "header",key.footer = "footer1"` adds addition text above and below the scale key. These arguments are passed to `drawOpenKey` via `quickText`, applying the `auto.text` argument, to handle formatting.

`key.footer` see `key.footer`.

`key.position` Location where the scale key is to plotted. Allowed arguments currently include `"top"`, `"right"`, `"bottom"` and `"left"`.

`key` Fine control of the scale key via `drawOpenKey`. See `drawOpenKey` for further 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.

`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`.

`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.

`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.

`...` Other graphical parameters passed onto `lattice:levelplot` and `cutData`. For example, `polarPlot` passes the option `hemisphere = "southern"` on to `cutData` to provide southern (rather than default northern) hemisphere handling of `type`

= "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to levelplot via quickText to handle routine formatting.

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 in the package openair. 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 which take account of the frequency of measurements.

Variants on polarPlot include polarAnnulus and polarFreq.

Value

As well as generating the plot itself, polarPlot also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame

of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-polarPlot(mydata,"nox")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

`polarPlot` surface data can also be extracted directly using the `results`, e.g. `results(object)` for `output <-polarPlot(mydata,"nox")`. This returns a data frame with 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. http:// dx.doi.org/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. http://www.sciencedirect.com/science/article/pii/S1352231016307166

See Also

The openair package for many more functions for analysing air pollution data.

Examples

```
# Use openair 'mydata'

# basic plot
polarPlot(openair::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)
```

---

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/unenable quickText using this option.

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.

Examples

```
#example 1
##see axis formatting in an openair plot, e.g.:
scatterPlot(mydata, x = "no2", y = "pm10")

#example 2
##using quickText in other plots
plot(mydata$no2, mydata$pm10, xlab = quickText("my no2 label"),
     ylab = quickText("pm10 [ ug.m-3 ]"))
```

---

rollingMean *Calculate rollingMean values*

Description

Calculate rollingMean values taking account of data capture thresholds

Usage

```
rollingMean(
  mydata,
  pollutant = "o3",
  width = 8,
  new.name = "rolling",
  data.thresh = 75,
  align = "centre",
  ...
)
```

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.
new.name	The name given to the new rollingMean variable. If not supplied it will create a name based on the name of the pollutant and the averaging period used.
data.thresh	The data capture threshold in 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	specifyies how the moving window should be aligned. "right" means that the previous hours (including the current) are averaged. This seems to be the default for UK air quality rolling mean statistics. "left" means that the forward hours are averaged, and "centre" or "center", which is the default.
...	other arguments, currently unused.

Details

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 e.g. 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.

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")
```

---

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 = TRUE,
  key.title = group,
  key.columns = 1,
  key.position = "right",
  strip = TRUE,
  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,
  map = FALSE,
  auto.text = TRUE,
  ...
)
```

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, a timeAverage of 2 months would be period = "2 month". See function timeAverage for further details on this. This option se useful as one method by which the number of points plotted is reduced i.e. by choosing a longer averaging time.
data.thresh	The data capture threshold to use (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. Not used if avg.time = "default".
statistic	The statistic to apply when aggregating the data; default is the mean. Can be one of "mean", "max", "min", "median", "frequency", "sd", "percentile". Note that "sd" is the standard deviation and "frequency" is the number (frequency) of valid records in the period. "percentile" is the percentile level (using the "percentile" option - see below. Not used if avg.time = "default".
percentile	The percentile level in % used when statistic = "percentile" and when aggregating the data with avg.time. The default is 95. Not used if avg.time = "default".

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.

Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.

smooth — A smooth line is fitted to the data if TRUE; optionally with 95% 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% 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). mod.line is appropriately transformed when x or y axes are on a log scale.

cols — Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for more details. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue")

plot.type — lattice plot type. Can be "p" (points — default), "l" (lines) or "b" (lines and points).

key — Should a key be drawn? The default is TRUE.

key.title — The title of the key (if used).

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

key.position — Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".

strip — Should a strip be drawn? The default is TRUE.

log.x — Should the x-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.

log.y — 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 checking linearity once logged.

x.inc — The x-interval to be used for binning data when method = "level".

y.inc — The 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 — This option allows a scatter plot to show the wind speed/direction shows as an arrow. The option is a list e.g. windflow = list(col = "grey",lwd = 2,scale = 0.1). This option requires wind speed (ws) and wind direction (wd) to be available.

The maximum length of the arrow plotted is a fraction of the plot dimension with the longest arrow being scale of the plot x-y dimension. Note, if the plot size is adjusted manually by the user it should be re-plotted to ensure the correct wind angle. The list may contain other options to panel.arrows in the lattice package. Other useful options include length, which controls the length of the arrow head and angle, which controls the angle of the arrow head.

This option works best where there are not too many data to ensure over-plotting does not become a problem.

y.relation — This determines how the y-axis scale is 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.

x.relation — This determines how the x-axis scale is 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.

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")). See panel.abline in the lattice package for more details on adding/controlling lines.

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.

map — Should a base map be drawn? This option is under development.

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.

... — Other graphical parameters are passed onto cutData and an appropriate lattice plot function (xyplot, levelplot or hexbinplot depending on method). For example, scatterPlot passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, for the default case method = "scatter" common axis and title labelling options (such as xlab, ylab, main) are passed to xyplot via quickText to handle routine formatting. Other common graphical parameters, e.g. layout for panel arrangement, pch for plot symbol and lwd and lty for line width and type, as also available (see examples below).

For method = "hexbin" it can be useful to transform the scale if it is dominated by a few very high values. This is possible by supplying two functions: one that that applies the transformation and the other that inverses it. For log scaling (the default) for example, trans = function(x) log(x) and inv = function(x) exp(x). For a square root transform use trans = sqrt and inv = function(x) x^2. To not carry out any transformation the options trans = NULL and inv = NULL should be used.

Details

The 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.

By default plots are shown with a colour key at the bottom and in the case of conditioning, strips on the top of each plot. Sometimes this may be overkill and the user can opt to remove the key and/or the strip by setting key and/or strip to FALSE. One reason to do this is to maximise the plotting area and therefore the information shown.

Value

As well as generating the plot itself, `scatterPlot` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-scatterPlot(mydata,"nox","no2")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw

See Also

`linearRelation`, `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

library(hexbin)
# basic use, single pollutant
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 - clear effect of
## increased O3

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 make it easier to select periods from a data frame before sending to a function

Usage

```
selectByDate(
  mydata,
```

```
  start = "1/1/2008",
  end = "31/12/2008",
  year = 2008,
  month = 1,
  day = "weekday",
  hour = 1
)
```

Arguments

mydata	A data frame containing a date field in hourly or high resolution format.
start	A start date string in the form d/m/yyyy e.g. "1/2/1999" or in 'R' format i.e. "YYYY-mm-dd", "1999-02-01"
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.

Details

This function makes it much easier to select periods of interest from a data frame based on dates in a British format. Selecting date/times in R format can be intimidating for new users. This function can be used to select quite complex dates simply - see examples below.

Dates are assumed to be inclusive, so start = "1/1/1999" means that times are selected from hour zero. Similarly, end = "31/12/1999" will include all hours of the 31st December. start and end can also be in standard R format as a string i.e. "YYYY-mm-dd", so start = "1999-01-01" is fine.

All options are applied in turn making it possible to select quite complex dates

Author(s)

David Carslaw

Examples

```
## select all of 1999
data.1999 <- selectByDate(mydata, start = "1/1/1999", end = "31/12/1999")
head(data.1999)
```

```
tail(data.1999)

# or...
data.1999 <- selectByDate(mydata, start = "1999-01-01", end = "1999-12-31")

# 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

Utility function to extract user-defined run lengths (durations) above a threshold

Usage

```
selectRunning(mydata, pollutant = "nox", run.len = 5, threshold = 500)
```

Arguments

`mydata`	A data frame with a `date` field and at least one numeric `pollutant` field to analyse.
`pollutant`	Name of variable to process. Mandatory.
`run.len`	Run length for extracting contiguous values of `pollutant` above the `threshold` value.
`threshold`	The threshold value for `pollutant` above which data should be extracted.

Details

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 500ppb for contiguous periods of 5 or more hours.

This function is useful, for example, for selecting pollution episodes from a data frame i.e. where concentrations remain elevated for a certain period of time. It may also be of more general use when analysing air pollution data. For example, `selectRunning` could be used to extract continuous periods of rainfall — which could be important for particle concentrations.

Value

Returns a data frame that meets the chosen criteria. See examples below.

Author(s)

David Carslaw

Examples

```
## extract those hours where there are at least 5 consecutive NOx
## concentrations above 500ppb

mydata <- selectRunning(mydata, run.len = 5, threshold = 500)

## make a polar plot of those conditions...shows that those
## conditions are dominated by low wind speeds, not
## in-canyon recirculation
## Not run: polarPlot(mydata, pollutant = "nox")
```

---

smoothTrend *Calculate nonparametric smooth trends*

Description

Use non-parametric methods to calculate time series trends

Usage

```
smoothTrend(
  mydata,
  pollutant = "nox",
  deseason = FALSE,
  type = "default",
  statistic = "mean",
  avg.time = "month",
  percentile = NA,
  data.thresh = 0,
  simulate = FALSE,
  n = 200,
  autocor = FALSE,
  cols = "brewer1",
  shade = "grey95",
  xlab = "year",
  y.relation = "same",
  ref.x = NULL,
  ref.y = NULL,
```

```
  key.columns = length(percentile),
  name.pol = pollutant,
  ci = TRUE,
  alpha = 0.2,
  date.breaks = 7,
  auto.text = TRUE,
  k = NULL,
  ...
)
```

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	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. Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.
statistic	Statistic used for calculating monthly values. Default is "mean", but can also be "percentile". See timeAverage for more details.
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 plit up into spring: March, April, May etc. Note that December is considered as belonging to winter of the following year.
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.
data.thresh	The data capture threshold to use (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. Not used if avg.time = "default".

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.
cols	Colours to use. Can be a vector of colours e.g. cols = c("black","green") or pre-defined openair colours — see openColours for more details.
shade	The colour used for marking alternate years. Use "white" or "transparent" to remove shading.
xlab	x-axis label, by default "year".
y.relation	This determines how the y-axis scale is plotted. "same" ensures all panels use the same scale and "free" will use panel-specfic 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.x	See ref.y.
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")). See panel.abline in the lattice package for more details on adding/controlling lines.
key.columns	Number of columns used if a key is drawn when using the option statistic = "percentile".
name.pol	Names to be given to the pollutant(s). This is useful if you want to give a fuller description of the variables, maybe also including subscripts etc.
ci	Should confidence intervals be plotted? The default is FALSE.
alpha	The alpha transparency of shaded confidence intervals - if plotted. A value of 0 is fully transparent and 1 is fully opaque.
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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of date.breaks up or down.
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.
k	This is the smoothing parameter used by the 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.
...	Other graphical parameters are passed onto cutData and lattice:xyplot. For example, smoothTrend passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type

= "season". Similarly, common graphical arguments, such as xlim and ylim for plotting ranges and pch and cex for plot symbol type and size, are passed on xyplot, although some local modifications may be applied by openair. For example, axis and title labelling options (such as xlab, ylab and main) are passed to xyplot via quickText to handle routine formatting. One special case here is that many graphical parameters can be vectors when used with statistic = "percentile" and a vector of percentile values, see examples below.

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 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 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

As well as generating the plot itself, smoothTrend also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. Note that data is a list of two data frames: data (the original data) and fit (the smooth fit that has details of the fit and teh uncertainties). If retained, e.g. using output <-smoothTrend(mydata,"nox"), this output can be output <-smoothTrend(mydata,"nox"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summarise.

Author(s)

David Carslaw

See Also

TheilSen for an alternative method of calculating trends.

Examples

```
# load example data from package
data(mydata)
```

```
# 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
## Not run: smoothTrend(mydata, pollutant = c("no2", "o3", "pm10", "pm25"), pch = NA)

# percentiles
## Not run: smoothTrend(mydata, pollutant = "o3", statistic = "percentile",
percentile = 95)
## End(Not run)

# several percentiles with control over lines used
## Not run: 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

Utility function to prepare input data for use in openair functions

Usage

```
splitByDate(
  mydata,
  dates = "1/1/2003",
  labels = c("before", "after"),
  name = "split.by"
)
```

Arguments

`mydata`	A data frame containing a `date` field in hourly or high resolution format.
`dates`	A date or dates to split data by.
`labels`	Labels for each time partition.
`name`	The name to give the new column to identify the periods split

Details

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 label than there are dates. See examples below and in full `openair` documentation.

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"))
```

---

summaryPlot *Function to rapidly provide an overview of air quality data*

Description

This function provides a quick graphical and numerical summary of data. The location presence/absence of data are shown, with summary statistics and plots of variable distributions. `summaryPlot` can also provide summaries of a single pollutant across many sites.

Usage

```
summaryPlot(
  mydata,
  na.len = 24,
  clip = TRUE,
  percentile = 0.99,
  type = "histogram",
  pollutant = "nox",
  period = "years",
  avg.time = "day",
  print.datacap = TRUE,
  breaks = NULL,
  col.trend = "darkgoldenrod2",
  col.data = "lightblue",
  col.mis = rgb(0.65, 0.04, 0.07),
  col.hist = "forestgreen",
  cols = NULL,
  date.breaks = 7,
  auto.text = TRUE,
  ...
)
```

Arguments

mydata	A data frame to be summarised. Must contain a date field and at least one other parameter.
na.len	Missing data are only shown with at least na.len *contiguous* missing vales. The purpose of setting na.len is for clarity: with long time series it is difficult to see where individual missing hours are. Furthermore, setting na.len = 96, for example would show where there are at least 4 days of continuous missing data.
clip	When data contain outliers, the histogram or density plot can fail to show the distribution of the main body of data. Setting clip = TRUE, will remove the top 1 yield what is often a better display of the overall distribution of the data. The amount of clipping can be set with percentile.
percentile	This is used to clip the data. For example, percentile = 0.99 (the default) will remove the top 1 percentile of values i.e. values greater than the 99th percentile will not be used.
type	type is used to determine whether a histogram (the default) or a density plot is used to show the distribution of the data.
pollutant	pollutant is used when there is a field site and there is more than one site in the data frame.
period	period is either years (the default) or months. Statistics are calculated depending on the period chosen.
avg.time	This defines the time period to average the time series plots. Can be "sec", "min", "hour", "day" (the default), "week", "month", "quarter" or "year". For much increased flexibility a number can precede these options followed by a space. For example, a timeAverage of 2 months would be avg.time = "2 month".
print.datacap	Should the data capture % be shown for each period?
breaks	Number of histogram bins. Sometime useful but not easy to set a single value for a range of very different variables.
col.trend	Colour to be used to show the monthly trend of the data, shown as a shaded region. Type colors() into R to see the full range of colour names.
col.data	Colour to be used to show the *presence* of data. Type colors() into R to see the full range of colour names.
col.mis	Colour to be used to show missing data.
col.hist	Colour for the histogram or density plot.
cols	Predefined colour scheme, currently only enabled for "greyscale".
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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of date.breaks up or down.
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.

... Other graphical parameters. Commonly used examples include the axis and title labelling options (such as `xlab`, `ylab` and `main`), which are all passed to the plot via `quickText` to handle routine formatting. As `summaryPlot` has two components, the axis labels may be a vector. For example, the default case (`type = "histogram"`) sets y labels equivalent to `ylab = c("","Percent of Total")`.

Details

`summaryPlot` produces two panels of plots: one showing the presence/absence of data and the other the distributions. The left panel shows time series and codes the presence or absence of data in different colours. By stacking the plots one on top of another it is easy to compare different pollutants/variables. Overall statistics are given for each variable: mean, maximum, minimum, missing hours (also expressed as a percentage), median and the 95th percentile. For each year the data capture rate (expressed as a percentage of hours in that year) is also given.

The right panel shows either a histogram or a density plot depending on the choice of `type`. Density plots avoid the issue of arbitrary bin sizes that can sometimes provide a misleading view of the data distribution. Density plots are often more appropriate, but their effectiveness will depend on the data in question.

`summaryPlot` will only show data that are numeric or integer type. This is useful for checking that data have been imported properly. For example, if for some reason a column representing wind speed erroneosly had one or more fields with charcters in, the whole column would be either character or factor type. The absence of a wind speed variable in the `summaryPlot` plot would therefore indicate a problem with the input data. In this particular case, the user should go back to the source data and remove the characters or remove them using R functions.

If there is a field `site`, which would generally mean there is more than one site, `summaryPlot` will provide information on a *single* pollutant across all sites, rather than provide details on all pollutants at a *single* site. In this case the user should also provide a name of a pollutant e.g. `pollutant = "nox"`. If a pollutant is not provided the first numeric field will automatically be chosen.

It is strongly recommended that the `summaryPlot` function is applied to all new imported data sets to ensure the data are imported as expected.

Author(s)

David Carslaw

Examples

```
# load example data from package
data(mydata)

# do not clip density plot data
## Not run: summaryPlot(mydata, clip = FALSE)

# exclude highest 5 % of data etc.
## Not run: summaryPlot(mydata, percentile = 0.95)

# show missing data where there are at least 96 contiguous missing
```

```
# values (4 days)
## Not run: summaryPlot(mydata, na.len = 96)

# show data in green
## Not run: summaryPlot(mydata, col.data = "green")

# show missing data in yellow
## Not run: summaryPlot(mydata, col.mis = "yellow")

# show density plot line in black
## Not run: summaryPlot(mydata, col.dens = "black")
```

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,
  cols = "brewer1",
  rms.col = "darkgoldenrod",
  cor.col = "black",
  arrow.lwd = 3,
  annotate = "centred\nRMS error",
  key = TRUE,
  key.title = group,
  key.columns = 1,
  key.pos = "right",
  strip = TRUE,
  auto.text = TRUE,
  ...
)
```

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 — 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.

Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.

Note that often it will make sense to use type = "site" when multiple sites are available. This will ensure that each panel contains data specific to an individual site.

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.

cols — Colours to be used for plotting. Useful options for categorical data are avilable from RColorBrewer colours — see the openair openColours function for more details. Useful schemes include "Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3" — but see ?brewer.pal for the maximum useful colours in each. For user defined the user can supply a list of colour

	names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`.
`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.
`key`	Should the key be shown?
`key.title`	Title for the key.
`key.columns`	Number of columns to be used in the key. With many pollutants a single column can make to key too wide. The user can thus choose to use several columns by setting `columns` to be less than the number of pollutants.
`key.pos`	Position of the key e.g. "top", "bottom", "left" and "right". See details in `lattice:xyplot` for more details about finer control.
`strip`	Should a strip be shown?
`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.
`...`	Other graphical parameters are passed onto `cutData` and `lattice:xyplot`. For example, `TaylorDiagram` passes the option `hemisphere = "southern"` on to `cutData` to provide southern (rather than default northern) hemisphere handling of `type = "season"`. Similarly, common graphical parameters, such as `layout` for panel arrangement and `pch` and `cex` for plot symbol type and size, are passed on to `xyplot`. Most are passed unmodified, although there are some special cases where `openair` may locally manage this process. For example, common axis and title labelling options (such as `xlab`, `ylab`, `main`) are passed via `quickText` to handle routine formatting.

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

As well as generating the plot itself, `TaylorDiagram` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. 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.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

Author(s)

David Carslaw

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).

IPCC, 2001: Climate Change 2001: The Scientific Basis, Contribution of Working Group I to the Third Assessment Report of the Intergovernmental Panel on Climate Change [Houghton, J.T., Y. Ding, D.J. Griggs, M. Noguer, P.J. van der Linden, X. Dai, K. Maskell, and C.A. Johnson (eds.)]. Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 881 pp.

See Also

`taylor.diagram` from the `plotrix` package from which some of the annotation code was used.

Examples

```
## in the examples below, most effort goes into making some artificial data
## the function itself can be run very simply
## Not run:
## dummy model data for 2003
```

```
dat <- selectByDate(mydata, year = 2003)
dat <- data.frame(date = mydata$date, obs = mydata$nox, mod = mydata$nox)

## now make mod worse by adding bias and noise according to the month
## do this for 3 different models
dat <- transform(dat, month = as.numeric(format(date, "%m")))
mod1 <- transform(dat, mod = mod + 10 * month + 10 * month * rnorm(nrow(dat)),
model = "model 1")
## lag the results for mod1 to make the correlation coefficient worse
## without affecting the sd
mod1 <- transform(mod1, mod = c(mod[5:length(mod)], mod[(length(mod) - 3) :
length(mod)]))

## model 2
mod2 <- transform(dat, mod = mod + 7 * month + 7 * month * rnorm(nrow(dat)),
model = "model 2")
## model 3
mod3 <- transform(dat, mod = mod + 3 * month + 3 * month * rnorm(nrow(dat)),
model = "model 3")

mod.dat <- rbind(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 <- transform(dat, mod = mod + 2 * month + 2 * month * rnorm(nrow(dat)),
model = "model 1")
mod2a <- transform(mod2, mod = mod * 1.3)
mod3a <- transform(dat, mod = mod + 10 * month + 10 * month * rnorm(nrow(dat)),
model = "model 3")
mod.dat2 <- rbind(mod1a, mod2a, mod3a)
mod.dat$mod2 <- mod.dat2$mod

## now we have a data frame with 3 models, 1 set of observations
## and TWO sets of model predictions (mod and mod2)

## do for all models
TaylorDiagram(mod.dat, obs = "obs", mod = c("mod", "mod2"), group = "model")

## End(Not run)
## Not run:
## 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.

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,
  xlab = "year",
  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,
  shade = "grey95",
  auto.text = TRUE,
  autocor = FALSE,
  slope.percent = FALSE,
  date.breaks = 7,
  date.format = NULL,
  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	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. Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second 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 (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% confidence intervals for the value of the trend, choose alpha = 0.01 etc.
dec.place	The number of decimal places to display the trend estimate at. The default is 2.
xlab	x-axis label, by default "year".
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	This determines how the x-axis scale is plotted. "same" ensures all panels use the same scale and "free" will use panel-specfic scales. The latter is a useful setting when plotting data with very different values.

y.relation — This determines how the y-axis scale is plotted. "same" ensures all panels use the same scale and "free" will use panel-specfic 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".

shade — The colour used for marking alternate years. Use "white" or "transparent" to remove shading.

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.

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 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 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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of date.breaks up or down.

date.format — This option controls the date format on the x-axis. While TheilSen generally sets the date format sensibly there can be some situations where the user wishes to have more control. For format types see strptime. For example, to format the date like "Jan-2012" set date.format = "%b-%Y".

plot — Should a plot be produced. FALSE can be useful when analysing data to extract trend components and plotting them in other ways.

silent — When FALSE the function will give updates on trend-fitting progress.

... — Other graphical parameters passed onto cutData and lattice:xyplot. For example, TheilSen passes the option hemisphere = "southern" on to cutData

to provide southern (rather than default northern) hemisphere handling of `type = "season"`. Similarly, common axis and title labelling options (such as `xlab`, `ylab`, `main`) are passed to `xyplot` via `quickText` to handle routine formatting.

Details

The `TheilSen` function provides a collection of functions to analyse trends in air pollution data. 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.

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 https://dornsife.usc.edu/labs/rwilcox/software/. 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

As well as generating the plot itself, `TheilSen` also returns an object of class "openair". The object includes three main components: `call`, the command used to generate the plot; `data`, the data frame of summarised information used to make the plot; and `plot`, the plot itself. If retained, e.g. using `output <-TheilSen(mydata,"nox")`, this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including `print`, `plot` and `summary`.

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 http://uk-air.defra.gov.uk/library/aqeg/.

See Also

See smoothTrend for a flexible approach to estimating trends using nonparametric regression. The smoothTrend function is suitable for cases where trends are not monotonic and is probably better for exploring the shape of trends.

Examples

```
# load example data from package
data(mydata)

# 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)

# trend plot by each of 8 wind sectors
## Not run: TheilSen(mydata, pollutant = "o3", type = "wd", ylab = "o3 (ppb)")

# and for a subset of data (from year 2000 onwards)
## Not run: TheilSen(selectByDate(mydata, year = 2000:2005), pollutant = "o3", ylab = "o3 (ppb)")
```

`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,
  ...
)
```

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, a timeAverage of 2 months would be `period = "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", "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 in % 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" ... start.date is therefore used to force this type of sequence.

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.

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`.

`...` 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

See `timePlot` that plots time series data and uses `timeAverage` to aggregate data where necessary.

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)

## 2-weekly averages
## Not run: fortnight <- timeAverage(mydata, avg.time = "2 week")

## 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*

Description

Plot time series quickly, perhaps for multiple pollutants, grouped or in separate panels.

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",
  plot.type = "l",
  key = TRUE,
  log = FALSE,
  windflow = NULL,
  smooth = FALSE,
  ci = TRUE,
  y.relation = "same",
  ref.x = NULL,
  ref.y = NULL,
  key.columns = 1,
  key.position = "bottom",
  name.pol = pollutant,
  date.breaks = 7,
  date.format = NULL,
  auto.text = 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.
group	If more than one pollutant is chosen, should they all be plotted on the same graph together? The default is FALSE, which means they are plotted in separate panels with their own scaled. If TRUE then they are plotted on the same plot with the same scale.
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, a timeAverage of 2 months would be period = "2 month". See function timeAverage for further details on this.

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. Not used if avg.time = "default".

statistic — The statistic to apply when aggregating the data; default is the mean. Can be one of "mean", "max", "min", "median", "frequency", "sd", "percentile". Note that "sd" is the standard deviation and "frequency" is the number (frequency) of valid records in the period. "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 % 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 — 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 currently allowed in timePlot.

cols — Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for more details. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue")

plot.type — The lattice plot type, which is a line (plot.type = "l") by default. Another useful option is plot.type = "h", which draws vertical lines.

key — Should a key be drawn? The default is TRUE.

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 — This option allows a scatter plot to show the wind speed/direction as an arrow. The option is a list e.g. windflow = list(col = "grey",lwd = 2,scale

= 0.1). This option requires wind speed (ws) and wind direction (wd) to be available.

The maximum length of the arrow plotted is a fraction of the plot dimension with the longest arrow being scale of the plot x-y dimension. Note, if the plot size is adjusted manually by the user it should be re-plotted to ensure the correct wind angle. The list may contain other options to panel.arrows in the lattice package. Other useful options include length, which controls the length of the arrow head and angle, which controls the angle of the arrow head.

This option works best where there are not too many data to ensure over-plotting does not become a problem.

smooth — Should a smooth line be applied to the data? The default is FALSE.

ci — If a smooth fit line is applied, then ci determines whether the 95% confidence intervals are shown.

y.relation — This determines how the y-axis scale is 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")). See panel.abline in the lattice package for more details on adding/controlling lines.

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

key.position — Location where the scale key is to plotted. Can include "top", "bottom", "right" and "left".

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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of date.breaks up or down.

date.format — This option controls the date format on the x-axis. While timePlot generally sets the date format sensibly there can be some situations where the user wishes to have more control. 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.

... Other graphical parameters are passed onto cutData and lattice:xyplot. For example, timePlot passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, most common plotting parameters, such as layout for panel arrangement and pch and cex for plot symbol type and size and lty and lwd for line type and width, as passed to xyplot, although some maybe locally managed by openair on route, e.g. axis and title labelling options (such as xlab, ylab, main) are passed via quickText to handle routine formatting. See examples below.

Details

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.

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.

By default plots are shown with a colour key at the bottom and in the case of multiple pollutants or sites, strips on the left of each plot. Sometimes this may be overkill and the user can opt to remove the key and/or the strip by setting key and/or strip to FALSE. One reason to do this is to maximise the plotting area and therefore the information shown.

Value

As well as generating the plot itself, timePlot also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-timePlot(mydata,"nox"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

Author(s)

David Carslaw

See Also

TheilSen, smoothTrend, linearRelation, selectByDate and timeAverage for details on selecting averaging times and other statistics in a flexible way

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
## Not run: timePlot(mydata, pollutant = c("nox", "no2"), group = TRUE)

# alternative by normalising concentrations and plotting on the same
  scale
## Not run:
timePlot(mydata, pollutant = c("nox", "co", "pm10", "so2"), group = TRUE, avg.time =
  "year", normalise = "1/1/1998", lwd = 3, lty = 1)

## End(Not run)

# examples of selecting by date

# plot for nox in 1999
## Not run: timePlot(selectByDate(mydata, year = 1999), pollutant = "nox")

# select specific date range for two pollutants
## Not run:
timePlot(selectByDate(mydata, start = "6/8/2003", end = "13/8/2003"),
pollutant = c("no2", "o3"))

## End(Not run)

# choose different line styles etc
## Not run: timePlot(mydata, pollutant = c("nox", "no2"), lty = 1)

# choose different line styles etc
## Not run:
timePlot(selectByDate(mydata, year = 2004, month = 6), pollutant =
c("nox", "no2"), lwd = c(1, 2), col = "black")

## End(Not run)

# different averaging times

#daily mean O3
## Not run: timePlot(mydata, pollutant = "o3", avg.time = "day")
```

```
# daily mean O3 ensuring each day has data capture of at least 75%
## Not run: timePlot(mydata, pollutant = "o3", avg.time = "day", data.thresh = 75)

# 2-week average of O3 concentrations
## Not run: timePlot(mydata, pollutant = "o3", avg.time = "2 week")
```

`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 = "cluster",
  avg.time = "day",
  type = "default",
  statistic = "mean",
  normalise = FALSE,
  cols = "Set1",
  date.breaks = 7,
  date.format = NULL,
  key.columns = 1,
  key.position = "right",
  key.title = proportion,
  auto.text = TRUE,
  ...
)
```

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 e.g. `proportion = "cluster"` if the output from `polarCluster` 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, a timeAverage of 2 months would be `period = "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` `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.

`type` must be of length one.

`statistic` Determines how the bars are calculated. The default ("mean") will provide the contribution to the overall mean for a time interval. `statistic = "frequency"` will give the proportion in terms of counts.

`normalise` If `normalise = TRUE` then each time interval is scaled to 100. This is helpful to show the relative (percentage) contribution of the proportions.

`cols` Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and `RColorBrewer` colours — see the `openair` `openColours` function for more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`

`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. This does not always work as desired automatically. The user can therefore increase or decrease the number of intervals by adjusting the value of `date.breaks` up or down.

`date.format` This option controls the date format on the x-axis. While `timePlot` generally sets the date format sensibly there can be some situations where the user wishes to have more control. For format types see `strptime`. For example, to format the date like "Jan-2012" set `date.format = "%b-%Y"`.

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

`key.position` Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".

`key.title` The title of the key.

auto.text	Either TRUE (default) or FALSE. If TRUE titles and axis labels etc. will automatically try and format pollutant names and units properly e.g. by subscripting the '2' in NO2.
...	Other graphical parameters passed onto timeProp and cutData. For example, timeProp passes the option hemisphere = "southern" on to cutData to provide southern (rather than default northern) hemisphere handling of type = "season". Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to xyplot via quickText to handle routine formatting.

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.

Author(s)

David Carslaw

See Also

See timePlot for time series plotting, polarCluster for cluster analysis of bivariate polar plots and trajCluster for cluster analysis of HYSPLIT back trajectories.

Examples

```
## See manual for more examples e.g. related to clustering


## monthly plot of SO2 showing the contribution by wind sector
timeProp(mydata, pollutant = "so2", avg.time = "month", proportion = "wd")
```

---

timeVariation *Diurnal, day of the week and monthly variation*

Description

Plots the diurnal, day of the week and monthly variation for different variables, typically pollutant concentrations. Four separate plots are produced.

Usage

```
timeVariation(
  mydata,
  pollutant = "nox",
  local.tz = NULL,
  normalise = FALSE,
  xlab = c("hour", "hour", "month", "weekday"),
  name.pol = pollutant,
  type = "default",
  group = NULL,
  difference = FALSE,
  statistic = "mean",
  conf.int = 0.95,
  B = 100,
  ci = TRUE,
  cols = "hue",
  ref.y = NULL,
  key = NULL,
  key.columns = 1,
  start.day = 1,
  auto.text = TRUE,
  alpha = 0.4,
  ...
)
```

Arguments

mydata	A data frame of hourly (or higher temporal resolution data). 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.
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.
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 sub-plot.

`name.pol` Names to be given to the pollutant(s). This is useful if you want to give a fuller description of the variables, maybe also including subscripts etc.

`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`.

`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. See examples below.

`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% confidence intervals of the difference in means are also calculated. A horizontal dashed line is shown at y = 0.

`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. `conf.int` can be of length 2, which is most useful for showing quantiles. For example `conf.int = c(0.75,0.99)` will yield a plot showing the median, 25/75 and 5/95th quantiles.

`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 be used for plotting. Options include "default", "increment", "heat", "jet" and `RColorBrewer` colours — see the `openair` `openColours` function for more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be `cols = c("yellow","green","blue")`

`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")). See panel.abline in the lattice package for more details on adding/controlling lines.
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. See below for an example.
key.columns	Number of columns to be used in the key. With many pollutants a single column can make to key too wide. The user can thus choose to use several columns by setting columns to be less than the number of pollutants.
start.day	What day of the week should the plots 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.
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.
alpha	The alpha transparency used for plotting confidence intervals. 0 is fully transparent and 1 is opaque. The default is 0.4
...	Other graphical parameters passed onto lattice:xyplot and cutData. For example, in the case of cutData the option hemisphere = "southern".

Details

The variation of pollutant concentrations by hour of the day and day of the week etc. 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 vehicles type e.g. less heavy vehicles at weekends.

The timeVariation function makes it easy to see how concentrations (and many other variable types) vary by hour of the day and day of the week.

The plots also show the 95% confidence intervals in the mean. The 95% confidence intervals in the mean are calculated through bootstrap simulations, which will provide more robust estimates of the confidence intervals (particularly when there are relatively few data).

The function can handle multiple pollutants and uses the flexible type option to provide separate panels for each 'type' — see cutData for more details. timeVariation can also accept a group option which is useful if data are stacked. This will work in a similar way to having multiple pollutants in separate columns.

The user can supply their own ylim e.g. ylim = c(0,200) that will be used for all plots. ylim can also be a list of length four to control the y-limits on each individual plot e.g. ylim = list(c(-100,500),c(200,300),c(-40
These pairs correspond to the hour, weekday, month and day-hour plots respectively.

The option difference will calculate the difference in means of two pollutants together with bootstrap estimates of the 95% confidence intervals in the difference in the mean. 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 as such. Considering differences in this way can provide many useful insights and is particularly useful for model evaluation when information is needed about where a model differs from observations by many different time scales. The manual contains various examples of using difference = TRUE.

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.

In addition, timeVariation will work well with other variables if available. Examples include meteorological and traffic flow data.

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

As well as generating the plot itself, timeVariation also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data used to make the four components of the plot (or subplots); and plot, the associated subplots. If retained, e.g. using output <-timeVariation(mydata,"nox"), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

The four components of timeVariation are: day.hour, hour, day and month. Associated data.frames can be extracted directly using the subset option, e.g. as in plot(object,subset = "day.hour"), summary(output,subset = "hour"), etc, for output <-timeVariation(mydata,"nox")

Author(s)

David Carslaw

See Also

polarPlot, linearRelation

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)")

## End(Not run)

# multiple pollutants with concentrations normalised
## Not run: 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

## Not run: timeVariation(mydata, pollutant = "nox", type = "dst", local.tz = "Europe/London")

## In this case it is better to group the results for clarity:
## Not run: 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

## Not run: 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:

## Not run: timeVariation(mydata, pollutant = c("nox", "no2"), type = "o3", normalise = TRUE)

## difference in concentrations
## Not run: 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

## Not run:
mydata <- splitByDate(mydata, dates= "1/1/2003", labels = c("before Jan. 2003", "After Jan. 2003"))
timeVariation(mydata, pollutant = "no2", group = "split.by", difference = TRUE)

## End(Not run)

## sub plots can be extracted from the openair object
## Not run:
myplot <- timeVariation(mydata, pollutant = "no2")
plot(myplot, subset = "day.hour") # top weekday and plot

## End(Not run)

## individual plots
## plot(myplot, subset="day.hour") for the weekday and hours subplot (top)
## plot(myplot, subset="hour") for the diurnal plot
## plot(myplot, subset="day") for the weekday plot
## plot(myplot, subset="month") for the monthly plot

## numerical results (mean, lower/upper uncertainties)
## results(myplot, subset = "day.hour") # the weekday and hour data set
## summary(myplot, subset = "hour") #summary of hour data set
## head(myplot, subset = "day") #head/top of day data set
```

```
## tail(myplot, subset = "month") #tail/top of month data set

## plot quantiles and median
## Not run:
timeVariation(mydata, stati="median", poll="pm10", col = "firebrick")

## with different intervals
timeVariation(mydata, stati="median", poll="pm10", conf.int = c(0.75, 0.99),
col = "firebrick")

## End(Not run)
```

`trajCluster` *Calculate clusters for back tracectories*

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,
  plot = TRUE,
  type = "default",
  cols = "Set1",
  split.after = FALSE,
  map.fill = TRUE,
  map.cols = "grey40",
  map.alpha = 0.4,
  projection = "lambert",
  parameters = c(51, 51),
  orientation = c(90, 0, 0),
  by.type = FALSE,
  origin = 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.
plot	Should a plot be produced?
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. Note that the cluster calculations are separately made of each level of "type".
cols	Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for more details. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue")
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.
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.alpha	The transpency 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.
projection	The map projection to be used. Different map projections are possible through the mapproj package. See?mapproject for extensive details and information on setting other parameters and orientation (see below).
parameters	From the mapproj package. Optional numeric vector of parameters for use with the projection argument. This argument is optional only in the sense that certain projections do not require additional parameters. If a projection does require additional parameters, these must be given in the parameters argument.
orientation	From the mapproj package. An optional vector c(latitude,longitude,rotation) which describes where the "North Pole" should be when computing the projection. Normally this is c(90,0), which is appropriate for cylindrical and conic projections. For a planar projection, you should set it to the desired point of tangency. The third value is a clockwise rotation (in degrees), which defaults to the midrange of the longitude coordinates in the map.
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.
origin	If TRUE a filled circle dot is shown to mark the receptor point.
...	Other graphical parameters passed onto lattice:levelplot and cutData. Similarly, common axis and title labelling options (such as xlab, ylab, main) are passed to levelplot via quickText to handle routine formatting.

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

Returns a list with two data components. The first (`data`) contains the orginal data with the cluster identified. The second (`results`) contains the data used to plot the clustered trajectories.

Author(s)

David Carslaw

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

`importTraj`, `trajPlot`, `trajLevel`

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,
  map = TRUE,
  lon.inc = 1,
  lat.inc = 1,
  min.bin = 1,
  map.fill = TRUE,
  map.res = "default",
  map.cols = "grey40",
  map.alpha = 0.3,
  projection = "lambert",
  parameters = c(51, 51),
  orientation = c(90, 0, 0),
  grid.col = "deepskyblue",
  origin = TRUE,
  ...
)
```

Arguments

`mydata`	Data frame, the result of importing a trajectory file using `importTraj`
`lon`	Column containing the longitude, as a decimal.
`lat`	Column containing the latitude, as a decimal.
`pollutant`	Pollutant to be plotted. By default the trajectory height is used.
`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.

type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.

smooth — Should the trajectory surface be smoothed?

statistic — For trajLevel. By default the function will plot the trajectory frequencies.

For trajLevel, the argument method = "hexbin" can be used. In this case hexagonal binning of the trajectory *points* (i.e. a point every three hours along each back trajectory). The plot then shows the trajectory frequencies uses hexagonal binning. This is an alternative way of viewing trajectory frequencies compared with statistic = "frequency".

There are also various ways of plotting concentrations.

It is also possible to set statistic = "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 freqeuncies of the origin of air masses. The comparsion is made by comparing the percentage change in gridded frequencies. For example, such a plot could show that the top 10% of concentrations of PM10 tend to orginate from air-mass origins to the east.

If statistic = "pscf" then a Potential Source Contribution Function map is produced. If statistic = "cwt" then concentration weighted trajectories are plotted.

If statistic = "cwt" then the Concentration Weighted Trajectory approach is used. See details.

percentile — For trajLevel. The percentile concentration of pollutant against which the all trajectories are compared.

map — Should a base map be drawn? If TRUE the world base map from the maps package is used.

lon.inc — The longitude-interval to be used for binning data for trajLevel.

lat.inc — The latitude-interval to be used for binning data when trajLevel.

min.bin — For trajLevel the minimum number of unique points in a grid cell. Counts below min.bin are set as missing. For trajLevel gridded outputs.

map.fill — Should the base map be a filled polygon? Default is to fill countries.

map.res — The resolution of the base map. By default the function uses the 'world' map from the maps package. If map.res = "hires" then the (much) more detailed base map 'worldHires' from the mapdata package is used. Use library(mapdata). Also available is a map showing the US states. In this case map.res = "state" should be used.

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.alpha` The transpency 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.

`projection` The map projection to be used. Different map projections are possible through the `mapproj` package. See `?mapproject` for extensive details and information on setting other parameters and orientation (see below).

`parameters` From the `mapproj` package. Optional numeric vector of parameters for use with the projection argument. This argument is optional only in the sense that certain projections do not require additional parameters. If a projection does not require additional parameters then set to null i.e. `parameters = NULL`.

`orientation` From the `mapproj` package. An optional vector c(latitude, longitude, rotation) which describes where the "North Pole" should be when computing the projection. Normally this is c(90, 0), which is appropriate for cylindrical and conic projections. For a planar projection, you should set it to the desired point of tangency. The third value is a clockwise rotation (in degrees), which defaults to the midrange of the longitude coordinates in the map.

`grid.col` The colour of the map grid to be used. To remove the grid set `grid.col = "transparent"`.

`origin` should the receptor origin be shown by a black dot?

`...` other arguments are passed to `cutData` and `scatterPlot`. This provides access to arguments used in both these functions and functions that they in turn pass arguments on to. For example, `plotTraj` passes the argument `cex` on to `scatterPlot` which in turn passes it on to the `lattice` function `xyplot` where it is applied to set the plot symbol size.

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 `method = "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 freqeuncies of the origin of air masses of the highest concentration trajectories compared with the trajectories on average. The comparsion is made by comparing the percentage change in gridded frequencies. For example, such a plot could show that the top 10% of concentrations of PM10 tend to orginate from air-mass origins to 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 de-termining 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`.

Note

This function is under active development and is likely to change

Author(s)

David Carslaw

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

`importTraj` to import trajectory data from the King's College server and `trajPlot` for plotting back trajectory lines.

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.

Usage

```
trajPlot(
  mydata,
  lon = "lon",
  lat = "lat",
  pollutant = "height",
  type = "default",
  map = TRUE,
  group = NA,
  map.fill = TRUE,
  map.res = "default",
  map.cols = "grey40",
  map.alpha = 0.4,
  projection = "lambert",
  parameters = c(51, 51),
  orientation = c(90, 0, 0),
  grid.col = "deepskyblue",
```

```
  npoints = 12,
  origin = TRUE,
  ...
)
```

Arguments

mydata	Data frame, the result of importing a trajectory file using `importTraj`.
lon	Column containing the longitude, as a decimal.
lat	Column containing the latitude, as a decimal.
pollutant	Pollutant to be plotted. By default the trajectory height is used.
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. `type` can be up length two e.g. `type = c("season","weekday")` will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.
map	Should a base map be drawn? If `TRUE` the world base map from the `maps` package is used.
group	It is sometimes useful to group and colour trajectories according to a grouping variable. See example below.
map.fill	Should the base map be a filled polygon? Default is to fill countries.
map.res	The resolution of the base map. By default the function uses the 'world' map from the `maps` package. If `map.res = "hires"` then the (much) more detailed base map 'worldHires' from the `mapdata` package is used. Use `library(mapdata)`. Also available is a map showing the US states. In this case `map.res = "state"` should be used.
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.alpha	The transpency 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.
projection	The map projection to be used. Different map projections are possible through the `mapproj` package. See `?mapproject` for extensive details and information on setting other parameters and orientation (see below).

`parameters` From the `mapproj` package. Optional numeric vector of parameters for use with the projection argument. This argument is optional only in the sense that certain projections do not require additional parameters. If a projection does not require additional parameters then set to null i.e. `parameters = NULL`.

`orientation` From the `mapproj` package. An optional vector c(latitude, longitude, rotation) which describes where the "North Pole" should be when computing the projection. Normally this is c(90, 0), which is appropriate for cylindrical and conic projections. For a planar projection, you should set it to the desired point of tangency. The third value is a clockwise rotation (in degrees), which defaults to the midrange of the longitude coordinates in the map.

`grid.col` The colour of the map grid to be used. To remove the grid set `grid.col = "transparent"`.

`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 togther 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.

`...` other arguments are passed to `cutData` and `scatterPlot`. This provides access to arguments used in both these functions and functions that they in turn pass arguments on to. For example, `plotTraj` passes the argument `cex` on to `scatterPlot` which in turn passes it on to the `lattice` function `xyplot` where it is applied to set the plot symbol size.

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 (see example below), 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 aslo 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

See Also

`importTraj` to import trajectory data from the King's College server and `trajLevel` for trajectory binning functions.

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)
# 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"))
## End(Not run)

# plot by day, need a column that makes a date
## Not run:
lond$day <- as.Date(lond$date)
trajPlot(selectByDate(lond, start = "15/4/2010", end = "21/4/2010"),
type = "day")

## End(Not run)

# or show each day grouped by colour, with some other options set
## Not run:
 trajPlot(selectByDate(lond, start = "15/4/2010", end = "21/4/2010"),
group = "day", col = "jet", lwd = 2, key.pos = "right", key.col = 1)

## End(Not run)
# more examples to follow linking with concentration measurements...
```

`trendLevel` *trendLevel*

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 three other categorical properties. The default version of the plot uses y = hour of day, x = month of year and type = year to provide information on trends, seasonal effects and diurnal variations. However, x, y and type and summarising statistics can all be modified to provide a range of other similar plots.

Usage

```
trendLevel(
  mydata,
  pollutant = "nox",
  x = "month",
```

```
  y = "hour",
  type = "year",
  rotate.axis = c(90, 0),
  n.levels = c(10, 10, 4),
  limits = c(0, 100),
  cols = "default",
  auto.text = TRUE,
  key.header = "use.stat.name",
  key.footer = pollutant,
  key.position = "right",
  key = TRUE,
  labels = NA,
  breaks = NA,
  statistic = c("mean", "max", "frequency"),
  stat.args = NULL,
  stat.safe.mode = TRUE,
  drop.unused.types = TRUE,
  col.na = "white",
  ...
)
```

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	The name of the data series to use as the trendLevel x-axis. This is used with the y and type options to bin the data before applying statistic (see below). Other data series in mydata can also be used. (Note: trendLevel does not allow duplication in x, y and type options within a call.)
y	The names of the data series to use as the trendLevel y-axis and for additional conditioning, respectively. As x above.
type	See y.
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. (Note: 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. (Notes: 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.)
limits	The colour scale range to use when generating the trendLevel plot.
cols	The colour set to use to colour the trendLevel surface. cols is passed to openColours for evaluation. See ?openColours for more details.

auto.text — Automatic routine text formatting. auto.text = TRUE passes common lattice labelling terms (e.g. xlab for the x-axis, ylab for the y-axis and main for the title) to the plot via quickText to provide common text formatting. The alternative auto.text = FALSE turns this option off and passes any supplied labels to the plot without modification.

key.header, key.footer — Adds additional text labels above and/or below the scale key, respectively. For example, passing the options key.header = "",key.footer = c("mean","nox") adds the addition text as a scale footer. If enabled (auto.text = TRUE), these arguments are passed to the scale key (drawOpenKey) via quickText to handle formatting. The term "get.stat.name", used as the default key.header setting, is reserved and automatically adds statistic function names or defaults to "level" when unnamed functions are requested via statistic.

key.position — Location where the scale key should be plotted. Allowed arguments currently include "top", "right", "bottom" and "left".

key — Fine control of the scale key via drawOpenKey. See ?drawOpenKey for further details.

labels — If a categorical colour scale is required then these labels will be used. Note there is one less label than break. For example, labels = c("good","bad","very bad"). breaks must also be supplied if labels are given.

breaks — If a categorical colour scale is required then these breaks will be used. For example, breaks = c(0,50,100,1000). In this case "good" corresponds to values berween 0 and 50 and so on. 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 must also be supplied.

statistic — The statistic method to be use to summarise locally binned pollutant measurements with. Three options are currently encoded: "mean" (default), "max" and "frequency". (Note: Functions can also be sent directly via statistic. However, this option is still in development and should be used with caution. See Details below.)

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 below.)

stat.safe.mode — An addition protection applied when using functions direclty with statistic that most users can ignore. This option returns NA instead of running statistic on binned subsamples 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 below.)

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.

... Addition options are passed on to cutData for type handling and levelplot in lattice for finer control of the plot itself.

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

As well as generating the plot itself, trendLevel also returns an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-trendLevel(mydata), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summary.

Summary statistics can also be extracted directly using results, e.g. results(object) for output <-trendLevel(mydata).

Author(s)

Karl Ropkins and David Carslaw

See Also

openColours and drawOpenKey for more detailed plot control.

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)
```

---

windRose *Traditional wind rose plot and pollution rose variation*

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", bias.corr = TRUE, cols
  = "default", grid.line = NULL, width = 1, seg = NULL, auto.text
  = TRUE, breaks = 4, offset = 10, normalise = FALSE, max.freq =
  NULL, paddle = TRUE, key.header = NULL, key.footer = "(m/s)",
  key.position = "bottom", key = TRUE, dig.lab = 5, statistic =
  "prop.count", pollutant = NULL, annotate = TRUE, angle.scale =
  315, border = NA, ...)


  pollutionRose(mydata, pollutant = "nox", key.footer = pollutant,
  key.position = "right", key = TRUE, breaks = 6, paddle = FALSE,
  seg = 0.9, normalise = FALSE, ...)
```

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 — The user can supply a second set of wind speed and wind direction values with which the first can be compared. See details below for full explanation.

wd2 — see ws2.

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. Note, this argument is superseded in pollutionRose. See breaks below.

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 — 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.

Type can be up length two e.g. type = c("season","weekday") will produce a 2x2 plot split by season and day of the week. Note, when two types are provided the first forms the columns and the second the rows.

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 be used for plotting. Options include "default", "increment", "heat", "jet", "hue" and user defined. For user defined the user can supply a list of colour names recognised by R (type colours() to see the full list). An example would be cols = c("yellow","green","blue","black").

grid.line — Grid line interval to use. If NULL, as in default, this is assigned by windRose 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 — For pollutionRose 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.

breaks — Most commonly, the number of break points for wind speed in windRose or pollutant in pollutionRose. For windRose and the ws.int default of 2 m/s, the default, 4, generates the break points 2, 4, 6, 8 m/s. For pollutionRose, 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.

offset — The size of the 'hole' in the middle of the plot, expressed as a percentage of the polar axis scale, default 10.

normalise — If TRUE each wind direction segment of a pollution rose is normalised to equal one. This is useful for showing how the concentrations (or other parameters) contribute to each wind sector when the proprtion 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 (default) or FALSE. If TRUE plots rose using 'paddle' style spokes. If FALSE plots rose using 'wedge' style spokes.

key.header — Adds additional text/labels above and/or below the scale key, respectively. For example, passing windRose(mydata,key.header = "ws") adds the addition text as a scale header. Note: This argument is passed to drawOpenKey via quickText, applying the auto.text argument, to handle formatting.

key.footer — see key.footer.

key.position — Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".

key — Fine control of the scale key via drawOpenKey. See drawOpenKey for further details.

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

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".

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 " " then only the stastic is below the plot. Custom annotations may be added by setting value to c("annotation 1","annotation 2").

angle.scale — The wind speed scale is by default shown at a 315 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set angle.scale to another value (between 0 and 360 degrees) to mitigate such problems. For example angle.scale = 45 will draw the scale heading in a NE direction.

border	Border colour for shaded areas. Default is no border.
...	For pollutionRose other parameters that are passed on to windRose. For windRose other parameters that are passed on to drawOpenKey, lattice:xyplot and cutData. Axis and title labelling options (xlab, ylab, main) are passed to xyplot via quickText to handle routine formatting.

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.

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.

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

As well as generating the plot itself, windRose and pollutionRose also return an object of class "openair". The object includes three main components: call, the command used to generate the plot; data, the data frame of summarised information used to make the plot; and plot, the plot itself. If retained, e.g. using output <-windRose(mydata), this output can be used to recover the data, reproduce or rework the original plot or undertake further analysis.

An openair output can be manipulated using a number of generic operations, including print, plot and summarise.

Summarised proportions can also 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.

Note

windRose and pollutionRose both use drawOpenKey to produce scale keys.

Author(s)

David Carslaw (with some additional contributions by Karl Ropkins)

References

Applequist, S, 2012: Wind Rose Bias Correction. J. Appl. Meteor. Climatol., 51, 1305-1309.

This paper seems to be the original?

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

See drawOpenKey for fine control of the scale key.

See polarFreq for a more flexible version that considers other statistics and pollutant concentrations.

Examples

```
# load example data from package data(mydata)

# 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)

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

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

## End(Not run)

## 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")
```

Index