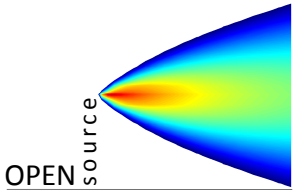
The **openair** manual

open-source tools for analysing air pollution data

King's College London



David Carslaw

version: 28th January 2015

NATURAL ENVIRONMENT RESEARCH COUNCIL

KING'S College LONDON

defra Department for Environment Food and Rural Affairs

UNIVERSITY OF LEEDS

The **openair** package and the documentation are provided in the hope that it is useful to you. While these tools are free, if you use them, we ask that acknowledgement be given to their source. Please cite:

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

Carslaw, D.C. (2015). The openair manual — open-source tools for analysing air pollution data. Manual for version 1.1-4, King's College London.

Every effort has been made to ensure the proper functioning of these tools and their effective and accurate documentation. We would like to be informed of any problems or queries related you have relating to their use. All correspondence should be sent to `mailto:david.carslaw@kcl.ac.uk`.

The website for this project is `http://www.openair-project.org`.

The development website is `https://github.com/davidcarslaw/openair`. **Please use the Github website 'Issues' tracker to report any bugs, make suggestions or ask questions.**

This document was produced using R version 3.1.2 and **openair** version 1.1-4.

Contents

Part I
Introduction to R for analysing data

1 Introduction

1.1 Background

This document provides information on the use of computer software called 'R' to analyse air pollution data. The document supports an initiative to develop and make available a consistent set of tools for analysing and understanding air pollution data in a free, open-source environment.

The amount of monitoring data available is substantial and increasing. In the UK alone there are thought to be over 1000 continuous monitoring sites. Much of the data available is only briefly analysed; perhaps with the aim of comparing pollutant concentrations with national and international air quality limits. However, as it will hopefully be seen, the critical analysis of air pollution data can be highly rewarding, perhaps yielding important information on pollutant sources that was previously unknown or unquantified.

There are however, several barriers that prevent the more effective analysis of air pollution data. These include user knowledge (knowing how to approach analysis and which techniques to use), cost (both in time and money) and access to specialist software that might be necessary to carry out all but the most simple types of analysis. Part of the aim of this document and its associated tools is to overcome some of these barriers.

1.2 Using this document

This document has been specifically written for those with an interest in analysing air quality data, although the techniques also lend themselves to wider atmospheric science problems and other sources of data e.g. traffic data. It is split into two parts.

Part I gives some background information on R and provides examples of using R with an air pollution data set from London. The intention here is to give an overview of how to use R and this is done mostly by considering actual air pollution data.

Part II describes dedicated functions for analysing air pollution data, which are available in the R 'package' called **openair** (Carslaw and Ropkins 2012). Even though the capabilities of the functions in Part II are greater than those highlighted in Part I, they are easier to apply.

The document assumes no previous knowledge of R. The document itself contains code which can be used directly in R (just copy and paste it in — see later). All the code used to make plots is shown in this manual. This code is the code that can be typed into R and can be copied directly from this document and pasted into R. The latter makes it easier to become familiar with the R language. The code also uses 'mark-up' to highlight functions, options etc. In addition, where a plot is produced, the code immediately precedes it. Users are encouraged to reproduce the plots shown and produce their own variations on them — for example, by plotting different pollutants. The document also contains extensive hypertext links to make it easy to navigate and cross-reference sections, figures etc. The document is meant to be a kind of work book, allowing users to work through the examples from start to finish.

However, it also acts as a reference guide to using R for the specific purposes of analysing monitoring data.

This document was produced entirely using free software. The document itself was produced using the LaTeX typesetting programme and all the plots were produced using R.

1.3 The open-source approach

The tools developed that are described here are *open-source*. This means that they are freely available to anybody and all the source code is open to scrutiny. Free software allows users the freedom to run, copy, distribute, study, change and improve the software. This philosophy is espoused at `http://www.gnu.org/philosophy/free-sw.html`.

The open-source approach is fundamental to this initiative. There are many advantages to open-source software tools beyond their zero direct cost. First is the belief that making tools open it will encourage their use and scrutiny. Second, by making tools available in this way it is more likely that others will contribute to them — perhaps identifying or fixing bugs, or maybe developing them further, as is the case for many open-source software projects. As described in Chambers (2007), the open-source approach can help lead to *trustworthy* software, and this is an important component of the aims here. It is a community approach that encourages trust and participation.

There are also difficulties in adopting an open-source approach however, not least the need for organisations to make money from their work. The development of these tools needs to be paid for somehow. However, there are organisations and individuals that see benefit in this way of working and are willing to fund or contribute to this initiative. Those who have contributed so far to this project are listed in the acknowledgements section.

In the environmental field there are maybe even more compelling arguments for using open-source tools. Many believe for example that those affected by environmental decisions reached through using tools/models/data should be able to scrutinise them — and why not? However, the reality is often far from this situation and there is often reliance on 'black boxes' where such scrutiny is not possible.

1.4 Aims

The aims of this document are as follows:

- To highlight the importance of looking at data effectively.
- To introduce the statistical software R and provide some background to the language.
- To show how R can be used to look at and understand air pollution monitoring data typically collected by local authorities, on behalf of governments and industry. However, the tools should also be very useful for those involved in academic research.
- To free-up time to think about air quality data rather than spending time preparing and analysing data. Part II in particular is aimed at providing functions that are *easy* to use so that more time can be spent on more productive activities.

The focus is very much on being pragmatic — the basic approach is to learn by doing. There are many books and on-line documents on R and these notes are not meant to

duplicate them. The approach used here is example-based, as it is our experience that this is the best way in which to learn to use R. Also, some of the concepts are easier to digest if applied to a problem or data set that is familiar. Thisf document cannot cover all the aspects of R that may be relevant or useful for the analysis of air pollution, but provides more of an introduction to how R can be used.

It is also important to stress that these functions are most useful when there is a clear purpose to the analysis. While *Exploratory Data Analysis* (EDA) is extremely valuable, there is no substitute for having a clear aim (Tukey 1977). This was perhaps best expressed by the statistician John Tukey who developed the idea of EDA:

> *The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.*

How this document was produced

One of the aims of this document was to ensure that users are able to reproduce all the analyses exactly as intended. This is not a straightforward task for a complex project under continual development. In addition, the large amount of code and functions presented provides many opportunities for things going wrong. It is easy, for example, to show how a function works and provide the results/plot; update the function and then find out that the options have changed and it no longer runs as intended. In other words, the documentation and the tools themselves go out of sync. Even cutting and pasting text can easily go wrong — as we have discovered.

For this reason we have adopted an innovative approach to ensuring that everything works as intended. This document blends text with code in that the whole document must be 'run' to produce it. Each time a version of this documentation is produced, all the code is run at the same time to generate all the various outputs e.g. plots. **This means that all users should be able to reproduce exactly the same outputs as shown in this report.**[1]

The approach uses a package called knitr (Xie 2013a; Xie 2013b). knitr mixes a typesetting system (LaTeX) with R. When a document is produced, blocks of code embedded in the LaTeX file are recognised and run in R. In some ways it reverses the 'normal' way of doing things — rather than document computer code, the documentation is written to contain the code. The document will not compile if the code does not function — it is as simple as that. In our document, most of the outputs are graphics, but increasingly quantitative information will also be produced.

In adopting this approach we found many problems with the manual (and some functions), even though we took care to develop this work. In time we also have ideas for using this approach to automatically carry out analyses. Imagine a report similar to this (but written more as a tutorial) where the data used are your own data. This approach would have the major advantage that all the analyses would be directly relevant to the user in question, and entirely reproducible.

1.5 Can I use these tools for commercial purposes?

In short, the answer is yes. Part of the aim of producing these tools was to allow *anybody* to use them for *any* purpose. Indeed, this is the principal purpose of the Knowledge Exchange grant. Our work is very much released in the true spirit of the Free Software Foundation `http://www.fsf.org/`. However, there are a few points users should note:

1. If you use these tools in reports, publications etc., we ask that you cite their source (see the preamble at the beginning concerning how to do this).
2. It is not possible to provide a guarantee or warranty for these tools, although we have tried hard to ensure they function as documented and are adopting methods for quality control.
3. We request that should you find these tools useful and enhance them, that you make such enhancements available to us for wider use.
4. For more detailed information on the various licenses under which R and its packages operate, the user is referred to the R website.

2 Statistical software R

2.1 Introduction to R

R is a computer programming language developed specifically for the purposes of analysing data (R-project). It is variously described as a statistical system, a system for statistical computation and graphics, and an environment for data analysis and statistics. Its origins go back to innovative developments at Bell Laboratories in the USA during the 1970s, where the language developed was called S. Since that time S has become commercial software and is sold as S-Plus by the Insightful Corporation.

Over the past 10 years or so an open-source version of S has been developed called R. Unlike some open-source software R is highly developed, highly capable and well established. It is very robust and works on a wide range of platforms (e.g. Windows, Mac, and Linux). One of its strengths is the large and growing community of leading researchers that contribute to its development. Increasingly, leading statisticians and computational scientists choose to publish their work in R; making their work available worldwide and encouraging the adoption and use of innovative methodologies.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License.

Another key strength of R is the package system of its operation. The base software, which is in itself highly capable (e.g. offering for example linear and generalized linear models, nonlinear regression models, time series analysis, classical parametric and nonparametric tests, clustering and smoothing), has been greatly extended by additional functionality. Packages are available to carry out a wide range of analyses including: generalized additive models, linear and non-linear modelling, regression trees, Bayesian approaches etc.

For air pollution purposes, R represents the *ideal* system with which to work. Core features such as effective data manipulation, data/statistical analysis and high quality graphics lend themselves to analysing air pollution data. The ability to develop one's own analyses, invent new plots etc. using R means that advanced tools can be developed for specific purposes. Indeed, Part II of this document is focussed on the use of dedicated tools for air quality analysis. The use of R ensures that analyses

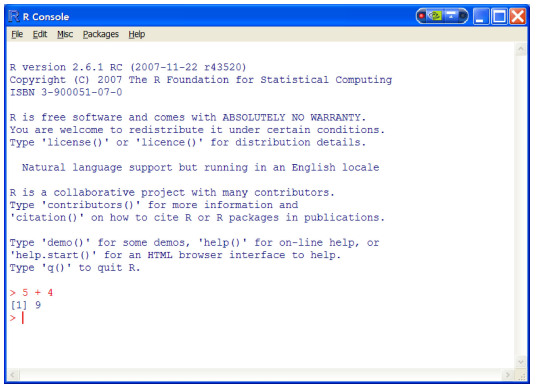


FIGURE 2.1 The basic R console.

and graphics are not constrained to 'off the shelf' tools. These tools will often contain functionalities that are either part of the R base system or that exist through specific packages.

The principal difficulty in using R is the steep learning curve in being able to use it effectively. Many with programming experience (e.g. FORTRAN, C++) tend to find R an easier language to learn than those that have no experience in programming. However, for most users this is not the experience. One of the principal difficulties is the lack of a nice graphical user interface (GUI). However, the more one develops tools in R, the more it is realised that a GUI approach significantly constrains capabilities (try plotting trends for each hour of the day. While it certainly takes considerable effort to learn to use R, the rewards are also high.

2.2 Why use R?

There are numerous reasons why R is a good choice for analysing data. A few are listed below.

- It is ***free***! This for many people is the key attraction. For this reason, R has become increasingly popular among a wide range of users including universities and businesses.
- It works on several platforms e.g. Windows, Mac OS, and Linux. This makes it very portable and flexible. It is also extremely robust; it is remarkably bug-free and crashes are very rare.
- It has been designed with data analysis in mind — to carry out analysis quickly, effectively and reliably.
- The base system offers a very wide range of data analysis and statistical abilities.
- Excellent graphics output that will grace any report. Furthermore, all the default plotting options have been carefully thought out unlike Excel, for example, whose default plotting options are very poor. There are over 4000 *packages* that offer all kinds of wonderful analysis techniques not found in any other software package. R continues to rapidly grow in popularity, which has resulted in better information for users e.g. there are now many dedicated books.

2.3 Why not use R?

For all its inherent strengths, R does have drawbacks. Here are a few.

- It is difficult to learn — there is a steep 'learning curve'.[2] Many would argue this is not the case; particularly if you are familiar with another programming language such as C++ or Fortran. However, our experience is that it is hard work for most people.
- There is no Graphical User Interface; instead one has something that looks like a DOS screen where one types commands. This seems very old-fashioned and at odds with the modern computing experience, but with use you will begin to see this as real advantage. A whole report can be based on a series of 'scripts' that can be run to carry out analysis. It is not so easy to record mouse movements and menu choices ...
- There is no help or support — or little that is apparent. This is true, especially compared with commercial software. However, there is extensive on-line help available and many people have written manuals and guides. This document also aims to address the lack of direct, specific support for air pollution analysis.

We have used several systems for data analysis over the years. These have included databases, Visual Basic, GIS, contouring software, statistics software and Excel. We always found it frustrating to do some analysis in Visual Basic, for example, then transfer it to another application for plotting and further analysis. We wrote code to carry out various statistical analyses too. For us, R does all these thing in one place to a very high standard.

Another increasingly important aspect is the ability to run *simulations*. Until relatively recently computers were not powerful enough to routinely run simulations using methods such as randomization, Monte Carlo and bootstrap calculations. All these approaches can greatly enhance many analyses and they are used in many of the functions described in this document. Often, the reason is to obtain a better estimate of uncertainties, which are important to consider when trying to draw inferences from data. Also, many of these methods were essentially inaccessible; or at least beyond consideration for most. This is where R excels — all these methods exist or can be coded building on some of the base functions.

2.4 Some useful R resources

The web is the best place to find information on R. There are many useful documents that people have written (see `http://www.r-project.org/`) under documentation/other. The official user guides can be hard going for many that are completely new to R, but worth a look later.

We also have the *R Book* by Michael Crawley, which is quite useful (Crawley 2007). This is a big book full of examples. It is not well laid-out and has come in for some criticism but we have nevertheless found it to be useful. Another useful book is by John Maindonald and John Braun (Maindonald and Braun 2007). Perhaps the best introductory book is that by Dalgaard (2008), which provides a gentle introduction to R, is well-written and up to date. Dalgaard (2008) also covers basics statistics, with the added benefit of their use in R. It is worth also having a look at the R-project pages as they provide a list of R books—even forthcoming titles not yet published. For those getting into R more seriously, I strongly recommend Spector (2008) for data manipulation and Sarkar (2007) for graph plotting using the `lattice` package.

There are some useful contributed documents on the R web-pages. Under the *Documentation* section on the main page have a look at Other | contributed documentation. We found that 'An Introduction to R: Software for Statistical Modelling &

[2] A steep learning curve means you learn a lot per unit time ...

Computing' by Petra Kuhnert and Bill Venables was very useful and also the document from David Rossiter (also check his web site — he has made available lots of high quality information on R and its uses at http://www.itc.nl/personal/rossiter/pubs/list.html#pubs_m_R).

A very good (and free) book is by Helsel and Hirsch (2002) from the US Geological Survey (see http://pubs.usgs.gov/twri/twri4a3/). Although the book does not consider R, it provides an excellent grounding in the types of statistics relevant to air pollution monitoring. This book is all about water resources and statistical methods that can be applied to such data. However, there is a great deal of similarity between water pollution and air pollution (e.g. seasonal cycles). One of the authors (Hirsch) is responsible for developing several important methods for analysing trends that are very widely used; including in this document. The document is also very well written and applied in nature — ideal reading for a background in statistical methods relevant to air pollution.

For those that really want to learn R itself, then Matloff (2011) is well worth considering. The book is full of useful examples about how R works and how to use it.

3 Basic use of R

3.1 Introduction

R will of course need to be installed on a computer and this is described in Appendix A. The install procedure is straightforward for Windows systems (the operating system most likely used by those with interests in **openair**). Using R will be a very different experience for most users used to software such as spreadsheets. One of the key differences is that data are stored as *objects*. These objects can take many forms, but one of the most common is called a *data frame*, which is widely used to store spreadsheet-like data. Before working with such data, it is useful to see how simpler data are stored.

3.2 Use it as a calculator

To use R, one should type directly in the console shown in Figure 2.1. Later, it will be shown when more than one line needs to be input, alternative methods can be used to send commands to the console.

R can be used to do simple maths; in this example type in '5 + 4' and press return. The [1] shows that this is the first (and only in this case) result.

```
5 + 4
```

```
## [1] 9
```

The output (9) has a '[1]' next to it showing it is the first (and only in this case) result. To assign a value to a variable, in this case x, type

```
x  = 5
```

Often it is useful to recall previous commands that have been entered, perhaps modifying them slightly. To recall previous commands the up (↑) and down arrow (↓) keys can be used.

To assign a variable to a value most R users will use the 'assignment operator' (<-). However, most new users to R find this unnecessarily unusual. In this document we mostly use <- but for most circumstances the = will work the same way.

```
x
```

```
## [1] 5
```

Note! R is case sensitive

In the case above using a capital *X* gives an error:

```
x * 5
```

```
## [1] 25
```

```
X * 5
```

```
## Error in eval(expr, envir, enclos): object 'X' not found
```

It is often necessary to modify a line that has been input previously (this is part of the interactive strength of R). To recall the previous line(s) for editing use the up arrow (↑) on the keyboard.

One of the very important features of R is that it considers 'vectors'. In the example above, `x` was a single integer 5. However, it can equally represent any sequence of numbers. In the example below, we are going to define two variables `x` and `y` to be samples of 10 random numbers between 0 and 1. In this case there is a special in-built function to generate these numbers called `runif`.

generate random numbers

```
x = runif(10)
y = runif(10)
```

To see what `x` looks like, just type it in:

```
x
```

```
##  [1] 0.41996169 0.88058533 0.56438332 0.76625773 0.22376443 0.91309150
##  [7] 0.77205452 0.02406509 0.28999768 0.70998951
```

This is one of the most powerful features of R; it also makes it easier to code functions. The 'traditional' way of doing this would be to have an array and loops, something like:

```
# for i = 1 to 10
#    x(i) = runif(1)
# next
```

The R code is much neater and can also be extended to matrices and arrays.

Another useful thing to do sometimes is to generate a sequence of numbers. Here the functions `seq` and `rep` are very useful. We first show their basic use and then show a useful example relevant to monitoring data.

generate a sequence numbers

To generate a sequence of numbers between 1 and 10:

```
z = seq(1:10)
z
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

To divide all these number by 10, simply type:

```
z / 10
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

This code again highlights *vectorised* calculations in R: you need not treat each element of `z` separately. For more information on selecting parts of vectors, and subsets of data in general see (§5.2).

In fact, for this particular example you can just type `z <- 1:10`. Many other possibilities are also available. For example, to generate a sequence of numbers between 1 and 10 using 23 numbers (bit odd, but you get the idea!):

```
z = seq(1, 10, length.out = 23)
z
```

```
##  [1]  1.000000  1.409091  1.818182  2.227273  2.636364  3.045455  3.454545
##  [8]  3.863636  4.272727  4.681818  5.090909  5.500000  5.909091  6.318182
## [15]  6.727273  7.136364  7.545455  7.954545  8.363636  8.772727  9.181818
## [22]  9.590909 10.000000
```

repeating a sequence of numbers or characters

The function `rep` on the other hand repeats numbers a certain number of times. For example, to repeat 1, 10 times:

```
rep(1, 10)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1
```

Now, these functions can be combined. A scenario is that you have a data set of concentrations exactly one year long and you want the hour of the day (if you have the date/time there are other ways to do this shown later). A year is 365×24 hours long (8760 hours). What is needed is to repeat the sequence 0–23, 365 times. This is done by:

```
hour = rep(seq(0, 23), 365)
```

Easy! There are loads of variations on this theme too. It's the sort of thing that should be easy but often is a pain to do in other software.

A very common thing to do in R is combine a series of numbers. This is done by *concatenating* them. For example, if one wanted `p` to represent the numbers 1, 3, 5, 10:

```
p = c(1, 3, 5, 10)
p
```

```
## [1]  1  3  5 10
```

The use of `c()` is extremely common in R and is used in almost every analysis. One example is when setting the axis limits in plots. A long-hand way of doing this might be to separately set the lower and upper ranges of the limits (e.g. x.lower = 0, x.upper = 100). However, with the `c` command, it is more compactly written as `xlim = c(0, 100)`.

3.3 Basic graph plotting

One of R's strengths is the ease with which graphs can be plotted. Almost all graphs in R use an 'ink on paper' approach — once something is added it cannot be changed. You will need to plot it again.

```
plot(x, y)
```

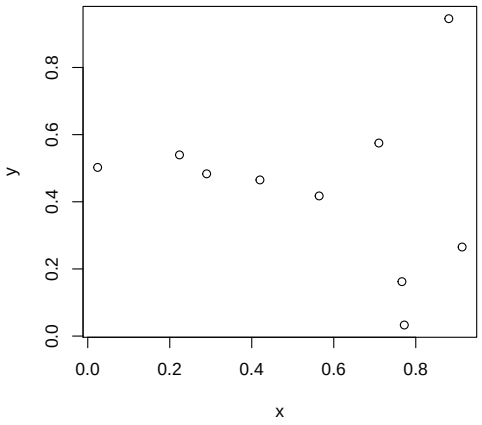


FIGURE 3.1 Plot of 10 random numbers between 0 and 1.

```
z <- rnorm(1000)
hist(z)
```

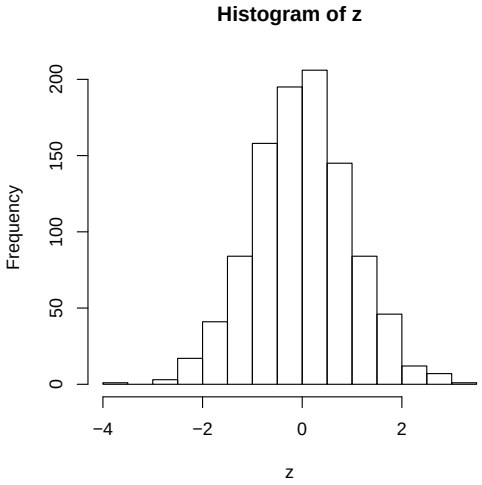


FIGURE 3.2 Histogram of 1000 random numbers drawn from a normal distribution.

To plot `x` against `y` as a scatter `plot`, simply use the plot function.

You should end up with something similar to Figure 3.1 (but not exactly the same, because we are plotting random numbers).

Because R is mostly considered as a system for doing statistics, it should come as no surprise that doing some basic statistics and plotting some basic graphs is easy. One useful plot is a histogram. There is an in-built function called `hist`. To make it a bit more interesting, we are first going to draw 1000 random samples from a normal distribution, again the in-built function `rnorm` can do this for you, as shown for Figure 3.2.

The plot should look something like Figure 3.2. This should give you some idea that using R can be very efficient and straightforward for basic plotting. To make a histogram in Excel by contrast is pretty painful. Note also that the bin width size has been automatically chosen — in this case to be 0.05.

3.4 Getting help

There are various ways to get help. For particular functions like `plot, hist, seq` etc., simply type `help(plot)` or `?plot` (or whatever) to bring up a screen that lists all the options of a function. Almost always shown are examples of use too. Failing that, try the R-project website, choose 'Search' on the first page and then 'R site search'. These web pages are very useful because they contain questions asked by R-users and answered by R experts. You need to read the 'posting guide' before sending question to this site — people supply answers in their own time so make sure you have explored all avenues first.

To search for information in installed packages, `help.search` is useful e.g.

```
help.search("polar plot")
```

4 Using R to analyse air pollution monitoring data

4.1 Getting data into R

So far we have just made up our data using some in-built R functions. A more likely scenario is that you have a data file that you want to analyse. R has many powerful facilities for importing data. However, we are going to keep things simple because it is worth getting your data into a simple format in the first place. Data more often than not is represented as columns on a rectangular grid — and unfortunately is often in Excel with all of its complexity of sheets and formatting. *It is strongly recommended that you work with data in a .csv format.* The files are simple, small and easily read by a wide variety of software.

The importance of data preparation

Experience shows that it is well worth putting a lot of effort into making sure your data are correct before you start analysing them. When you prepare a file carry out some basic checks. For R it is advisable to have variable names with no spaces. It is also more convenient to keep variable names in lower case because it is more difficult to type in capital letters.

It is also convenient to keep your variables names simple because you may need to refer to them lots. Also, we find it is a good idea to keep everything in lower case letters, again because they are quicker to type.

A file containing real monitoring data from Marylebone Road in London has been put together. This is a 'warts n' all' file and contains missing data — typical of that available elsewhere. We have also put in some basic meteorological data (wind speed and direction). The file is called 'example data long.csv'. This file contains a column representing dates/times in the format dd/mm/yyyy HH:MM, which is very common in data sets of monitoring data. The file is available from `http://www.openair-project.org`.

The easiest way to work with this file is first to set the working directory of R to be the same as that where the file is. This is done using the File|Change dir... command in R.

Read in a csv file

To read the file in, type (or better still copy and paste this directly into R):

```
## note! - remember to change the directory
mydata <- read.csv("~/openair/Data/example data long.csv", header = TRUE)
mydata$date <- as.POSIXct(strptime(mydata$date, format = "%d/%m/%Y %H:%M", tz = "GMT"))
```

Note that the **openair** package has tools to make this easier — see Getting data into **openair**. The information presented here shows how to do things using basic R functions.

format date-times

This reads the file into something called a `data.frame`, which is the most common way in which R stores data (others include vectors, matrices and arrays).

An alternative if you want to bring up a dialog box that allows you to browse the file system (and then format the date) is:

```
mydata <- read.csv(file.choose(), header = TRUE)
mydata$date <- as.POSIXct(strptime(mydata$date, format = "%d/%m/%Y %H:%M", tz = "GMT"))
```

Another neat way of getting data into R if you are interested in having a quick look at data is to read data directly from the clipboard. If you are using Excel you can copy some data (which may or may not include column headers; better if it does) and read it into R easily by:

```
mydata <- read.delim("clipboard", header = TRUE)
```

Which assumes that the data did have a header field. If not R will provide column names like V1, V2 etc. and the option `header = FALSE` should be used instead.

We will come onto the second line in a minute. It is *always* useful to check to see what has been loaded and using the `summary` command in R is one useful way to check:

view a summary of the data

```
summary(mydata)
```

```
##       date                          ws              wd           nox
##  Min.   :1998-01-01 00:00:00   Min.   : 0.000   Min.   :  0   Min.   :   0.0
##  1st Qu.:1999-11-14 15:00:00   1st Qu.: 2.600   1st Qu.:140   1st Qu.:  82.0
##  Median :2001-09-27 06:00:00   Median : 4.100   Median :210   Median : 153.0
##  Mean   :2001-09-27 06:00:00   Mean   : 4.489   Mean   :200   Mean   : 178.8
##  3rd Qu.:2003-08-10 21:00:00   3rd Qu.: 5.760   3rd Qu.:270   3rd Qu.: 249.0
##  Max.   :2005-06-23 12:00:00   Max.   :20.160   Max.   :360   Max.   :1144.0
##                                NA's   :632      NA's   :219   NA's   :2423
##       no2              o3              pm10             so2
##  Min.   :  0.00   Min.   : 0.000   Min.   :  1.00   Min.   : 0.000
##  1st Qu.: 33.00   1st Qu.: 2.000   1st Qu.: 22.00   1st Qu.: 2.175
##  Median : 46.00   Median : 4.000   Median : 31.00   Median : 4.000
##  Mean   : 49.13   Mean   : 7.122   Mean   : 34.38   Mean   : 4.795
##  3rd Qu.: 61.00   3rd Qu.:10.000   3rd Qu.: 44.00   3rd Qu.: 6.500
##  Max.   :206.00   Max.   :70.000   Max.   :801.00   Max.   :63.205
##  NA's   :2438     NA's   :2589     NA's   :2162     NA's   :10450
##        co              pm25
##  Min.   : 0.000   Min.   :  0.0
##  1st Qu.: 0.635   1st Qu.: 13.0
##  Median : 1.140   Median : 20.0
##  Mean   : 1.464   Mean   : 21.7
##  3rd Qu.: 1.980   3rd Qu.: 28.0
##  Max.   :19.705   Max.   :398.0
##  NA's   :1936     NA's   :8775
```

The `summary` function is extremely useful. It shows that there are 9 variables: date, ws, wd .... It provides the minimum, maximum, mean, median, and the first and third quantiles. It shows for example that NO_x ranges from 0 to 1144 ppb and the mean is 178.8 ppb. Also shown is something called `NA's`, which are missing data. For NO_x

there are 2423 missing values for example. These missing values are very important and it is also important to know how to deal with them. When R read the .csv file, it automatically assigned missing data the value NA.

Also note that the date is read in as a character string and R does not know it is a date. Dealing properly with dates and times in any software can be very difficult and frustrating. There are time zones, leap years, varying lengths of seconds, minutes, hours etc.; all in all it is highly idiosyncratic.[3] R does however have a robust way of dealing with dates and times. In this case it is necessary to deal with dates and times and two functions are used to convert the date to something recognised as such by R. The function `strptime` tells R what format the data are in — in this case day/month/year hour:minute i.e. it 'strips' the date out of the character string. In addition, R is told that the data are GMT. This allows great flexibility for reading in dates and times in a wide range of formats. The `as.POSIXct` function then converts this to a convenient format to work with. This may appear to be complicated, but it can be applied in the same way to all files and once done, it is possible to proceed without difficulty. The other need for storing dates/times in this way is to deal with GMT/BST (or 'daylight savings time'). Some of the functions in Part II use time zone information to process the data, because, for example, emissions tend to vary by local time and not GMT. Note that the **openair** package will automatically do these conversions if the data are in a format dd/mm/yyyy HH:MM.[4] For advice on dealing with other time zones, please see Section 9.1.

show the variable names

If you just want to know what variables have been read in, it is easier to type

```
names(mydata)
```

```
##  [1] "date" "ws"   "wd"   "nox"  "no2"  "o3"   "pm10" "so2"  "co"   "pm25"
```

To access a particular variable, one has to refer to the data frame name `mydata` and the variable itself. For example, to refer to `nox`, you must type `mydata$nox`. There are other ways to do this such as attaching the data frame and using the `with` command, but this is the basic way of doing it.

Now one can get some summary information on one variable in the data frame e.g.

```
summary(mydata$nox)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     0.0    82.0   153.0   178.8   249.0  1144.0    2423
```

missing data are represented as NA (not available) in R

Missing data can have an important effect on various R functions. For example, to find the mean NO_x concentration use the function `mean`:

```
mean(mydata$nox)
```

```
## [1] NA
```

The result NA is because `nox` contains some missing data. Therefore, it is necessary to exclude them using the `na.rm` command:

```
mean(mydata$nox, na.rm = TRUE)
```

```
## [1] 178.7986
```

[3] Just to emphasise this difficulty, note that a 'leap second' will be added at the end of 2008 due to the occasional correction needed because of the slowing of the Earth's rotation.

[4] The **openair** package will in time contain several functions to make it easier to import data and deal with dates and times.

```
hist(mydata$no2)
```

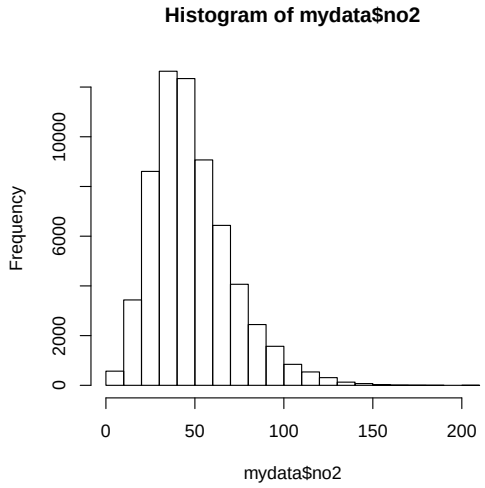


FIGURE 4.1 Histogram of NO_2 concentrations at Marylebone Road.

```
hist(mydata$no2, main = "Histogram of nitrogen dioxide",
     xlab = "Nitrogen dioxide (ppb)")
```

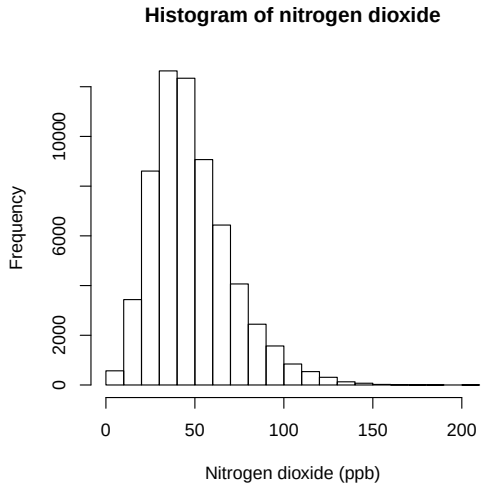


FIGURE 4.2 Histogram of NO_2 concentrations at Marylebone Road — with better labels.

Often it is sensible (and easier) to remove all the missing data from a data frame:

```
newdata <- na.omit(mydata)
```

This makes a new data frame with *all* the missing data removed. However, we will work with the original data which includes missing data.

4.2 More sophisticated plotting

Let's have a look at a histogram of NO_2 concentrations, shown in Figure 4.1. Looks good, but it can easily be tidied up. It needs a new title and x-axis caption. This is easy using some of the in-built options in the **hist** function (Figure 4.2).

```
hist(mydata$no2, main = "Histogram of nitrogen dioxide",
     xlab = "Nitrogen dioxide (ppb)", col = "lightblue")
```

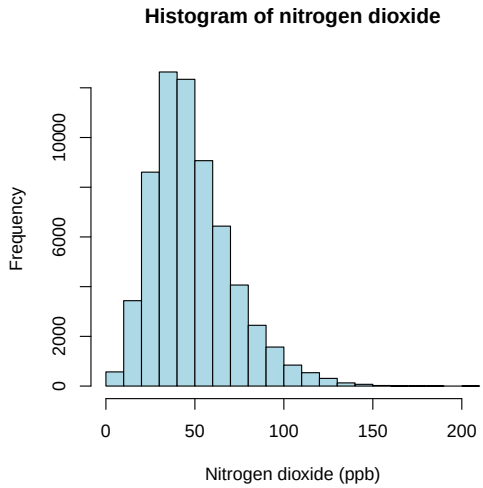


FIGURE 4.3 Histogram of NO_2 concentrations at Marylebone Road – with better labels and some colour.

```
dens <- density(mydata$no2, na.rm = TRUE)
plot(dens, main = "Density plot of nitrogen dioxide",
     xlab = "Nitrogen dioxide (ppb)")
```

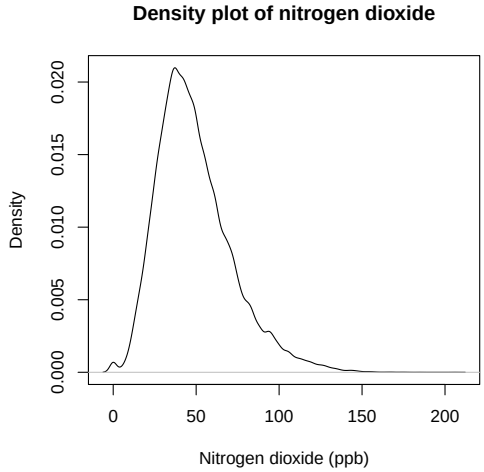


FIGURE 4.4 Density plot of NO_2 concentrations at Marylebone Road.

Or maybe you want to shade the bars (Figure 4.3)...

Another very useful plotting function is a *density* plot using the `density` function (Figure 4.4). This has the advantage over the histogram of avoiding trying to select a bin width and for some data can give a much clearer indication of its distribution.

4.3 Plotting time series with different averaging times

This is one of the most useful things to do with air pollution data. It can be a pain in some software to plot time series data with different averaging times — try for example plotting daily or weekly means in Excel. Now that we have a proper date format in R,

```
plot(mydata$date, mydata$nox, type = "l", xlab = "year",
     ylab = "Nitrogen oxides (ppb)")
```

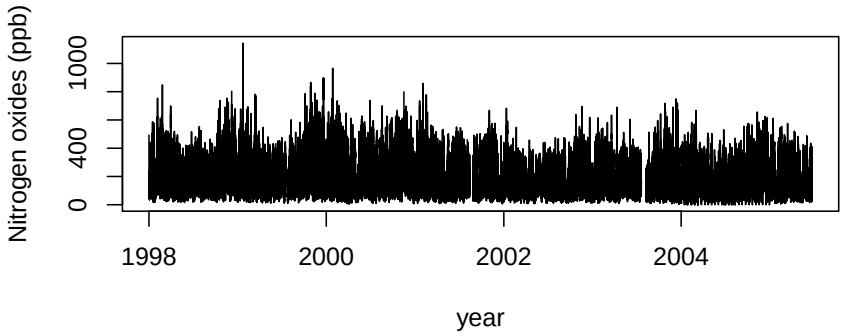


FIGURE 4.5 Hourly time series plot of NO_x at Marylebone Road.

we can do all sorts of things. First we show the basic plot of hourly data using a line plot. This plots all the data in its native format (hours), shown in Figure 4.5.

Use functions in openair to flexibly aggregate data on different time bases

The **openair** package has functions to calculate and plot data on almost any time-averaging basis. See §(31.4) and §(17) for more details.

Say you just want to plot a section of the data — say the first 500 points. How do you do this? Selecting subsets of data is another of the real strengths of R. Remember that the variables are in vector form. Therefore, to select only the first 500 values of a variable *x* you can type `x[1:500]`, or values from the 300th to 400th data points `x[300:400]`. In the code below we choose the first 500 of date values *and* NO_x. The result is shown in Figure 4.6. Note also that R automatically adjusts the x-axis labels to day of the month, rather than year as before. Also note some missing data before 4 January. This is the advantage of keeping all the data and not removing missing values. If we had removed the values, the gap in data would not have been shown and there would have been a discontinuity. For more information on selecting parts of a data frame and selecting by date, see (§5.2).

To plot the data over different averaging times requires that the data are summarised in some way. This can seem to get quite complicated — because it is. Here we use a function called `aggregate`, which can summarise data in different ways. The code required to plot monthly means is shown in Figure 4.7.

So what does this code do? The first line is a command that will summarise the data by month and year. Note that we need to remove missing data from the calculations, hence the `na.rm` option. To get monthly maximum values you simply replace the `mean` by `max`. We then generate some dates that are as long as the monthly time series, `means` using the `seq` function. This function generates data the same length as `means`, starting at the beginning of the series `mydata$date[1]` and ending at the end `mydata$date[nrow(mydata)]`. Finally, a plot is produced; in this case without axes properly labelled shown in see Figure 4.7.

Now, let's plot some daily averages. Here the averaging time is given as %j (meaning decimal day of year; see `strptime` and Table 4.1 for more information on this), see Figure 4.8. To plot annual mean use %Y and weekly means %U.

```
plot(mydata$date[1:500], mydata$nox[1:500], type = "l", xlab = "date",
     ylab = "Nitrogen oxides (ppb)")
```

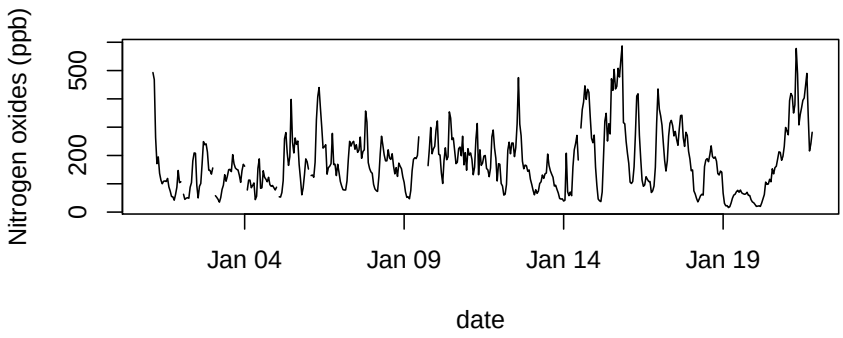


FIGURE 4.6 Hourly time series plot of NO_x at Marylebone Road — first 500 records.

```
# calculate monthly means
means <- aggregate(mydata["nox"], format(mydata["date"],"%Y-%m"),
                   mean, na.rm = TRUE)

# derive the proper sequence of dates
means$date <- seq(min(mydata$date), max(mydata$date), length = nrow(means))

# plot the means
plot(means$date, means[, "nox"], type = "l")
```

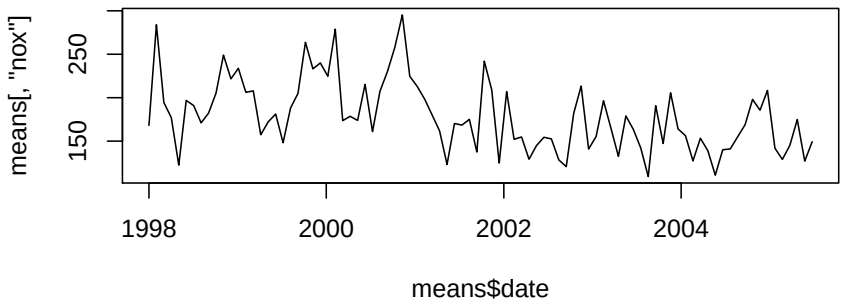


FIGURE 4.7 Monthly time series plot of NO_x at Marylebone Road.

TABLE 4.1 Some commonly used date-time formats useful when averaging data.

Code	function
%Y	annual means
%m	monthly means
%Y-%m	monthly averages for whole time series
%Y-%j	daily averages for whole time series
%Y-%W	weekly averages for whole time series
%w-%H	day of week — hour of day

```
# calculate daily means
means <- aggregate(mydata["nox"], format(mydata["date"],"%Y-%j"),
                   mean, na.rm = TRUE)

# derive the proper sequence of dates
means[ ,"date"] <- seq(min(mydata[ , "date"]), max(mydata[ , "date"]), length = nrow(means))

# plot the means
with(means, plot(date, nox, type = "l"))
```

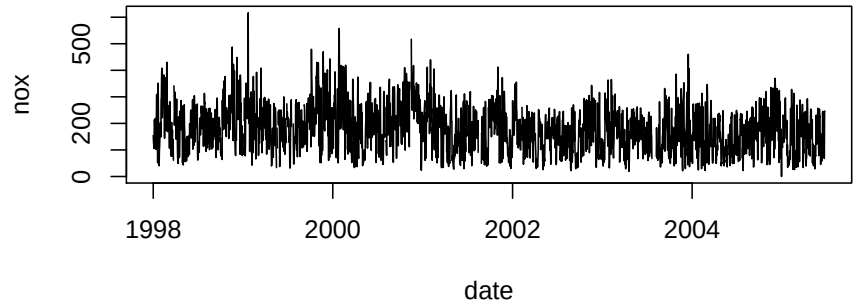


FIGURE 4.8 Daily time series plot of NO_x at Marylebone Road.

The `aggregate` function can work with more than one column of data at a time. So, to calculate monthly means of all values, the following code can be used:

```
means <- aggregate(mydata[-1], format(mydata[1],"%Y-%m"), mean, na.rm = TRUE)
head(means)
```

```
##      date       ws       wd      nox      no2       o3     pm10      so2
## 1 1998-01 5.088775 171.4382 168.0960 42.17421 4.345628 29.18378 5.181171
## 2 1998-02 4.258571 216.7113 283.9152 58.31515 2.486322 40.21184 9.648729
## 3 1998-03 4.665949 224.5726 194.4899 49.71594 5.287966 32.65223 8.975457
## 4 1998-04 4.197833 206.4306 176.9719 47.93108 8.814867 28.86792 6.596971
## 5 1998-05 3.456774 162.5672 122.3760 43.74259 9.458221 32.46289 5.022226
## 6 1998-06 4.931715 219.1389 196.8450 47.13268 5.507353 32.97893 6.298831
##         co     pm25
## 1 1.945224      NaN
## 2 2.904194      NaN
## 3 2.045046      NaN
## 4 1.779968      NaN
## 5 1.254057 20.99145
## 6 2.047912 20.00000
```

In this code the `mydata[-1]` selects all columns in `mydata`, except the first column (which is the date), and `mydata[1]` is the date column i.e. the one we want to average by. Functions of this type can be very useful, allowing quite complex summaries of data to be derived with little effort.

In this case `aggregate` returns the year-month, which is not recognised as a date. Because we are averaging the data, it might be better to represent the data as the middle of each month. We can paste the day onto the end of the year-month column and then convert it to a `Date` object:

```
means$date <- paste(means$date, "-15", sep = "")
means$date <- as.Date(means$date)
```

openair has a flexible functions called `timeAverage` that makes aggregating data

```
# derive the proper sequence of dates
dates <- with(mydata, seq(date[1], date[nrow(mydata)], length = nrow(means)))
plot(dates, means[, "nox"],
     type = "b",
     lwd = 1.5,
     pch = 16,
     col = "darkorange2",
     xlab = "year",
     ylab = "nitrogen oxides (ppb)",
     ylim = c(0, 310),
     main = "Monthly mean nitrogen oxides at Marylebone Road.")
 grid()
```

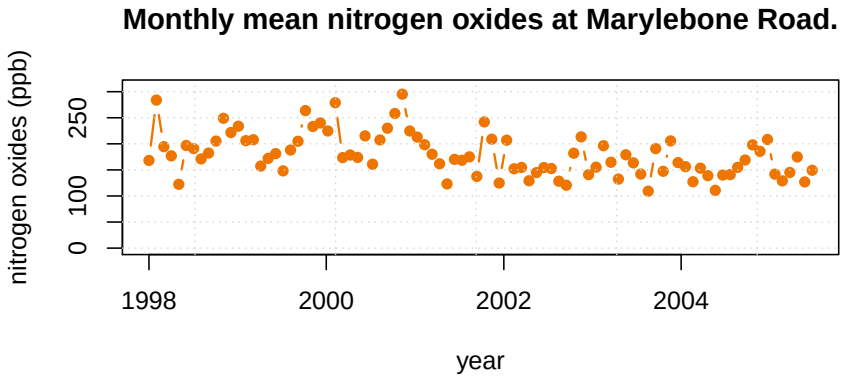


FIGURE 4.9 Monthly mean time series plot of NO_x at Marylebone Road, with enhanced plotting options.

like this much easier so it can be done in a single step for almost any averaging period (see Section 31.4).

4.3.1 Enhancing plots

This seems a good point at which to describe some of the enhancements that can be made to plots. There are numerous ways in which plots can be enhanced and it is worth checking help(plot) to see some of the options available. As an example, we are going to enhance Figure 4.7 using the data `means` previously calculated.

The different enhancements are shown separately on each line of the plotting code. `type = "b"` plots points and lines, which makes it easier to see that months are plotted `lwd = 1.5` makes the line a bit thicker (1 is the default); `pch = 16` is the code for a solid filled circle (other shapes available too); `col = "darkorange2"` makes the plot line and shape a different colour; `ylim = c(0, 310)` sets the lower and upper limits for the y-axis, and finally `grid()` automatically adds grid lines to help navigate the plot. Other enhancements were made by adding a title etc. The result is shown in Figure 4.9.

4.4 A more complicated example — plot construction

So far it has been possible to run only a few lines of code to make the plots. This section considers using many lines of code to design and make a dedicated plot. This example shows one of the strengths of R compared with other plotting software: you are not restricted by 'off the shelf' plots. We work with plotting time series data but want to put a rather different plot together based on the average diurnal profile in

concentration of pollutant by day of the week. Plots of this type are very useful for showing how emissions might change by hour of the day *and* day of the week.

Figure 4.10 shows that the NO_x concentrations are higher during the weekdays and that the weekday variation is very similar on each day. On a Saturday, concentrations tend to peak before midday and rise slowly to midnight. By contrast, concentrations are relatively high in the early hours of the morning (probably due to traffic activity from late night party-goers!) and peaks later in the afternoon. These types of patterns can very often reveal important information about source characteristics — and R allows you to investigate these.

The plot is put together in several parts:

1. First, mean values of NO_x are calculated in the same way as shown previously. In this case the averaging period is `%w-%H`, which means average by day of the week (0–6, 0 = Sunday, 1 = Monday ...) and then by hour of day.

2. A plot is generated of the means. The main effort involved here is to make a decent x-axis with appropriate labels. The plot option `type = "n"` means that no data are actually plotted. This is chosen because we want to add some other features (notably grid lines) first *before* we plot the data. This is just a neater way of doing things: plotting grid lines on top of data looks worse. The option `xaxt = "n"` suppresses the x-axis altogether. This is done because we want to make our own. Finally, we chose some sensible names for the axis and title captions.

3. Next, some tick marks are added using the `axis` function. These are added every 24 hours starting at 1.

4. We then annotate the x-axis with day of the week names. These are placed at the middle of each day e.g. hour 13 for Sunday etc.

5. Some grid lines are added using the function `abline`. In this case, a sequence of vertical lines are added (hence the `v` option). These are chosen to have a light grey colour with the option `col = "grey85"`.

6. Finally, the data themselves are added as a line with the function `lines`, with a colour of `darkorange2` and a line width of 2.

As an alternative to using grid lines to distinguish between different days, shading can be useful too. Here, we shade alternate days with the `rect` function. See help(rect) for all the options available.

4.5 Summarising time series data

There are some nice ways of quickly summarising data over different time scales including day of year, month of year and day of week. The way pollutant concentrations vary over different time scales can provide some useful clues as to what sources are important. Again, this is the sort of thing that can be tricky in other software. One of the most useful; plots available is called a *box and whisker* plot, which is a very effective way of summarising large amounts of data. Say for example, we were interested to see how ozone concentrations vary by month of the year as shown in Figure 4.12.

The results shown in Figure 4.12 show several interesting features. The dark lines shows the median concentration, which peaks in May. However, the highest hourly concentrations are observed in August — presumably due to regional-scale photochemical pollution episodes.

```
# calculate means
means <- aggregate(mydata["nox"], format(mydata["date"],"%w-%H"),
                   mean, na.rm = TRUE)

plot(means$nox, xaxt = "n", type = "n",
     xlab = "day of week",
     ylab = "nitrogen oxides (ppb)",
     main = "Nitrogen oxides at Marylebone Road by day of the week")

# add some tick marks at 24 hr intervals
axis(1, at = seq(1, 169, 24), labels = FALSE)

# add some labels to x-axis
days = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")

loc.days = seq(13, 157, 24) # location of labels on x-axis

# write text in margin
mtext(days, side = 1, line = 1, at = loc.days)

# add some grid lines
abline(v = seq(1, 169, 24), col = "grey85")

# add the line
lines(means$nox, col = "darkorange2", lwd = 2)
```

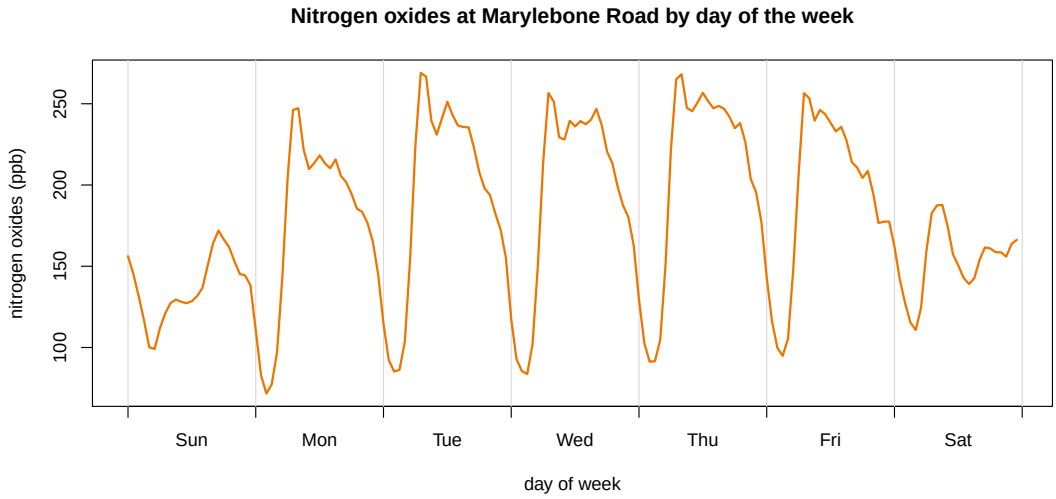


FIGURE 4.10 Day of week and hour of day plot of NO_x at Marylebone Road.

It is also possible to view an entire series of data as monthly means, as shown in Figure 4.13. This plot shows quite nicely that the median and the peak concentrations of NO_2 increased in 2003, which is now known to be due to increased emissions of primary NO_2. Other useful summary functions include day of the week (%A) and year (%Y). There are many possibilities for plotting here and it is suggested you try some of your own.

4.6 Relationships between variables

Exploring how variables are related to one another is a very useful thing to do, but often tricky when you have lots of variables and lots of data. At a basic level, a scatter plot of one variable against another is useful. However, when you have more than a

```
means <- aggregate(mydata["nox"], format(mydata["date"],"%w-%H"), mean,
                   na.rm = TRUE)
plot(means$nox, xaxt = "n", type = "n",
     ylim = c(60, 270),
     xlab = "day of week",
     ylab = "nitrogen oxides (ppb)",
     main = "Nitrogen oxides at Marylebone Road by day of the week")

axis(1, at = seq(1, 169, 24), labels = FALSE)

# add some labels to x-axis
days = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")
loc.days = seq(13, 157, 24) # location of labels on x-axis

# write text in margin
mtext(days, side = 1, line = 1, at = loc.days)
ylow = 60; yhigh = 270 # extent of shading in y direction
xleft = seq(1, 145, 48) # left part of rectangles
xright = xleft + 24 # right part of rectangles

# draw rectangles
rect(xleft, ylow, xright, yhigh, col = "lightcyan", border = "lightcyan")

# addline
lines(means$nox, col = "darkorange2", lwd = 2)
```

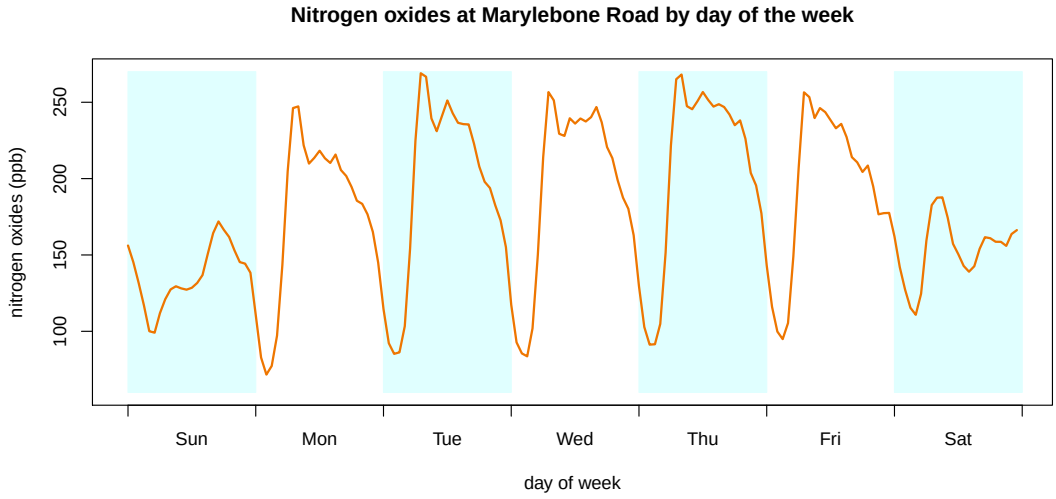


FIGURE 4.11 Day of week and hour of day plot of NO_x at Marylebone Road, with shading.

few variables it becomes quite an effort to manually plot one against another. Luckily, R has some excellent facilities to help you out; in particular the `pairs` function. We could just use the command `pairs(mydata)` and this would plot each variable all other variables. Our data frame now has 11 columns, so this would be 121 plots! Furthermore, each variable is of length 65,533 long — so the plots would look busy to say the least.

We therefore take the opportunity to introduce a new function in R called `sample`. What we want to do is randomly select 500 lines from our data set, which is the function `sample(1:nrow(mydata), 500)`. What this does is randomly select 500 numbers from a sample a long as our data set i.e. `nrow(mydata)`. We also want to limit the columns chosen to plot. We will plot `date, ws, wd, nox, no2`; which correspond to columns 11, 2, 3, 4 and 5. You can of course choose others, or more or less than 500 points. The code below is one function but is spread over several lines for clarity. The

```
plot(as.factor(format(mydata$date, "%m")), mydata$o3)
```

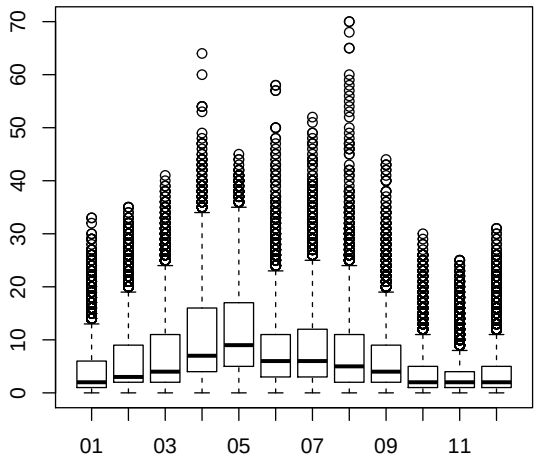


FIGURE 4.12 Monthly box and whisker plot of O_3 at Marylebone Road.

```
plot(as.factor(format(mydata$date, "%Y-%m")), mydata$no2, col = "lightpink")
```

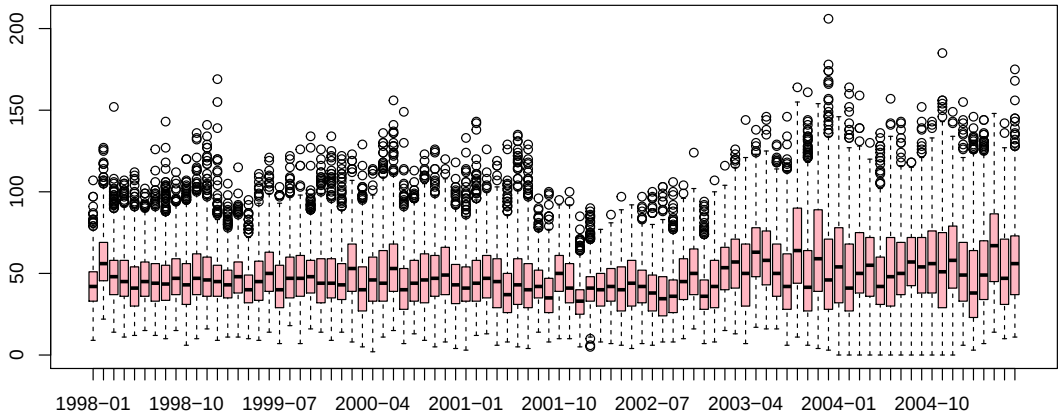


FIGURE 4.13 Yearly-monthly box and whisker plot of NO_2 at Marylebone Road.

plot is shown in Figure 4.14.

So what does this plot show? Well, if you look at the first column (date), the plots essentially show the trend in the different variables i.e. no change in wind speed, slight dip in wind direction, a decrease in NO_x in 2001 and an increase in NO_2 from around 2003. They also show that concentrations of NO_x and NO_2 do not change much with wind speed (street canyon effects) and that the highest concentrations are recorded when the wind is westerly. These plots can be extremely useful and are worth exploring in different ways.

```
pairs(mydata[sample(1:nrow(mydata), 500), c(1, 2, 3, 4, 5)],
      lower.panel = panel.smooth,
      upper.panel = NULL,
      col = "skyblue3")
```

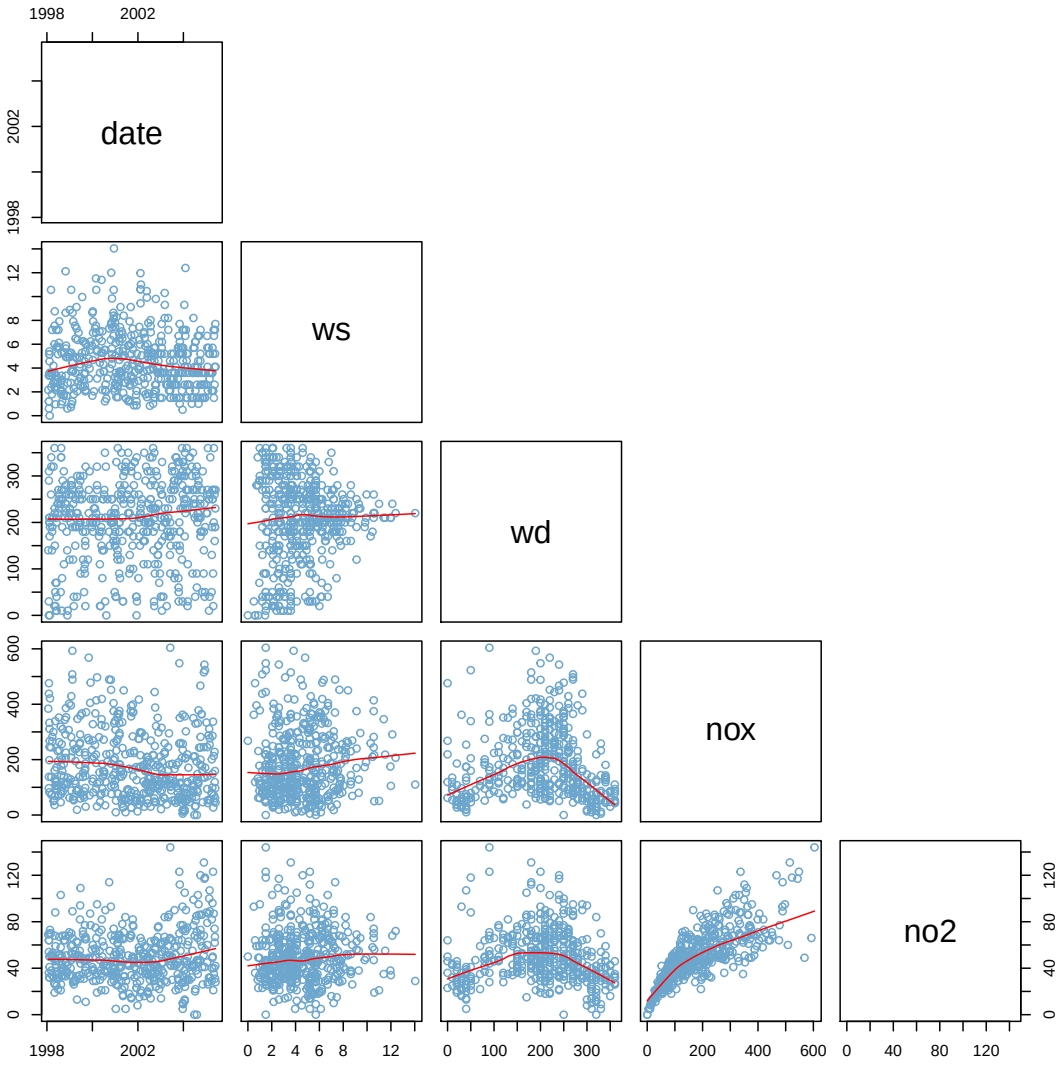


FIGURE 4.14 Pairs plot for 500 randomly selected hours of data from Marylebone Road.

5 General use of R — advice and examples

5.1 Data input and output

Getting data into and out of R is an obvious requirement. There are many options for doing so, but we only cover some of the common methods in this section.

5.1.1 Data import

Sometimes it is useful to input just a few bits of data where it is unnecessary to import from a file. This is simply done using the `c` function:

```
small <- c(1, 5, 10, 16)
small
```

```
## [1]  1  5 10 16
```

More commonly it is necessary to import data from a file. As has been discussed before it is best to keep the format of such files as simple as possible e.g. coma-delimited (.csv) or text (.txt). You can use R to list files in a particular directory. Usually a user would have set a 'working directory', and all file commands will relate to that (in the Windows version of R, go to File menu and choose Change Dir ...It is also possible to set working directories with a command e.g.

```
setwd("~/openair/Documentation")
```

Or if you want to know what the current working directory is:

```
getwd()
```

```
## [1] "/Users/davidcarslaw/openair/Documentation"
```

Rather than using an external program to list files, you can do that in R too (in this case using the 'pattern' option to list only csv files:

```
list.files(pattern = ".csv")
```

```
## [1] "example data long.csv" "f-no2 input.csv"       "hour-day.csv"
```

Now we have a working directory set it is possible just to refer to the file name and not the path. So, to import the file 'hour-day.csv' we can:

```
hour.day <- read.csv("hour-day.csv", header = FALSE)
head(hour.day)
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 V21
## 1 15 27 32 17 21 11 15 32  29  21  25  32  42  34  31  44  63  71  59  34  29
## 2 31 19 25 25 21 32 38 78 109 128 105  88  86  86  94  97  NA 176 208 126  86
## 3 59 31 29 29 31 34 55 65 130 134 113 107 117 115 126 113 103  88  92  50  31
## 4 15 13 10 17 19 31 44 53 126 128  96  71 103  82  90  94 120  97 139  94  74
## 5 53 31 25 31 21 46 53 82 162 174 149 136 113 126 120 157 201 220 174 149  92
## 6 73 52 42 42 29 34 52 59  97 122 204 269 323 285 248 193 264 254 241 183 124
##   V22 V23 V24
## 1  29  25  36
## 2 111  92 113
## 3  36  21  27
## 4  67  65  59
## 5 109 134  86
## 6  97  92  55
```

In this case the data did not have a header line and we told R not to expect one. R automatically labels the fields V1, V2 ...

If the file was tab-delimited then the `read.table` function should be used (`read.csv` is a version of `read.table`).

One potentially useful capability is the direct reading of data over the internet. For example, we have example data on a web server, and this can be read directly just like a file on a computer file system:

```
test.data <- read.csv("http://www.openair-project.org/CSV/OpenAir_example_data_long.csv",
                      header = TRUE)
```

This capability might be useful for making data available for anyone to access easily.

5.1.2 Data export

Exporting data can also take a number of forms. A common issues is exporting a data frame(s) as a csv file.

```
write.csv("exportedData.csv", row.names = FALSE)
```

To save data in an 'R format' (.RData file), it is necessary to use the `save` function. If, for example there was a data frame called 'test.data', this can be save directly:

```
save(test.data, file = "testData.RData")
```

in fact several objects can be saved at one in this way:

```
save(test.data, more.data, file = "testData.RData")
```

To load this data back into R, use the `load` function:

```
load("testData.RData")
```

5.2 Selecting and replacing parts of vectors and data frames

Selecting parts of a data frame is one of the more useful things that one can learn. This often causes new users lots of difficulties, in part because of the way variables are stored in R. Earlier it was seen that in the data frame `mydata`, one could refer to all `nox` values simply as `mydata$nox`. What if one just wanted to select parts of this large data frame? Here are some examples of the sorts of things you might want to do, and in each case we read data into a new data frame called `subdata`[5]. We first consider various ways of selecting parts of vectors.

Consider the vector *x* defined as integers 1, 4, 5, 18, 22, 3, 10, 33, −2, 0.

```
x = c(1, 4, 5, 18, 22, 3, 10, 33, -2, 0)
```

To select the 4th element, we use the square brackets [] to subsample:

```
x[4]
```

```
## [1] 18
```

To select the 3rd to 6th integers:

```
x[3:6]
```

```
## [1]  5 18 22  3
```

To select everything except the first and second value, elements can be *omitted* using the − sign.

```
x[c(-1, -2)]
```

```
## [1]  5 18 22  3 10 33 -2  0
```

[5] You might not always wish to make a new data frame because it will take up extra memory – many of the examples shown can be done 'on the fly'.

Values greater than 5:

```
x[x > 5]
```

```
## [1] 18 22 10 33
```

The indexes corresponding to integers > 5 can be found using the `which` command. This basically finds the location of numbers in a vector that meet a certain criterion. In this example, the fourth element is 18. This is a very useful function for subsetting.

```
which(x > 5)
```

```
## [1] 4 5 7 8
```

To select a specific value it is necessary to use the double = sign i.e.

```
x[x == 18]
```

```
## [1] 18
```

It is also easy to *reverse* a sequence of numbers, which is useful on many occasions:

```
rev(x)
```

```
##  [1]  0 -2 33 10  3 22 18  5  4  1
```

The next thing to do is consider how to replace parts of a vector. This is something that is often necessary when preparing data. To replace the −2 by 0:

```
x[x == -2] = 0
x
```

```
##  [1]  1  4  5 18 22  3 10 33  0  0
```

Note that all the usual operators such as > can be used here.

Next we are going to select different parts of the data frame `mydata`. This can be more complicated because the data comprise both rows and columns. Select the first 500 rows of data. This selects rows 1 to 500 and the blank space after the comma means 'select all columns' i.e. all variables:

```
subdata = mydata[1:500, ]
```

One can check the number of rows selected:

```
nrow(subdata)
```

```
## [1] 500
```

Select a few variables from a data frame. Here the function `subset` is easy to use. We select just the `nox` and `no2` data. Note that when using this command, one does not need to use the $ operator, which makes selecting a bit easier to see.

the `subset` function is very useful in R

```
subdata = subset(mydata, select = c(nox, no2))
```

If one wanted to select all `nox`, `no2` and `date` values where `nox` concentrations were greater than 600 ppb:

```
subdata = subset(mydata, nox > 600, select = c(nox, no2, date))
```

Selecting by date is very useful but a bit more complicated. However, once learnt it is extremely flexible and useful. We want to select all `nox`, `no2` and `date` values for 2004, although any start/end time can be used. We start by defining a start and end date, then carry out the actual selection. In this example, we must first convert our date (which is in character format) into a date/time format that R understands. Note that dates/times in R conventionally work in a hierarchical way (biggest to smallest component). Therefore '2004-02-03 00:00' is the 3rd of February and not the 2nd March. In most cases dates would have been read in and converted appropriately anyway, but in this particular case we need to specify a particular date. The conversion from character string to a recognised date/time in R is done using the `as.POSIXct` function. This may seem complicated, but once learnt is both convenient and powerful. The **openair** package makes this much easier — see (§31.1) for more details.

```
start.date <- as.POSIXct("2004-01-01 00:00", tz = "GMT")
end.date <- as.POSIXct("2004-12-31 23:00", tz = "GMT")
subdata <- subset(mydata, date >= start.date & date <= end.date,
                  select = c(date, nox, no2))
```

View the first or last few lines of a data frame

One can easily check what this subset looks like the the functions `head` and `tail`, which give the first and last few lines of a data frame:

```
head(subdata)
```

```
##                     date nox no2
## 52585 2004-01-01 00:00:00  98  38
## 52586 2004-01-01 01:00:00 141  62
## 52587 2004-01-01 02:00:00 159  56
## 52588 2004-01-01 03:00:00  97  44
## 52589 2004-01-01 04:00:00  60  26
## 52590 2004-01-01 05:00:00  64  31
```

and,

```
tail(subdata)
```

```
##                     date nox no2
## 61363 2004-12-31 18:00:00 114  53
## 61364 2004-12-31 19:00:00 183  67
## 61365 2004-12-31 20:00:00 206  69
## 61366 2004-12-31 21:00:00 237  73
## 61367 2004-12-31 22:00:00 232  68
## 61368 2004-12-31 23:00:00 212  68
```

selecting different time periods

Another useful way of selecting subsets is using the `%in%` (or `match`) function. Some examples are given below with dates.

```
subdata <- subset(mydata, format(date, "%Y") %in% 1998)

# select 1998 and 2005
subdata <- subset(mydata, format(date, "%Y") %in% c(1998, 2005))

# select weekends
subdata <- subset(mydata, format(date, "%A") %in% c("Saturday", "Sunday"))
```

This function is very useful for selecting subsets of data where there are multiple search criteria. For example, if a data frame had a field such as site name and the aim was to select data from several sites, this would be a good way to do it.

selecting columns based on characters in them

It is sometimes useful to select columns (or rows) of a data frame based on their

names. One extremely powerful command in R is **grep**. **grep** does character matching. It is potentially useful in numerous circumstances, but we only consider a simple case here. Say, for example we had a very large data frame with 50 column names, but we only want to extract those with the characters 'nox' in. We could search through and find those columns by number and refer to them in that way — but that requires a lot of manual work and has lots of potential to go wrong. In the example below we create a simple dummy data frame as an example.

```
test.dat <- data.frame(lond.nox = 1, lond.no2 = 3, nox.back = 4, no2.back = 1)
test.dat

##   lond.nox lond.no2 nox.back no2.back
## 1        1        3        4        1
```

First, for information we can print the names of the data frame:

```
names(test.dat)

## [1] "lond.nox" "lond.no2" "nox.back" "no2.back"
```

To find those names that contain the character string 'nox' we use **grep**:

```
grep(pattern = "nox", names(test.dat))

## [1] 1 3
```

So, columns 1 and 3 contain the character string 'nox'. We can put this altogether and do it in one line to select those columns in the data frame that contain 'nox':

```
sub.dat <- test.dat[ , grep(pattern = "nox", names(test.dat))]
sub.dat

##   lond.nox nox.back
## 1        1        4
```

The **grep** command is potentially useful for selecting pollutants to plot in **openair** plots e.g. to choose any column with 'pm' (PM_{10} and $PM_{2.5}$) in it:

```
timePlot(mydata, pollutant = names(mydata)[grep(pattern = "pm", names(mydata))])
```

5.3 Combining and cleaning up files

So far the emphasis has been on manually importing a single .csv file to a data frame. Often with monitoring data there are numerous files all in the same format that somehow need to be read and merged. R has some very powerful and convenient ways of dealing with this situation and only the simplest case is shown here. The scenario is that you have loads of .csv files in a directory, all the same headings (although not necessarily so) and the aim is to read and combine them all. This can be done using the code below.

reading in lots of files

```
path.files <- "D:\\temp\\" # directory containing files
test.data <- lapply(list.files(path = path.files, pattern = ".csv"),
                    function(.file) read.csv(paste(path.files, .file, sep = ""),
                                             header = TRUE))
test.data <- do.call(rbind, test.data)
```

There are a few things to note here. In R for Windows, file paths are shown using

'\\'. The function `list.files` will search for files (in this case .csv) in the D:\Temp. In the code above it is assumed a header is also present. For more refined searching see help(list.files). The `lapply` function is extremely useful in R and can help avoid looping through data. In this case the function `function(.file)` is applied to the list of file names/paths supplied by `list.files`. This is a neat way of applying a function without knowing beforehand how many files there are. The traditional way of doing it would be to have a loop such as `for i = 1 to n` where `n` would be the number of files.

Note, different numbers of columns can also be dealt with using the `rbind.fill` function from the `reshape2` package as described below. In this case, the `do.call(rbind, test.data)` would be modified to `do.call(rbind.fill, test.data)`.

A common task is combining different files into one for processing. First we consider the scenario of a file with air pollution measurements and another with meteorological data. The aim is to combine them into one data frame. Rather than import data, we generate it instead. The first is a data frame called `airpol` with 1 day of data at the beginning of 2007 with pollutants NO_x and SO_2. The other is a meteorological data set, with the same dates but with wind speed and direction.

combine two data frames

```
airpol <- data.frame(date = seq(as.POSIXct("2007-01-01"),
                     by = "hours", length = 24), nox = 1:24, so2 = 1:24)
met <- data.frame(date = seq(as.POSIXct("2007-01-01"),
                  by = "hours", length = 24), ws = rep(1, 24), wd = rep(270, 24))
```

You can check the contents of these data frames:

```
head(airpol)
```

```
##                  date nox so2
## 1 2007-01-01 00:00:00   1   1
## 2 2007-01-01 01:00:00   2   2
## 3 2007-01-01 02:00:00   3   3
## 4 2007-01-01 03:00:00   4   4
## 5 2007-01-01 04:00:00   5   5
## 6 2007-01-01 05:00:00   6   6
```

```
head(met)
```

```
##                  date ws  wd
## 1 2007-01-01 00:00:00  1 270
## 2 2007-01-01 01:00:00  1 270
## 3 2007-01-01 02:00:00  1 270
## 4 2007-01-01 03:00:00  1 270
## 5 2007-01-01 04:00:00  1 270
## 6 2007-01-01 05:00:00  1 270
```

To combine them, use the `merge` function:

```
test.data <- merge(airpol, met)
head(test.data)
```

```
##                  date nox so2 ws  wd
## 1 2007-01-01 00:00:00   1   1  1 270
## 2 2007-01-01 01:00:00   2   2  1 270
## 3 2007-01-01 02:00:00   3   3  1 270
## 4 2007-01-01 03:00:00   4   4  1 270
## 5 2007-01-01 04:00:00   5   5  1 270
## 6 2007-01-01 05:00:00   6   6  1 270
```

When called like this the `merge` function combines data frames only where both

had data. So, for example, if the `met` data frame only had the first 12 hours of 2007, merging would produce a file with only 12 hours i.e. where they match (a *natural join* in database terminology). The behaviour can be changed by selecting various options in `merge`. Following on from the previous example, the option `all` could have been set to `TRUE`, thus ensuring *all* records from each data frame would be combined—with the missing 12 hours in the `met` data frame included as `NA`. Type `help(merge)` to see the details. Functions of this type can save lots of time aligning various time series in spreadsheets.

Note, that given a data frame with multiple air pollution sites and a column called 'site' (i.e. values for the field 'date' are repeated the same number of times there are numbers of sites) it is easy to merge a *single* meteorological data set. This is the type of analysis where several air quality sites in a region are associated with a single meteorological data set. Given a data frame `aq` with multiple sites in a format like 'date', 'nox', 'site' and a meteorological data set `met` in the form something like 'date', 'ws', 'wd' then the merging is done by:

```
all.data <- merge(aq, met, by = "date", all = TRUE)
```

This code ensures that for each site for a particular date/time there are associated meteorological values. In other words, it is not necessary to think about separately joining meteorological and air quality data for each individual air quality site. See (§9) for scenarios where doing this may be useful, such as importing data for multiple AURN sites from the UK air quality archive.

Sometimes it is necessary to combine data frames that have the *same* field names. For example, data from two monitoring sites that measure the same pollutants. In the example below, we make two copies of the data frame `airpol` and name them `site1` and `site2`, respectively. Normally, of course, the data frames would contain different data, perhaps spanning different time periods. A new data frame is made using the `merge` function but with additional options set. Now, we explicitly state that we want to merge on the date field (`by = "date"`). In order to tell the NO_x and SO_2 fields apart, suffixes are used. The resulting data frame has now been merged and the NO_x from `site1` is called `nox.st1` etc.

```
site1 <- airpol
site2 <- airpol

both <- merge(site1, site2, by = "date", suffixes = c(".st1", ".st2"), all = TRUE)

head(both)
```

```
##                  date nox.st1 so2.st1 nox.st2 so2.st2
## 1 2007-01-01 00:00:00       1       1       1       1
## 2 2007-01-01 01:00:00       2       2       2       2
## 3 2007-01-01 02:00:00       3       3       3       3
## 4 2007-01-01 03:00:00       4       4       4       4
## 5 2007-01-01 04:00:00       5       5       5       5
## 6 2007-01-01 05:00:00       6       6       6       6
```

A problem that is often encountered is combining files for different years, perhaps with different numbers of columns. We consider the slightly more difficult latter situation; although the former one is tackled in the same straightforward way. This situation can arise frequently with monitoring data. For example, in year 1, two pollutants are measured (say NO_x and NO_2), then in year 2 another pollutant is added as the monitoring is expanded. In year 2 data are available for NO_x, NO_2 and PM_{10}. This is a straightforward enough problem to deal with but can be surprisingly frustrating

and time consuming to do in spreadsheets (particularly if the column order changes). However, help is at hand with the **merge** function. Given the situation mentioned, **merge** will deal with this:

```
# make some data
year1 <- data.frame(date = seq(as.POSIXct("2007-01-01"),
                    by = "hours", length = 24), nox = 1:24, so2 = 1:24)
year2 <- data.frame(date = seq(as.POSIXct("2008-01-01"),
                    by = "hours", length = 24), nox = 1:24, so2 = 1:24, pm10 = 1:24)
test.data <- merge(year1, year2, all = TRUE)
head(test.data)
```

```
##                  date nox so2 pm10
## 1 2007-01-01 00:00:00   1   1   NA
## 2 2007-01-01 01:00:00   2   2   NA
## 3 2007-01-01 02:00:00   3   3   NA
## 4 2007-01-01 03:00:00   4   4   NA
## 5 2007-01-01 04:00:00   5   5   NA
## 6 2007-01-01 05:00:00   6   6   NA
```

In this example, **year1** contains hourly data for all of 2007 for NO_x and NO_2, and **year2** contains hourly data for all of 2008 for NO_x, NO_2 and PM_{10}. The data frame **test.data** then contains two years of data and has all variables present. For year 1 where there are no PM_{10} data, these data are shown as missing i.e. **NA**.

Another useful application of the **merge** function is to fill in gaps due to missing data. The scenario is that you have a file (say a year long of hourly data), but some lines are missing. This sort of situation arises frequently and can be time consuming to sort out. What is needed is to 'pad out' the file and fill in the gaps with the missing dates and set the other fields to missing (**NA**, in R-speak). To show this, we first deliberately remove 2 of the hours from the **airpol** data frame. We then create a data frame with all the hours (note that only 24 are used here, but it can of course be any length), then the data are merged:

padding-out missing hours

```
airpol <- airpol[-c(2, 3), ] # select everything except record 2 and 3
# create all the dates that should exist
all.dates = data.frame(date = seq(as.POSIXct("2007-01-01"),
                       by = "hours", length = 24))
# merge the two
test.data <- merge(all.dates, airpol, all = TRUE)
head(test.data)
```

```
##                  date nox so2
## 1 2007-01-01 00:00:00   1   1
## 2 2007-01-01 01:00:00  NA  NA
## 3 2007-01-01 02:00:00  NA  NA
## 4 2007-01-01 03:00:00   4   4
## 5 2007-01-01 04:00:00   5   5
## 6 2007-01-01 05:00:00   6   6
```

The missing hours are thus inserted, but the variables themselves are set to missing.

Finally (and not surprisingly) a package already exists that does this for you. The **reshape2** package can manipulate data in very flexible ways. However, the function **rbind.fill** is particularly useful if you have lots of different data frames to combine because **merge** can only merge two data frames at once. Note you can download this package from CRAN.

```
library(reshape2)
test.data <- rbind.fill(year1, year2)
```

interpolate missing data

Sometimes it is useful to *fill in missing data* rather than ignore it. Here, we show two options from the `zoo` (zero-ordered observations) package. The first function `na.locf` will fill missing data with the value of the last non-missing data point. To do a linear interpolation between points, the `na.locf` function should be used:

```
library(zoo)

##
## Attaching package: 'zoo'
##
## The following objects are masked from 'package:base':
##
##    as.Date, as.Date.numeric

# make some data with missing values
a = c(1, NA, NA, NA, 3, NA, 18, NA, NA, 20)
# show data
a

##  [1]  1 NA NA NA  3 NA 18 NA NA 20

# fill with last non-missing point
na.locf(a)

##  [1]  1  1  1  1  3  3 18 18 18 20

# interpolate missing points
na.approx(a)

##  [1]  1.00000  1.50000  2.00000  2.50000  3.00000 10.50000 18.00000 18.66667
##  [9] 19.33333 20.00000
```

There are various other options that can be used with these functions, which can be considered by typing help(zoo). These functions can also be applied to data frames (or columns of). Say we want to interpolate all the missing NO_x concentrations:

```
mydata$nox <- na.approx(mydata$nox, na.rm = FALSE)
```

Note that the `na.rm = FALSE` option ensures that trailing NAs are not removed, making the number of records the same as the original data. The code above would replace the NO_x concentrations. If preferred, a new column could be made, in this case called `nox.all`:

```
mydata$nox.all <- na.approx(mydata$nox, na.rm = FALSE)
```

Once data are imported into R — say by loading a .csv file into a data frame, there are often tasks that need to be carried out to alter the data before processing it. Some of these common tasks are considered in this section.

One of the most immediate tasks with air pollution data is to convert the date/time field into something understood by R and this has already been discussed.

changing variable names

Next, it may be necessary to change the name of a variable. To list the existing variables names:

```
names(mydata)

##  [1] "date"    "ws"      "wd"      "nox"     "no2"     "o3"      "pm10"
##  [8] "so2"     "co"      "pm25"    "nox.all"
```

You can also refer to a single column name. In the code below, we show an example of how to change one of the names (in this case `nox`) to `nitrogen.oxides`.

```
names(mydata)[4] # display name of 4th column (nox)

## [1] "nox"

names(mydata)[4] = "nitrogen.oxides" # change the name
names(mydata) # show new names

##  [1] "date"           "ws"             "wd"             "nitrogen.oxides"
##  [5] "no2"            "o3"             "pm10"           "so2"
##  [9] "co"             "pm25"           "nox.all"

# change it back again
names(mydata)[4] = "nox"
```

To change more than one name at a time (say the 4th and 5th column names):

```
names(mydata)[c(4, 5)] = c("new1", "new2")
```

If you have imported data that has lots of upper case names and you want them all in lower (because they are easier to refer to), use the `tolower` function e.g.

```
names <- c("NOx", "PM10") #make some upper case names
tolower(names)

## [1] "nox"  "pm10"
```

If you import data that has rather verbose descriptions, which become a pain to refer to, you can abbreviate them. In this example, we have two site names and the aim is to abbreviate them using only two letters.

```
names = c("North Kensington", "Marylebone Road")
abbreviate(names, 2)

## North Kensington  Marylebone Road
##             "NK"             "MR"
```

There is potential for this going wrong, if, for example two of the names were very similar:

```
names <- c("North Kensington roadside", "North Kensington background",
           "Marylebone Road")
abbreviate(names, 2)

##   North Kensington roadside North Kensington background
##                        "NKr"                        "NKb"
##             Marylebone Road
##                        "MR"
```

However, R is clever enough to work this out, and uses an extra letter as necessary. The `abbreviate` function can be very effective at simplifying files for processing and generally makes logical simplifications. Note that in the examples above, one could have chosen to abbreviate the names to any length.

Data can easily be ordered and this might be necessary if for example, the date field was not sequential in time. An example is:

```
mydata <- mydata[order(mydata$date), ]
```

which keeps data in the same data frame `mydata` but ensures that *all* the data are ordered by date.

5.4 Reshaping data

Data are stored in a wide variety of ways and it is often necessary to do some data manipulation in order to analyse or plot data. This section distinguishes between two main storage options: stacked or column format (narrow or wide). By way of an example, consider the simple case of two sites each measuring NO_x. One way of storing all this data in a single data frame would be to have columns: 'date', 'site1.nox', 'site2.nox'. An alternative would be to stack the data and have columns 'date', 'nox', 'site'. For such a simple example there isn't much difference between the two options. But what if there were 10, 20 or 100 sites? Having columns 'site1.nox', 'site2.nox' ...would get rather tedious, whereas the stacked data would still only have three columns.

For **openair** functions there is a *big* advantage in stacking data like this, and all the **openair** import functions do this. This is because it then becomes easy to plot any number of quantities **without referring to them individually and without knowing how many there are**. This will become clearer as **openair** functions are used, but imagine trying to plot NO_x at 10 sites using the two different approaches using the **openair** `timePlot` function:

For column format:

```
timePlot(mydata, pollutant = c("site1.nox", "site2.nox", "site3.nox", ...,
                               "site10.nox"))
```

And stacked data:

```
timePlot(mydata, pollutant = "nox", type = "site")
```

The latter example works for any number of sites without having to know the number.

So how can data be re-shaped to get it into the appropriate format? This is best answered with an example using the `reshape2` package that is loaded with **openair**. We'll work with the first 3 lines of `mydata`.

```
## select first 3 lines
thedata <- head(mydata, 3)
thedata
```

```
##                  date   ws  wd      nox no2 o3 pm10    so2     co pm25  nox.all
## 1 1998-01-01 00:00:00 0.60 280 285.0000  39  1   29 4.7225 3.3725   NA 285.0000
## 2 1998-01-01 01:00:00 2.16 230 354.3333  NA NA   37     NA     NA   NA 354.3333
## 3 1998-01-01 02:00:00 2.76 190 423.6667  NA  3   34 6.8300 9.6025   NA 423.6667
```

The `reshape2` package comes with two main functions `melt` and `dcast`. `melt` organises data according to 'measured' and 'id' variables. In our example the measured values are the pollutants and id is the date. It is possible to list either the measured or id values, but in this case it is easier with id because there is only one:

```
library(reshape2)
library(plyr)
thedata <- melt(thedata, id.vars = "date")
thedata
```

```
##                   date variable    value
## 1  1998-01-01 00:00:00       ws   0.6000
## 2  1998-01-01 01:00:00       ws   2.1600
## 3  1998-01-01 02:00:00       ws   2.7600
## 4  1998-01-01 00:00:00       wd 280.0000
## 5  1998-01-01 01:00:00       wd 230.0000
## 6  1998-01-01 02:00:00       wd 190.0000
## 7  1998-01-01 00:00:00      nox 285.0000
## 8  1998-01-01 01:00:00      nox 354.3333
## 9  1998-01-01 02:00:00      nox 423.6667
## 10 1998-01-01 00:00:00      no2  39.0000
## 11 1998-01-01 01:00:00      no2       NA
## 12 1998-01-01 02:00:00      no2       NA
## 13 1998-01-01 00:00:00       o3   1.0000
## 14 1998-01-01 01:00:00       o3       NA
## 15 1998-01-01 02:00:00       o3   3.0000
## 16 1998-01-01 00:00:00     pm10  29.0000
## 17 1998-01-01 01:00:00     pm10  37.0000
## 18 1998-01-01 02:00:00     pm10  34.0000
## 19 1998-01-01 00:00:00      so2   4.7225
## 20 1998-01-01 01:00:00      so2       NA
## 21 1998-01-01 02:00:00      so2   6.8300
## 22 1998-01-01 00:00:00       co   3.3725
## 23 1998-01-01 01:00:00       co       NA
## 24 1998-01-01 02:00:00       co   9.6025
## 25 1998-01-01 00:00:00     pm25       NA
## 26 1998-01-01 01:00:00     pm25       NA
## 27 1998-01-01 02:00:00     pm25       NA
## 28 1998-01-01 00:00:00  nox.all 285.0000
## 29 1998-01-01 01:00:00  nox.all 354.3333
## 30 1998-01-01 02:00:00  nox.all 423.6667
```

which makes two columns: ‘variable’ (pollutant name) and ‘value’.
It is possible to go from this ‘long’ format back to wide:

```
thedata <- dcast(thedata, ... ~ variable)
```

Anything to the right of ~ will make new columns for each unique value of ‘variable’.
Imagine now we have data from two sites that is stacked (first we’ll make some):

```
site1 <- thedata
## add column with site name
site1$site <- "site1"
site1
```

```
##                  date   ws  wd      nox no2 o3 pm10    so2     co pm25  nox.all
## 1 1998-01-01 00:00:00 0.60 280 285.0000  39  1   29 4.7225 3.3725   NA 285.0000
## 2 1998-01-01 01:00:00 2.16 230 354.3333  NA NA   37     NA     NA   NA 354.3333
## 3 1998-01-01 02:00:00 2.76 190 423.6667  NA  3   34 6.8300 9.6025   NA 423.6667
##    site
## 1 site1
## 2 site1
## 3 site1
```

```
site2 <- thedata
site2$site <- "site2"
site2
```

```
##                  date   ws  wd      nox no2 o3 pm10    so2     co pm25  nox.all
## 1 1998-01-01 00:00:00 0.60 280 285.0000  39  1   29 4.7225 3.3725   NA 285.0000
## 2 1998-01-01 01:00:00 2.16 230 354.3333  NA NA   37     NA     NA   NA 354.3333
## 3 1998-01-01 02:00:00 2.76 190 423.6667  NA  3   34 6.8300 9.6025   NA 423.6667
##    site
## 1 site2
## 2 site2
## 3 site2
```

```
## combine all the data
alldata <- rbind.fill(site1, site2)
alldata
```

```
##                  date   ws  wd      nox no2 o3 pm10    so2     co pm25  nox.all
## 1 1998-01-01 00:00:00 0.60 280 285.0000  39  1   29 4.7225 3.3725   NA 285.0000
## 2 1998-01-01 01:00:00 2.16 230 354.3333  NA NA   37     NA     NA   NA 354.3333
## 3 1998-01-01 02:00:00 2.76 190 423.6667  NA  3   34 6.8300 9.6025   NA 423.6667
## 4 1998-01-01 00:00:00 0.60 280 285.0000  39  1   29 4.7225 3.3725   NA 285.0000
## 5 1998-01-01 01:00:00 2.16 230 354.3333  NA NA   37     NA     NA   NA 354.3333
## 6 1998-01-01 02:00:00 2.76 190 423.6667  NA  3   34 6.8300 9.6025   NA 423.6667
##    site
## 1 site1
## 2 site1
## 3 site1
## 4 site2
## 5 site2
## 6 site2
```

Now we have data that is stacked — but how do we get it into column format?

```
## this time date AND site are the id variables
library(reshape2)
alldata <- melt(alldata, id.vars = c("site", "date"))
## want unique combinations of site AND variable
alldata <- dcast(alldata, ... ~ site + variable)
alldata
```

```
##                  date site1_ws site1_wd site1_nox site1_no2 site1_o3 site1_pm10
## 1 1998-01-01 00:00:00     0.60      280  285.0000        39        1         29
## 2 1998-01-01 01:00:00     2.16      230  354.3333        NA       NA         37
## 3 1998-01-01 02:00:00     2.76      190  423.6667        NA        3         34
##   site1_so2 site1_co site1_pm25 site1_nox.all site2_ws site2_wd site2_nox
## 1    4.7225   3.3725         NA      285.0000     0.60      280  285.0000
## 2        NA       NA         NA      354.3333     2.16      230  354.3333
## 3    6.8300   9.6025         NA      423.6667     2.76      190  423.6667
##   site2_no2 site2_o3 site2_pm10 site2_so2 site2_co site2_pm25 site2_nox.all
## 1        39        1         29    4.7225   3.3725         NA      285.0000
## 2        NA       NA         37        NA       NA         NA      354.3333
## 3        NA        3         34    6.8300   9.6025         NA      423.6667
```

These functions are very useful for getting data into the right shape for analysis.

5.5 Example: converting hour-day data to column format

In this example we deal with a common problem in manipulating data – reformatting data in one format to another. Often data are stored as rows representing days and columns representing hours. This is often a format used by AEA for AURN data. However, this example highlights a more general requirement to reformat data.

The aim is to convert the 24 × 365 'matrix' of data to a single column of data. An example file is provided called 'hour-day.csv'. To make it simple the file only contains hourly data with no column headings (hour of day) or row names (days). Fortunately there are some built-in functions available in R that make the reformatting of these data easy:

```
nox <- read.csv("~/openair/Data/hour-day.csv", header = FALSE)
nox <- as.data.frame(t(nox))
nox <- stack(nox)
nox <- nox$values
```

In the code above, the data are first read in — note the option `header = FALSE` in this case. Next, the data are transposed using the `t` (transpose) function, which produces a matrix of data and transposes the rows/columns. Transposing the data ensures that the hours are now in columns. Note that this operation instead of representing hours in rows, puts them into 365 columns. The columns can now be stacked on top of each other. We convert the matrix back to a data frame with the `as.data.frame` function. Next we use the `stack` function that literally *stacks* columns of data, working from column 1 to column 365. Finally, we extract the values resulting from applying the `stack` function. This sort of data manipulation is straightforward in R but would be much trickier in Excel. The code can actually be written in two lines, but becomes less easy to understand:

```
nox <- read.csv("~/openair/Data/hour-day.csv", header = FALSE)
nox <- stack(as.data.frame(t(nox)))[, "values"]
```

Even if you don't understand this code, this example should provide sufficient information on how to apply it.

In fact, in this case, there is an easier way to do this:

```
nox <- read.csv("~/openair/Data/hour-day.csv", header = FALSE)
nox <- t(nox)
nox <- as.vector(nox)
```

In this code, the data are transformed as before, producing a matrix, and the matrix is converted to a vector. When converting a matrix to a vector, it works on columns rather than by rows. The `stack` function is therefore a better choice if we were interested in groups of data for further processing, as in the example below:

```
test.data = data.frame(grp1 = 1:3, grp2 = 10:12, grp3 = 20:22)
test.data
```

```
##   grp1 grp2 grp3
## 1    1   10   20
## 2    2   11   21
## 3    3   12   22
```

```
stacked = stack(test.data)
stacked
```

```
##   values  ind
## 1      1 grp1
## 2      2 grp1
## 3      3 grp1
## 4     10 grp2
## 5     11 grp2
## 6     12 grp2
## 7     20 grp3
## 8     21 grp3
## 9     22 grp3
```

This then makes it much easier to work with the different groups e.g. calculate means, or plotting the data.

5.6 Daily means from hourly means — processing wind direction data

Sometimes it is necessary or useful to calculate daily means from hourly data. Many particle measurements, for example, are measured as daily means and not hourly means. If we want to analyse such particle data for example, by considering how it varies with meteorological data, it is necessary to express the meteorological (and maybe other data) as daily means. It is of course straightforward to calculate daily means of concentrations and wind speeds, as shown elsewhere in this document. However, this is not the case for wind directions. For example the average of 10° and 350° is 0° (or 360°) and not 180°.

The way to deal with this is to average with *u* and *v* wind components. A function has been written to do this:

```
dailymean <- function(mydata) {
    ## for wind direction, calculate the components
    mydata$u = sin(2 * pi * mydata$wd / 360)
    mydata$v = cos(2 * pi * mydata$wd / 360)
    dailymet = aggregate(mydata,  list(Date = as.Date(mydata$date)), mean,
                         na.rm = TRUE)
    ## mean wd
    dailymet = within(dailymet, wd <- atan2(u, v) * 360 / 2 / pi)
    ## correct for negative wind directions
    ids = which(dailymet$wd < 0) # ids where wd < 0
    dailymet$wd[ids] = dailymet$wd[ids] + 360
    dailymet = subset(dailymet, select = c(-u, -v, -date))
    dailymet
}
```

In this function a data frame is supplied containing hourly data and the returned data frame contains correctly formatted daily data. Note that very similar functions can be used to calculate means over other time periods e.g. hourly means from 15-minute data. The code below shows the use of this function.

```
mydaily = dailymean(mydata) # calculate daily means
# show top of data frame
head(mydaily)
```

```
##         Date     ws       wd      nox      no2       o3     pm10      so2
## 1 1998-01-01  6.835 190.1582 173.5417 39.36364 6.869565 18.16667 3.152609
## 2 1998-01-02  7.070 225.9475 129.7917 39.47826 6.478261 27.75000 3.944891
## 3 1998-01-03 11.015 221.4572 119.6250 37.95652 8.409091 20.16667 3.203986
## 4 1998-01-04 11.485 219.2077 105.9583 35.26087 9.608696 20.95833 2.963043
## 5 1998-01-05  6.610 238.1862 170.6250 46.04348 4.956522 24.20833 4.523261
## 6 1998-01-06  4.375 196.1912 211.1042 45.30435 1.347826 34.62500 5.702935
##         co pm25  nox.all
## 1 2.699239  NaN 173.5417
## 2 1.768043  NaN 129.7917
## 3 1.742101  NaN 119.6250
## 4 1.620435  NaN 105.9583
## 5 2.125870  NaN 170.6250
## 6 2.533478  NaN 211.1042
```

5.7 Using an Editor

5.7.1 Using the built-in editor

As you begin to use R, you will quickly realise there are more efficient ways to do things other than just typing in commands. Often, you will want access to a series of commands for say, plotting a graph with defaults of your choosing. What is needed is an *Editor*. R has an in-built editor — just select File|New script... and the editor window will open. This is a bit like Notepad in Windows. While working it can be useful to put together lines of code in the editor, select the code, right-click and run it.

Advice

As you increase the amount of coding you do, it becomes increasingly difficult to remember what the code actually does. It is always a good idea to liberally comment your code with lines starting with a #. This is especially important if you intend making your code available to others. See Figure 5.1 for an example

of how commented lines are coloured differently in a dedicated editor, making it easy to distinguish between the code and code comments.

5.7.2 Using a dedicated editor

The built-in editor is useful for small amounts of work. However, with use you will find a 'dedicated' editor easier to use. We recommend something called *RStudio*, a screen shot is shown in Figure 5.1. At the time of writing RStudio is a beta version of the software, but is already very good — and likely to get much better in time. More details can be found at `http://http://rstudio.org/`. There are also a lot of well-written 'knowledge base' articles, see `http://support.rstudio.org/help/kb`. Below are a few features that makes RStudio useful for working with **openair** and other R projects.

- It works on Windows, linux and Apple Mac.
- It has been developed by people that clearly use R.
- In the top left pane of Figure 5.1 is where you can work on your R script e.g. use it to develop and save a series of analyses for a report. Note that if you type the name of a function (or part of) R Studio will offer *completions* if you press TAB. This feature also works in the R consule, shown in the bottom left pane.
- It is easy to send a selection or line from the script to the R console by selecting 'Run Line(s)'.
- In the top right pane you can view the objects in your workspace. If you click on one you can view all or most of an object as appropriate. The 'history' tab gives a summary of all the commands you have input, which you can search through.
- At the bottom right there are several tabs. In Figure 5.1 the plot tab is shown, which shows the most recent plot made using the console. The 'Packages' tab provides a neat way of loading a package — just select the one you want and it will load it and all dependent packages.

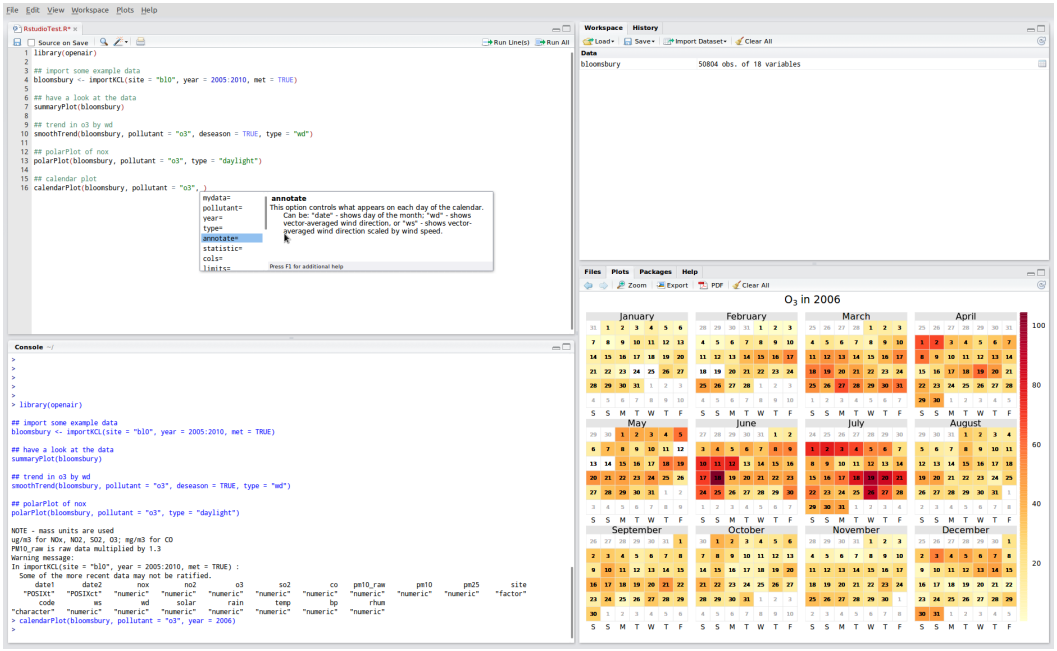


FIGURE 5.1 RStudio is one of the best R editors around.

5.8 Several plots on one page

Often it is useful or necessary to plot more than one plot on a page. This is the sort of task that can be fiddly to carry out if the plots are produced separately and then need to be combined. Problems include alignment and sizing, which as a minimum can be frustrating to get right. R makes it easy to plot any number of plots on a page in a neat and consistent way. The key is to use the `par` function to set up the page as you want it *before* you plot your graphs. The `par` function can control and fine-tune a vast number of plot options — see `help(par)` for specific information.[6]

To set up a page to plot several plots in a regular grid, the esoterically named `mfrow` or `mfcol` option is set. For example, to plot two graphs side-by-side one types in:

```
par(mfrow = c(1, 2))
```

This sets up the plot window for 1 row and 2 columns. And `par(mfrow = c(2, 3))` therefore would allow for six plots in 2 rows and 3 columns etc.

Therefore, the code below produces two plots, side-by-side of NO_x and NO_2, of montly mean concentrations as shown in Figure 5.2.

Note, however that all **openair** functions use `lattice` graphics, where a slightly different approach is required. See Section 8.7 for more details on how to plot several **openair** plots on one page.

[6] Note that this will only work with base graphics and not `lattice`.

```
par(mfrow = c(1, 2))
# first plot
means <- tapply(mydata$nox, format(mydata$date,"%Y-%m"), mean, na.rm = TRUE)
dates <- seq(mydata$date[1], mydata$date[nrow(mydata)], length = nrow(means))
plot(dates, means,
     type = "l",
     col = "darkgreen",
     xlab = "year",
     ylab = "nitrogen oxides (ppb)")
means <- tapply(mydata$no2, format(mydata$date,"%Y-%m"), mean, na.rm = TRUE)
dates <- seq(mydata$date[1], mydata$date[nrow(mydata)], length = nrow(means))

# second plot
plot(dates, means,
     type = "l",
     col = "skyblue3",
     xlab = "year",
     ylab = "nitrogen dioxide (ppb)")
```

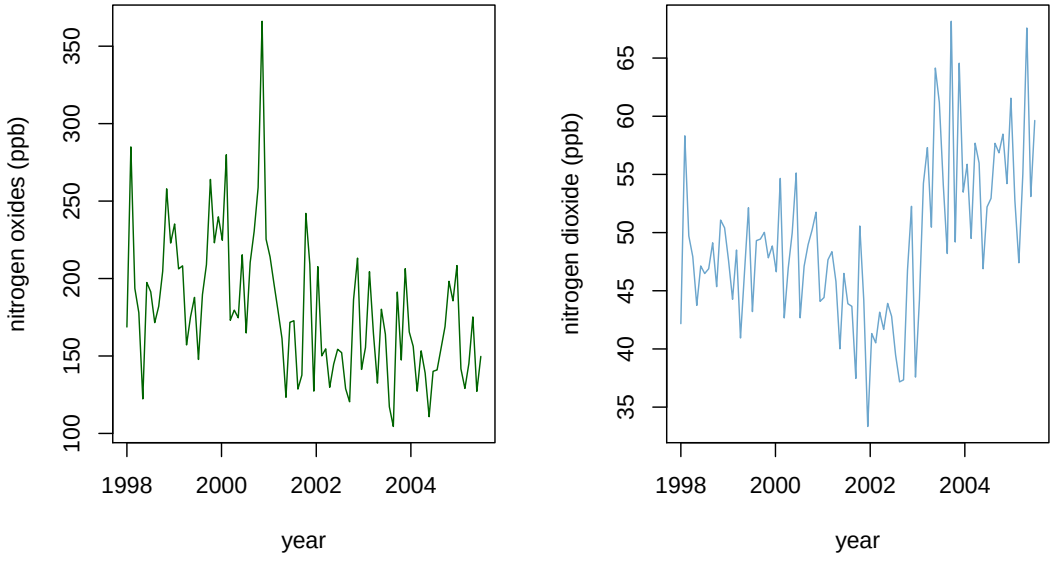


FIGURE 5.2 Plotting two plots side-by-side using the `par` setting.

5.9 Saving and using plots

There are several ways of saving and using plots and some general advice is given in this section. The simplest way to use a plot from the R GUI is to size to the required size, right-click on it and choose to copy as a metafile or a bitmap; this being on Windows. A *metafile* is a Windows-specific format, whereas a *bitmap* is a more general system independent format. An important difference between the two formats is that a metafile is in a *vector* graphics format whereas a bitmap is a *raster* format made up of a regular grid of pixels. Vector-based graphics are made up of lines and shapes that remain sharp at any resolution. Zoom in on a metafile plot and it will remain sharp. By contrast zooming in on a bitmap image will reveal fuzziness — and eventually the individual pixels. Other vector formats include pdf and encapsulated postscript (eps) and other bitmap images include png (portable network graphics) and jpeg (the format most often used for photographs).

Most plots in **openair** work best using vector graphics. This manual for example uses LaTeX and the plots are in pdf format i.e. they are sharp when you zoom in.

However, most users of **openair** probably use applications such as Microsoft Word. Word does accept a wide range of graphic formats, but unfortunately even recent versions cannot use pdf graphics. In this case it will generally be best to copy plots as metafiles. Unfortunately, Windows metafiles in R cannot handle alpha transparency. What this means is for certain plots that use semi-transparent shading (e.g. `timeVariation`, Section 21), copying the plot as a metafile will not work for those semi-transparent areas in the plot. The other problem with metafiles is that a Word document containing many of them will be slow to render and frustrating to work with, as well as being potentially very large in file size.

So what's best? A good compromise is the png format. These images capture **openair** graphics well, are generally small file sizes and can be used by almost all applications. They also capture alpha transparency. It is possible to save as a png format using the R GUI and RStudio from the file menu, or the export button, respectively. By default the png will be saved at 72 dots per inch, which might not be sufficiently detailed, especially if it is resized later. A better way to work (in my view) is to always use scripts to make the plots. This has two main advantages. First, all analyses *including plots* will entirely and easily reproducible, which is a major advantage. Second, it is possible to have more control over how the plots are saved.

The way to save a plot using code is as follows.

```
png("myPlot.png")
polarPlot(mydata,  pollutant = "so2")
dev.off()
```

The code above first prepares a *device* for printing on (in this case a png device) and save *myPlot.png* in your working directory. Second, a plot is produced and third (importantly!) the device is closed. Note that if you fail to add the final line the device will remain open and the file will not be written. By default, `png` will save a file 480×480 pixels; see ?png for details. This resolution might be fine for simple graphics, but a much better approach is to be sure of the final resolution. Most commonly it is recommended that bitmap graphics in printed documents are 300 dpi (dots per inch). So, an approach that guarantees this resolution is as follows. Think about the size of the plot needed in inches (in this case 5 inches wide by 4.5 inches high) and use this in the call to `png`:

```
png("myPlot.png", width = 5 * 300, height = 4.5 * 300, res = 300)
polarPlot(mydata,  pollutant = "so2")
dev.off()
```

This will produce a sharp plot which at 5×4.5 inches will be 300 dpi. In some cases e.g. in quality publications, it might be necessary to save at an even higher resolution. For example to save at 600 dpi just replace the 300 above with 600.

It should also be noted that this approach can usefully control the relative size of the font in a plot. Users often ask how to change font size — but this is currently hard-wired in most functions. To see the effect try comparing the output of the plot above (5×4.5 inches) with the one below (3.5×3 inches). You will see that the latter has font that is relatively larger compared with the rest of the plot.

```
png("myPlot.png", width = 3.5 * 300,
    height = 3 * 300, res = 300)
polarPlot(mydata,  pollutant = "so2")
dev.off()
```

Another point worth mentioning is that in some cases it is useful to overlay plots on a map. In these cases it is useful to set the background to be transparent rather than white (the default). To do this just supply the option `bg = "transparent"`.

5.10 Graphing lots of data — using level plots

It is often the case when plotting monitoring data that there are so many data points it gets hard to see relationships. Consider the scatter plot for NO_x and NO_2 shown in Figure 5.3 — it is very difficult to see how NO_x and NO_2 are related because of the large number of points. An alternative way of plotting such data is to 'bin' it first and count the number of points in each bin and plot it as a *level* plot.

The first part of the code in above will produce a basic plot using the base graphics function `image`, shown in Figure 5.3. Now it is possible to see the relationship between NO_x and NO_2 much more clearly. It also has the benefit of showing where most of the data are, which is not very apparent in Figure 5.3.

A better looking plot can be produced with a bit more work using *lattice* graphics, as shown in Figure 5.4. This plot has the advantage of also showing a scale, which in this case in the number of points in each bin. Lattice graphic are considerd more in(§6).

5.11 Special symbols for use in plotting air pollution data

Air pollution concentrations are expressed in many ways, perhaps most commonly in µg m^{-3}. It is always preferable to display these units properly rather than, for example as ug/m3. The same is also true for subscripts in pollutant names such as NO_x and $PM_{2.5}$. R has its own way of dealing with specialist symbols, which is similar to LaTeX. This section provides code for commonly used expressions. For more information type `help(plotmath)`. We illustrate the use of these symbols through examples shown in Table 5.1, by setting the y-axis label. However, these labels can be used elsewhere too, such as in titles or to annotate specific parts of a plot.

To demonstrate what these symbols look like, Figure 5.5 provides an example. The code is shown below.

```
x <- mydata$nox
y <- mydata$no2
# find maximum values
x.max <- max(x, na.rm = TRUE)
y.max <- max(y, na.rm = TRUE)
# set the bin interval
x.int <- 5
y.int <- 2
# bin the data
x.bin <- cut(x, seq(0, x.max, x.int))
y.bin <- cut(y, seq(0, y.max, y.int))
# make a frequency table
freq <- table(x.bin, y.bin)
# define x and y intervals for plotting
x.range <- seq(0 + x.int/2, x.max - x.int/2, x.int)
y.range <- seq(0 + y.int/2, y.max - y.int/2, y.int)
# plot the data
image(x = x.range, y = y.range, freq, col = rev(heat.colors(20)))
```

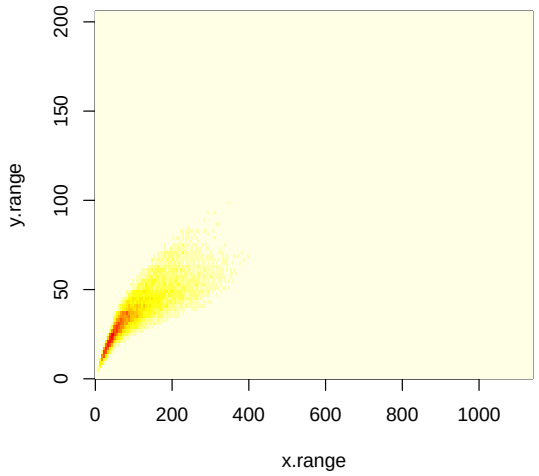


FIGURE 5.3 Using the `image` function to plot NO_x against NO_2.

TABLE 5.1 Examples of commonly used text formats for air pollution.

Text required	Expression
NO_x	`ylab = expression("NO"[X])`
$PM_{2.5}$	`ylab = expression("PM"[2.5])`
($\mu g\ m^{-3}$)	`ylab = expression("(" * mu * "g m" ^-3 * ")")`
PM_{10} ($\mu g\ m^{-3}$)	`ylab = expression("PM"[10] *  " (" * mu * "g m" ^-3 * ")")`
Temperature (°C)	`xlab = expression("Temperature (" * degree * "C)")`

```
library(lattice)
grid <- expand.grid(x = x.range, y = y.range)
z <- as.vector(freq)
grid <- cbind(grid, z)
levelplot(z ~ x * y, grid, col.regions = rev(heat.colors(20)))
```

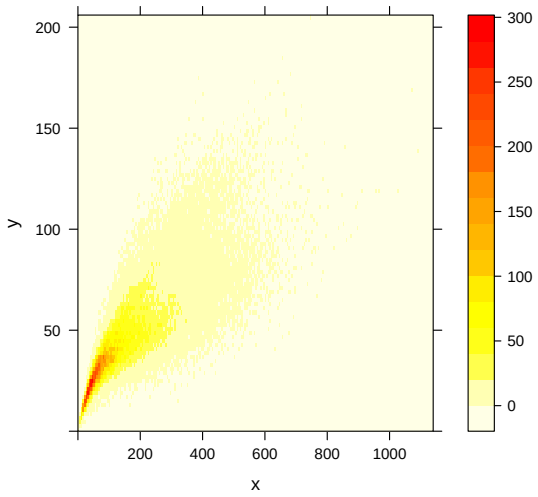


FIGURE 5.4 Using the lattice graphics `levelplot` function to plot NO_x against NO_2.

```
plot(1, 1,
     xlab = expression("Temperature (" * degree * "C)"),
     ylab = expression("PM"[10]~"(" * mu * "g m" ^-3 * ")"),
     main = expression("PM"[2.5] * " and NO"[x] * " at Marylebone Road"),
     type = "n")
text(1, 1, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(.8, .8, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})), cex = 1.2)
text(1.2, 1.2, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
```

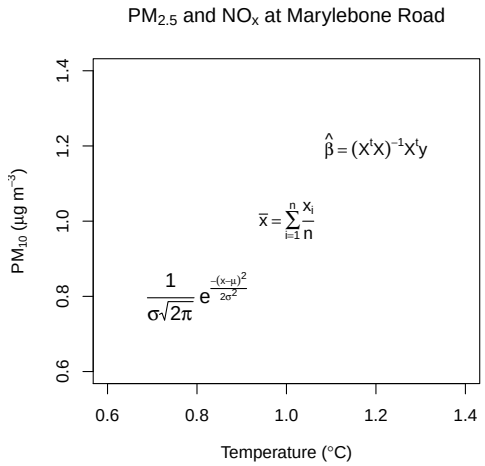


FIGURE 5.5 Examples of different symbols that can be used in R plots.

5.12 Using databases with R

So far the discussion has focussed on working with .csv files, which might be adequate for most purposes. However, as the amount of data increases, the storage of it in this way is not to be recommended. A much better approach is to store data in a database. There are several advantages in doing so. First, it forces a more disciplined approach to storage (e.g. variables names and formats). Second, it is possible to store a lot more information in this way. Finally, for very large amounts of information R can run out of memory because all the calculations are done in RAM.[7] In the latter case it can be much better to use the SQL database language to do some of the work first and then bring smaller data sets into R.

There are several database types that R can work with; perhaps the most common being Microsoft Access. For those interested in open-source databases MySQL is highly recommended (we use this with many large data sets). However, a discussion of MySQL goes beyond the aims of this document. It should also be noted that you do not actually need to have Microsoft Access to read or write data to it.

Here is an example of how to connect to an Access database file (Access 2007) (file available from David Carslaw), which contains exactly the same data as the 'example data long.csv'.

[7] Now that there is a 64-bit version of R for Windows, this is less of a problem and the limitation is more to do with the amount of RAM the computer has.

```
library(RODBC)
## set time zone to GMT
Sys.setenv(TZ = "GMT")
## connect to a database file
channel <- odbcConnectAccess2007("c:/users/david/openair/Data/example data long.mdb")

## read all data in
test.data <- sqlQuery(channel, "select * from dbdata")

## read date, nox and no2
test.data <- sqlQuery(channel, "select date, nox, no2 from dbdata")
head(test.data)

##                  date nox no2
## 1 1998-01-01 00:00:00 285  39
## 2 1998-01-01 01:00:00  NA  NA
## 3 1998-01-01 02:00:00  NA  NA
## 4 1998-01-01 03:00:00 493  52
## 5 1998-01-01 04:00:00 468  78
## 6 1998-01-01 05:00:00 264  42

## select data where nox > 500 ppb
test.data <- sqlQuery(channel, "select * from dbdata where nox > 500")
head(test.data)

##                  date  ws  wd nox no2 o3 pm10     so2     co pm25
## 1 1998-01-15 14:00:00 7.2 230 504  83  3   54  7.6450 4.5050   NA
## 2 1998-01-15 17:00:00 4.8 230 508  74  2   42  7.8125 6.6225   NA
## 3 1998-01-15 19:00:00 5.4 200 535  48  2   47  9.2600 7.1625   NA
## 4 1998-01-15 20:00:00 4.8 190 587  79  2   49 10.2750 8.1225   NA
## 5 1998-01-21 07:00:00 1.2 180 578  75  2   65 17.4275 3.5950   NA
## 6 1998-02-02 07:00:00 2.4   0 607  85  3   79 17.0525 4.7025   NA

## select between two dates
test.data <- sqlQuery(channel, "select * from dbdata where date >= #1/1/1998# and date <= #31/12/1999
tail(test.data)

##                      date   ws  wd nox no2 o3 pm10    so2    co pm25
## 17515 1999-12-31 18:00:00 4.68 190 226  39 NA   29 5.4550 2.375   23
## 17516 1999-12-31 19:00:00 3.96 180 202  37 NA   27 4.7850 2.150   23
## 17517 1999-12-31 20:00:00 3.36 190 246  44 NA   30 5.8750 2.450   23
## 17518 1999-12-31 21:00:00 3.72 220 231  35 NA   28 5.2800 2.225   23
## 17519 1999-12-31 22:00:00 4.08 200 217  41 NA   31 4.7875 2.175   26
## 17520 1999-12-31 23:00:00 3.24 200 181  37 NA   28 3.4825 1.775   22

## close the connection
close(channel)
```

In the code above a connection is first made to the data base file, followed by examples of various SQL queries. When connecting to databases in the way described above, the date field is automatically recognised by R and there is no need to convert it as in the case for the .csv file. For those interested in using databases with R, it is worth looking in the help files of RODBC for a more comprehensive explanation of the capabilities.

The RODBC package will automatically try and preserve data formats, including those for date/time. We have experienced a few difficulties here to do with British Summertime and GMT. RODBC will bring data in in a format consistent with what the operating system is set to, which can be either BST or GMT (or other time zones). The best option in our view is that before the data are imported, set the system to GMT as above. This will avoid all sorts of potential problems.

6 Multivariate plots — introduction to the Lattice package

6.1 Introduction to the Lattice package

In (§2.1) one of the benefits highlighted in using R was the extensive number of *packages* available that extend the core features of R. One package called `lattice` is particularly useful for plotting and analysing monitoring data. The `lattice` package is based on the original S (S-Plus) *Trellis* package that provides excellent multivariate plotting capabilities.[8] This is one of the stronger capabilities that R has and greatly enhances the possibilities for plotting monitoring data. The original trellis graphics were designed by William Cleveland at Bell laboratories and were based on research into how best to visualise graphics. For those interested in this there are a couple of books available (Cleveland 1985; Cleveland 1993). Lattice graphics can be used to produce similar plots to those shown elsewhere in this document, but in some cased can produce better looking plots with better-chosen defaults. However, the real strength of Lattice is the ability to deal with multivariate data and to plot several plots on one page.

Installing and loading a package in R

The capabilities of R are greatly enhanced by a number of optional packages. To use different packages, they must first be installed. Many packages such as `lattice` are installed as part of the R installation itself. However, they need to be loaded to use them. This can be done in two principal ways: use the menu and choose Packages | Load package... and choose from the available packages listed; or in code you can issue a command `library(package name)`.

In many cases the package you want may not be installed on your system. In this case you can choose Packages | Install package(s)..., where you are then prompted for a location to install from (chose one in the UK). It is possible that this option will not work due to firewalls etc. information is being downloaded from a remote server. An alternative way of doing this is to go the main R web pages, select CRAN (Comprehensive R Archive Network), choose the appropriate web site address (again UK) and under the heading 'Software' choose 'packages'. Choose the package you want (for Windows, choose Windows binary zip file). Download the file to your hard disk and then choose Packages | Install package(s) from local zip files....

The drawback of using the `lattice` package is that most new users (and some experienced one too) find it difficult to use. The emphasis is very much on the use of code to make plots. The focus of this section therefore is to provide some examples of the use of Lattice graphics on our data set that can be readily applied to other data sets with minimum or no alteration.

6.2 Example simple plots

We start with plotting the basic time series of NO_x as shown in Figure 4.5. The code is shown below. For basic plotting like this, the terminology is straightforward. One of the first things to note is the use of a formula to represent the plot `nox ~ date`. You can think of this as an equation for plotting data i.e. y = fn(x). Therefore, what appears on the y-axis is given first (in this case `nox`), then the x-axis data (in this case

[8] The names trellis and lattice are meant to reflect the idea of multiple plots i.e. like a garden trellis.

```
xyplot(nox ~ date, data = mydata, type = "l")
```

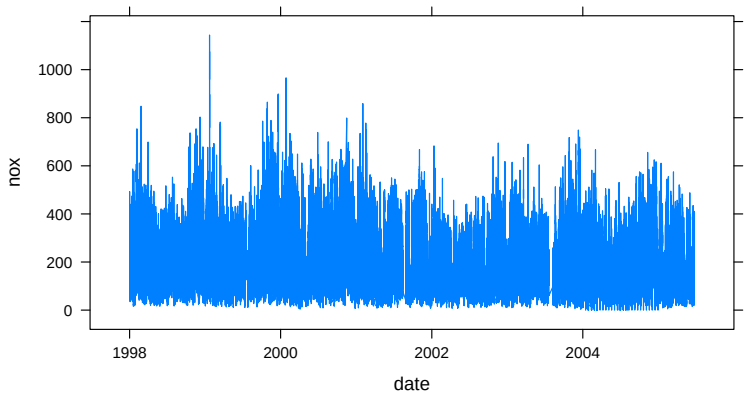


FIGURE 6.1 Example plot using the `lattice` package to plot hourly concentrations of NO_x at Marylebone Road

`date`). Also given is the argument `data = mydata`, and the type of plot `type = "l"` as before.

The plot generated is shown in Figure 6.1, which can be compared with Figure 4.5. There are a few differences to note: the default colour is blue, the y-axis labels are horizontal (for easier reading) and their are tick marks shown on all all sides (outside the plot so that they do not clutter-up the data actually shown). The lattice plot can be annotated in much the same way as the base plots, with options such as `ylim`, `ylab` etc.

Much of the power of lattice graphics lies in the ability to plot one variable against another *dependent* on a third. To get an idea of what is meant here, consider how NO_2 varies by NO_x by day of the week. Now, day of the week is a *categorical* variable, which in R is referred to as a *factor*. We illustrate the use of this type of plotting by making some simple artificial data. In the code below we first define the days of the week. We then make a data frame where the NO_x concentrations are 70 random numbers between 0 to 5 (`nox = 5 * runif(70)`) i.e. 10 for each day of the week. The NO_2 concentrations are similarly assumed to be 70 random numbers, which are between 0 to 1 in this case and the days of the week `weekday` are each of the days repeated 10 times. When lattice plots a factor, it does so alphabetically. However, this makes little sense for the days of the week and therefore we force the ordering of the days with the code shown. Finally, a plot is produced. Note that in the plot command the formula `no2 ~ nox | weekday` is used. This means in simple terms 'plot no2 against nox, but show how it varies by day of the week'. For some reason, lattice always fills plots from the bottom left corner to the top right. The `as.table = TRUE` command forces it to plot it from the top left to the bottom right, which more most applications seems like a more logical way of plotting. The result of the plotting is shown in Figure 6.2.

6.3 A more complicated plot — plot each year of data in a separate panel

Now we get onto the real power of Lattice: multiple plots on a page that can convey lots of useful information. When a lot of data are available, it is very useful to be able to plot it all quickly and view it. In our data set we have over 65,000 lines of data, which if plotted as a typical x-y plot would be hard to assess. A better way is to plot each year separately and plot all years on 1-page. The code below performs this function.

```
# weekday names
weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
              "Saturday", "Sunday")

# make data frame
test.data = data.frame(nox = 5 * runif(70), no2 = runif(70),
            weekday = as.factor(rep(weekdays, each = 10)))

# order the days i.e. do not want the default alphabetical
test.data$weekday = ordered(test.data$weekday, levels = weekdays)

# plot
xyplot(no2 ~ nox | weekday, data = test.data, as.table = TRUE)
```

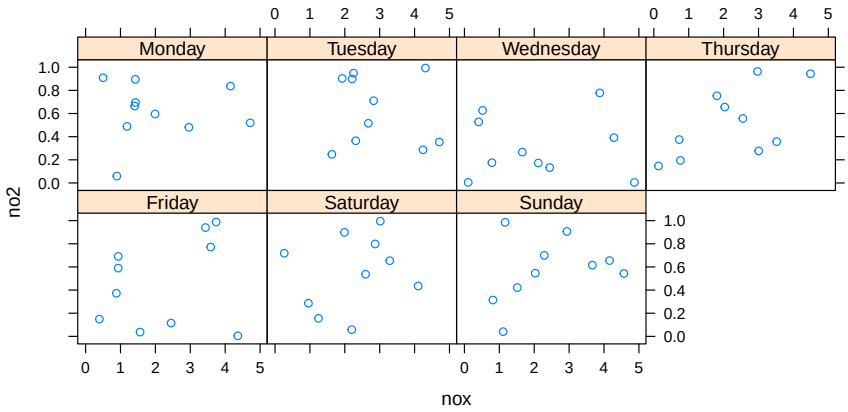


FIGURE 6.2 Example plot using the `lattice` package to plot two variables against each other (`nox` and `no2`) , but dependent on a third (`weekday`).

The code is explained in three main sections.

Section 1 What is needed first is to convert the date to a year and convert this year to a factor, so that it can be plotted as a categorical variable. The code `format(mydata$date, "%Y")` converts the date to year format and the `as.factor` converts this (numerical value) to a factor. The year is then added to the data frame `mydata` using the column bind command `cbind`.

Section 2 This bit of code simply finds the start and end years of the data, which are read into two variables `begin.year` and `end.year`. These variables are used in the plot function and makes the function rather more easy to read.

Section 3 The third part of the code plots the data. The `aspect` option sets the dimensions of the plot (1 would be a square; 0.5 is a plot twice as wide as it is long). The `scales` option is quite complicated. What this does is manually set the x-axis points every two months and uses the three letter month summary option `%b`. The plot itself contains several panel functions that add grid lines and plot the actual data. Again, it will take some digesting to understand this code, but it should be usable with most hourly data sets and can be applied without knowing all the details.

What Figure 6.3 shows is a huge amount of data in a very compact form. It is easy to see for example some missing data in July 1999, or very high concentrations of NO_x in January 1999.

```
# SECTION [1]
mydata$year <- as.factor(format(mydata$date, "%Y"))

# SECTION [2]
# determine begin/end year (+1) for gridlines and axis
begin.year <- min(mydata$date)
end.year <- max(mydata$date)

# SECTION [3]
xyplot(nox ~ date | year,
       data = mydata,
       aspect = 0.4,
       as.table = TRUE,
       scales = list(relation = "free", x = list(format = "%b",
                                    at = seq(begin.year, end.year,
                                             by = "2 month"))),
       panel = function(x, y) {
           ## add grid lines every month by finding start/end date
           panel.abline(v = seq(begin.year, end.year, by = "month"),
                        col = "grey85")
           panel.abline(h = 0, col = "grey85")
           panel.grid(h = -1, v = 0)
           panel.xyplot(x, y, type = "l", lwd = 1)
       })
```

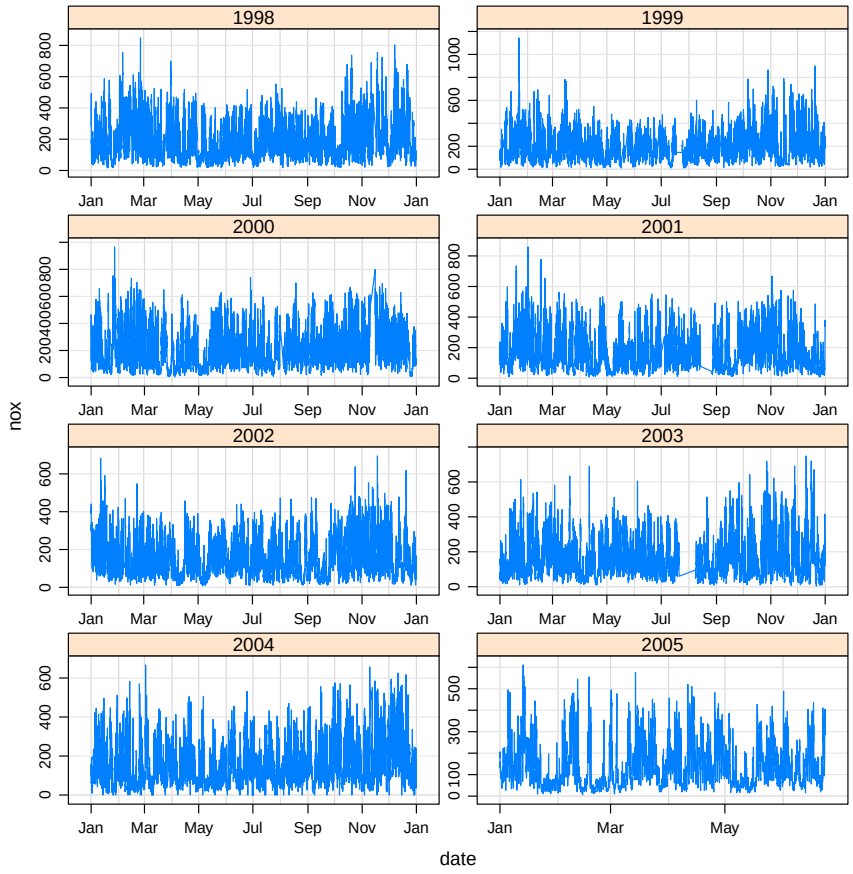


FIGURE 6.3 Example plot using the `lattice` package to plot hourly concentrations of NO_x at Marylebone Road

6.4 Showing trends dependent on a third variable

We now analyse the data in quite a complex way. Here we see the real power of lattice plots in analysing data. The code below can be modified to look at the data in all kinds of ways with only simple modification. For this analysis however, we aim to do three things:

1. Average the data by month of the year. Monthly averages are a convenient way of summarising data.
2. Split these averages by different wind sectors. By considering the trends by different wind sectors, some insights can be gained the trends in different source types.
3. Apply a smoothing line to highlight the trends. In this case a locally-weighted regression line is applied.

This analysis also makes use of some very useful functions, which are part of the base system of R. The first is `cut`, which provides a powerful way of dividing data up in different ways; in this case creating eight different wind direction sectors. The second is `aggregate`, which neatly summarises the data by monthly mean *and* wind sector. A summary of the main parts of the analysis is given next.

Divide the wind directions into eight sectors This code uses the `cut` command.

Define the levels for the different wind sectors This code gives a nicer description of the wind sectors that will be used when plotting the graphs.

Summarise the data This part of the code calculates the mean concentrations of NO_x by year-month and by wind sector.

The results from the analysis are shown in Figure 6.4, which highlights several interesting features.

```
# divide-up date by wd
wd.cut <- cut(mydata[, "wd"], breaks = seq(0, 360, length = 9))

# define the levels for plotting
wd <- seq(0, 360, by = 45)
levels(wd.cut) <- paste(wd[-length(wd)], "-", wd[-1], " degrees", sep = "")

# summarise by year/month and wd
summary.data <- aggregate(mydata["nox"], list(date = format(mydata$date,"%Y-%m"),
                wd = wd.cut), mean, na.rm = TRUE)

# need to get into year/month/day
newdate = paste(summary.data$date,"-01", sep = "")
newdate = as.Date(newdate, format = "%Y-%m-%d")

# add to summary
summary.data <- cbind(summary.data, newdate)

# plot
xyplot(nox ~ newdate | wd,
       data = summary.data,
       layout = c(4, 2),
       as.table = TRUE,
       xlab = "date",
       ylab = "nitrogen oxides (ppb)",
       panel = function(x, y) {
           panel.grid(h = -1, v = 0)
           panel.abline(v = seq(as.Date("1998/1/1"), as.Date("2007/1/1"),
                        "years"),
                        col = "grey85")
           panel.xyplot(x, y, type = "l", lwd = 1)
           panel.loess(x, y, col = "red", lwd = 2, span = 0.2)
})
```

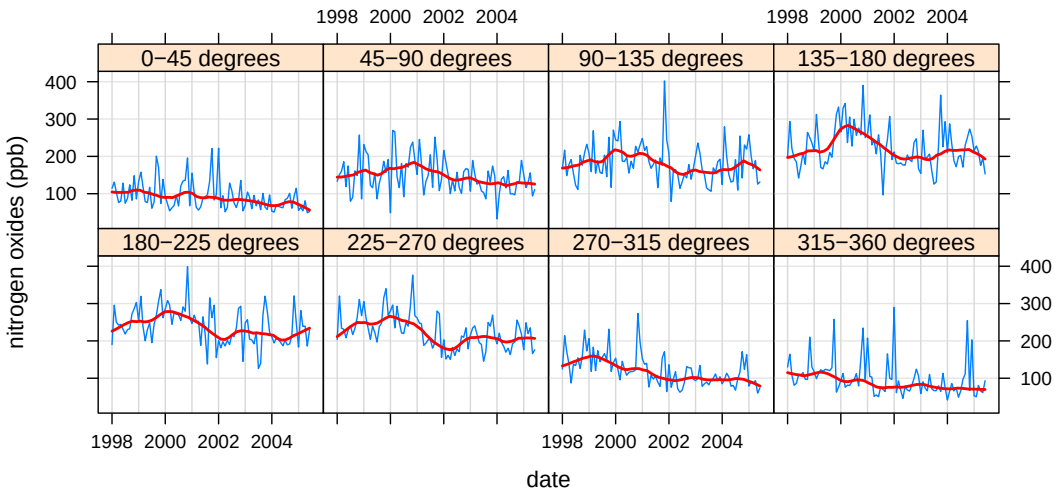


FIGURE 6.4 Example plot showing how a time series can a) be summarised by monthly means, b) split by wind sector, and c) show a locally-weighted smooth trend for each panel.

7 Functions in R

This section highlights the importance of functions in R to carry out specific tasks. Most of the rest of Part II of this report considers the use of dedicated functions written to analyse air pollution data. This section gives an overview of why functions are important and how they work. Functions are useful in many ways:

- For repetitive tasks they help take the effort out of processing data. An example would include a function to import and clean-up data.
- Functions provide a much more structured way of working. They help to break down big problems into smaller bits that are easier to work with.
- For air pollution analysis, dedicated functions can (and have) been written to offer unique analysis capabilities that are not offered in any other software. R offers excellent capabilities here for creating new analyses and plots.

Let's consider a very simple function that adds two numbers (although one would not actually ever need to write such a simple function!):

```
add.two <- function(a = 1, b = 2)
          {a + b}
```

The function name is `add.two`, it accepts two arguments `a` and `b`. The body of the function is written within the braces {}. This function can be read into R simply by pasting it is — and it then becomes available for use. Let's use the function to add 10 and 5:

```
add.two(10, 5)
```

```
## [1] 15
```

```
add.two(c(1, 2, 3), 2)
```

```
## [1] 3 4 5
```

```
add.two(c(1, 2, 3), c(5, 6, 7))
```

```
## [1]  6  8 10
```

Easy! Note that in the definition of the function itself, we provided default values for `a` and `b`. If one called the function without supplying `a` or `b` it would return 3:

```
add.two()
```

```
## [1] 3
```

This functionality is useful for testing purposes, but also for providing defaults for some of the variables. If the user does not want to alter a default value, there is no need to supply it.

Part II
Dedicated functions for analysing air pollution data

8 Introduction

Part II of this report focuses on the development and use of dedicated functions written to process air pollution data. These functions greatly extend the capabilities outlined in Part I, where the focus was on developing an understanding of R.

While many of the options in these functions allow quite a sophisticated analysis to be undertaken, the defaults generally use the simplest (and fastest) assumptions. A more detailed analysis can refine these assumptions e.g. by accounting for autocorrelation, or fine-control over the appearance of a plot.

It should be noted that while the aim is to keep this documentation up to date, the primary source of information related to the different functions is contained within the package itself. Once loaded, type `?openair` to see all the help pages associated with the package. The website for **openair** is `http://www.openair-project.org`.

This section contains important information on loading the openair package for the first time and the input data requirements. Users will need to consider the advice in this section to ensure that openair can be used without problems.

8.1 Installing and loading the openair package

The dedicated functions for the analysis of air pollution data have been made available in the **openair** package. As of September 2010, **openair** is available on CRAN. This means it should be very straightforward to install. In Windows, choose the packages menu in R and then choose 'Install package(s)'. You will be prompted for a location from which to download — so scroll down to the appropriate country. Once selected you will then be shown a large list of available packages — choose **openair**.

Second approach to installing openair

The second approach to installing **openair** is slightly more involved but should still be easy. This situation arises if you are using a computer that does not let R communicate externally. These are the steps:

1. Download the **Windows binary** (zip) files from `http://cran.r-project.org/web/packages/` for the following packages and store them somewhere convenient on your computer: 'openair', 'reshape2', 'plyr' and 'RColorBrewer'. This is done by clicking on the link for each package, which will show a page with a downloads section where the zip file is shown.

2. Install **openair** and all the other dependent packages by choosing 'Packages' from the R menu, then 'Install packages(s) from local zip files...', and choose all the zip files that were downloaded.

The package can be tested by going into the 'Packages' menu in R and choosing 'Load package' and then choose **openair**. The package comes with some test data — several years of data from the Marylebone Road site in London ('mydata'). Test it by trying a function e.g.

```
summaryPlot(mydata)
```

Note that it is only necessary to install packages once — unless a package has been updated or a new version of R is installed. Occasionally it is useful to update the packages that have been installed through the 'Update packages' option under the Packages menu. Because the **openair** package (and R itself) are continually updated, it will be useful to know this document was produced using R version 3.1.2 and **openair** version 1.1-4.

8.2 Access to source code

All R code is accessible. On CRAN, you will see there are various versions of packages: Package source, MacOS X binary and Windows binary. The source code is contained in the Package source, which is a tar.gz (compressed file).

For **openair** all development is carried out using Github for version control. Users can access all code used in openair at `https://github.com/davidcarslaw/openair`.

8.3 Brief introduction to openair functions

This section gives a brief overview of the functions in **openair**. The core functions are summarised in Table 8.3, which shows the input variables required, the main purpose of the function, whether multiple pollutants can be considered and a summary of the `type` option. The `type` option given in Table 8.3 gives the maximum number of conditioning variables allowed in each function — more on this later.

Having read some data into a data frame it is then straightforward to run any function. Almost all functions are run as:

```
functionName(thedata, options, ...)
```

The usage is best illustrated through a specific example, in this case the `polarPlot` function. The details of the function are shown in Section 15 and through the help pages (type ?polarPlot). As it can be seen there are a large number of options associated with `polarPlot` — and most other functions and each of these has a default. For example, the default pollutant considered in `polarPlot` is 'nox'. If the user has a data frame called `theData` then `polarPlot` could minimally be called by:

```
polarPlot(theData)
```

which would plot a 'nox' polar plot if 'nox' was available in the data frame `theData`.

Note that the options do not need to be specified in order nor is it always necessary to write the whole word. For example, it is possible to write:

```
polarPlot(theData, type = "year", poll = "so2")
```

In this case writing `poll` is sufficient enough to uniquely identify that the option is `pollutant`.

Also there are many common options available in functions that are not explicitly documented, but are part of lattice graphics. Some of the common ones are summarised in Table 8.1. The `layout` option allows the user to control the layout of multi-panel plots e.g. `layout = c(4, 1)` would ensure a four-panel plot is 4 columns by 1 row.

TABLE 8.1 Common options used in **openair** plots that can be set by the user but are generally not explicitly documented.

option	description
`xlab`	x-axis label
`ylab`	y-axis label
`main`	title of the plot
`pch`	plotting symbol used for points
`cex`	size of symbol plotted
`lty`	line type used
`lwd`	line width used
`layout`	the plot layout e.g. c(2, 2)

Controlling font size

All **openair** plot functions have an option `fontsize`. Users can easily vary the size of the font for each plot e.g.

```
polarPlot(mydata, fontsize = 20)
```

The font size will be reset to the default sizes once the plot is complete. Finer control of individual font sizes is currently not easily possible. See also Section 5.9, which also describes a way of controlling font size when saving graphics.

The openair 'type' option

One of the central themes in **openair** is the idea of *conditioning* plots. Rather than plot *x* against *y*, considerably more information can usually be gained by considering a third variable, *z*. In this case, *x* is plotted against *y* for many different intervals of *z*. This idea can be further extended. For example, a trend of NO_x against time can be *conditioned* in many ways: NO_x vs. time split by wind sector, day of the week, wind speed, temperature, hour of the day...and so on. This type of analysis is rarely carried out when analysing air pollution data, in part because it is time consuming to do. However, thanks to the capabilities of R and packages such as `lattice`, it becomes easier to work in this way.

In most **openair** functions conditioning is controlled using the `type` option. `type` can be any other variable available in a data frame (numeric, character or factor). A simple example of `type` would be a variable representing a 'rbefore' and 'after' situation, say a variable called `period` i.e. the option `type = "period"` is supplied. In this case a plot or analysis would be separately shown for 'before' and 'after'. When `type` is a numeric variable then the data will be split into four *quantiles* and labelled accordingly. Note however the user can set the quantile intervals to other values using the option `n.levels`. For example, the user could choose to plot a variable by different levels of temperature. If `n.levels = 3` then the data could be split by 'low', 'medium' and 'high' temperatures, and so on. Some variables are treated in a special way. For example if `type = "wd"` then the data are split into 8 wind sectors (N, NE, E, ...) and plots are organised by points of the compass.

There are a series of pre-defined values that `type` can take related to the temporal components of the data. To use these there *must* be a `date` field so that the can be calculated. These pre-defined values of `type` are shown in Table 8.2 are both useful and convenient. Given a data frame containing several years of data it is easy to analyse the data e.g. plot it, by year by supplying the option `type = "year"`. Other

useful and straightforward values are ”hour” and ”month”. When `type = "season"` **openair** will split the data by the four seasons (winter = Dec/Jan/Feb etc.). Note for southern hemisphere users that the option `hemisphere = "southern"` can be given. When `type = "daylight"` is used the data are split between nighttime and daylight hours. In this case the user can also supply the options `latitude` and `longitude` for their location (the default is London).

TABLE 8.2 Pre-defined time-based values for the **openair** `type` option.

option	splits data by ...
”year”	year
”hour”	hour of the day (0 to 23)
”month”	Month of the year
”season”	spring, summer, autumn, winter
”weekday”	Monday, Tuesday, ...
”weekend”	Saturday, Sunday, weekday
”monthyear”	every month-year combination
”gmtbst”	separately considers daylight saving time periods
”daylight”	nighttime and daylight

Table 8.3 summarises the functions that accept the option ‘type’ and the number of types that can be set. Numerous examples of conditioning are given throughout this document.

TABLE 8.3 Summary of main **openair** analysis functions. Click on function name to be taken to the section on that function.

Function	Mandatory variables	Purpose	Multiple pollutants	type option
calcFno2	see §(30) for details	estimate primary NO_2 emissions ratio from monitoring data	no	no
calendarPlot	date, one numeric field	Calendar-type view of mean values	no	no
conditionalEval	observed and modelled values and other variables(s)	extensions to `conditionalQuantile`	no	yes [1]
conditionalQuantile	observed and modelled values	quantile comparisons for model evaluation	no	yes [2]
GoogleMapsPlot	two numeric fields for latitude/longitude	annotate Google maps	no	Yes [2]
`kernelExceed`	date, ws, wd, one other numeric field	bivariate kernel density estimates for exeedance statistics	no	Yes [1]
linearRelation	date, two numeric fields	explore linear relationships between variables in time	no	limited
TheilSen	date, one numeric field	Calculate Theil-Sen slope estimates and uncertainties	no	Yes [2]
modStats	observed and modelled values	calculate a range of model evaluation statistics	no	yes [≥1]
percentileRose	wd, one other numeric field	percentiles by wind direction	no	Yes [2]
polarAnnulus	date, ws, wd, one other numeric field	polar annulus plot for temporal variations by wind direction	yes	Yes [2]
polarCluster	ws, wd, one other numeric field	cluster analysis of bi-variate polar plots for feature extraction	No	No
polarFreq	ws, wd	alternative to wind rose/pollution rose	no	Yes [2]
polarPlot	ws, wd, one other numeric field	bi-variate polar plot	yes	Yes [2]
pollutionRose	ws, wd, one other numeric field	pollution rose	no	Yes [2]
scatterPlot	x and y values to plot	traditional scatter plots with enhanced options	no	Yes [2]
smoothTrend	date, one numeric field	smooth trend estimates	yes	Yes [2]
summaryPlot	date, one numeric field	summary view of a data frame	yes	no
TaylorDiagram	two numeric fields	model evaluation plot	no	Yes [2]
timePlot	date, one numeric field	Time-series plotting	yes	Yes [1]
timeProp	date, one numeric, one category field	Time-series plotting with categories as stacked bar chart	yes	Yes [1]
timeVariation	date, one numeric field	diurnal, day of week and monthly variations	yes	Yes [1]
trajCluster	data from `importTraj`	HYSPLIT back trajectory cluster analysis	no	Yes [2]
trajPlot	data from `importTraj`	HYSPLIT back trajectory plots — points of lines	no	Yes [2]
trajLevel	data from `importTraj`	HYSPLIT back trajectory plots — binned or smoothed	no	Yes [2]
trendLevel	date, one other numeric field	flexible level plots or 'heat maps'	no	Yes [2]
windRose	date, ws, wd	traditional wind rose	no	Yes [2]

TABLE 8.4 Summary of **openair** utility functions. Click on function name to be taken to the section on that function.

Function	Mandatory variables	Purpose	Multiple pollutants	type option
calcPercentile	date, one numeric variable	calculate percentiles for numeric variables in a data frame	NA	NA
corPlot	a data frame	correlation matrix with conditioning	yes	yes [1]
cutData	a data frame	partition data into groups for conditioning plots and analysis	yes	yes [≥1]
importADMS	an ADMS output file e.g. .pst, .met, .mop, .bgd	import outputs from the ADMS suite of dispersion models (McHugh et al. 1997)	NA	NA
importAirbase	site code	import data from the EEA *airbase* database	NA	NA
importAURN	site code and year	import hourly data from the UK air quality data archive (`http://www.airquality.co.uk/data_and_statistics.php`)	NA	NA
importKCL	site code and year	import hourly data from the London Air data archive (`http://www.londonair.org.uk/LondonAir/Default.aspx`)	NA	NA
importTraj	site code and year	import HYSPLIT back trajectory data from KCL servers	NA	NA
quickText	a string	properly format common pollutant names and units	NA	NA
selectByDate	date and one other variable	flexibly select date periods by year, day of week etc.	NA	NA
selectRunning	date and one other variable	select contiguous periods of a certain run-length above a specified threshold	NA	NA
splitByDate	date and one other variable	partition and label a data frame by time periods	NA	NA
timeAverage	date, one numeric variable	calculate statistics over flexible time periods account for data capture rates etc.	NA	NA

8.4 Input data requirements

The **openair** package applies certain constraints on input data requirements. **It is important to adhere to these requirements to ensure that data are correctly formatted for use in openair.** The principal reason for insisting on specific input data format is that there will be less that can go wrong and it is easier to write code for a more limited set of conditions.

The **openair** package requires as an input a data frame, which generally consists of hourly date/time, pollution and meteorological data. As shown elsewhere in this document, the recommended way of inputting data into R is through reading a .csv file. This in itself avoids potential issues with 'awkward' file formats e.g. with varying header lines. Of course, anyone familiar with R will know how to do this and may choose to import their data from a range of sources such as databases. A few important requirements and advice are given below.

Use openair functions to help import data!

There are several functions in (§9) that make the process of importing data into **openair** much simpler. Where possible, these functions should be used. (§9) also contains some useful functions for manipulating data.

1. Data should be in a 'rectangular' format i.e. columns of data with a header on the first line. The file 'example data long.csv' provides a template for the format and users should refer to that file if in doubt. The best approach is to use the `import` function that is part of **openair**, described in (§9).

2. Where fields should have numeric data e.g. concentrations of NO_x, then the user should ensure that no other characters are present in the column, accept maybe something that represents missing data e.g. 'no data'. Even here, it is essential to tell the `import` function how missing data are represented; see (§9).

3. The date/time field should be given the heading `date` — note the lower case. No other name is acceptable.

4. The wind speed and wind direction should be named `ws` and `wd`, respectively (note again, lower case). There is an implicit assumption that wind speed data are in units of m s^{-1}. Most functions have been written assuming reasonable ranges in wind speed in m s^{-1}. However, the functions will work if the units were in knots, for example and several functions allow the user to annotate the plots with the correct units. Wind directions follow the UK Met Office format and are represented as degrees from north e.g. 90° is east. North is taken to be 360°.

5. Other variables names can be upper/lower case *but should not start with a number*. If column names do have white spaces, R will automatically replace them with a full-stop. While 'PM2.5' as a field name is perfectly acceptable, it is a pain to type it in—better just to use 'pm25' (**openair** will recognise pollutant names like this and automatically format them as $PM_{2.5}$ in plots).

Note if users wish to assume non-zero wind speeds to be calm e.g. any wind speed below 0.5 m s^{-1}, then these can be set directly e.g.

```
mydata$ws[mydata$ws < 0.5] <-  0
```

It should be mentioned again that any reasonably large amount of data should be kept in a database and not Excel sheets and the like. Much less will go wrong if this is the case; see §(5.12) for some information on Access databases.

8.4.1 Dealing with more than one site

In many situations users will have more than one site available and most **openair** functions can deal with this situation. However, it does require that the data are in a certain format. If the data are available via the AURN archive or via the KCL LAQN then it is possible to use the `importAURN` or `importKCL` functions to select multiple sites at once and the data will be correctly formatted for use by the functions.

If it is not possible to import the data in this way, it is necessary to format the data in such a way that can be used. The format is very similar to that described above for several pollutants at a single site. With more than one site it is necessary to have another column (with name `site`) with the site name in. Data are therefore 'stacked'.

Sometimes data will not be in this format and site data will be in separate columns. (§31.7.2) shows the approach that can be used to format such data.

If users need help with formatting their data, please contact us for advice.

8.5 Using colours

Type colors() or colours() into R to see full list of named colours

Many of the functions described require that colour scales are used; particularly for plots showing surfaces. It is only necessary to consider using other colours if the user does not wish to use the default scheme, shown at the top of Figure 8.1. The choice of colours does seem to be a vexing issue as well as something that depends on what one is trying to show in the first place. For this reason, the colour schemes used in **openair** are very flexible: if you don't like them, you can change them easily. R itself can handle colours in many sophisticated ways; see for example the `RColorBrewer` package.

Several pre-defined colour schemes are available to make it easy to plot data. In fact, for most situations the default colour schemes should be adequate. The choice of colours can easily be set; either by using one of the pre-defined schemes or through a user-defined scheme. More details can be found in the **openair** `openColours` function. Some of the defined colours are shown in Figure 8.1, together with an example of a user defined scale that provides a smooth transition from yellow to blue. The code that produced this plot is shown for Figure 8.1:[9]

The user-defined scheme is very flexible and the following provides examples of its use. In the examples shown next, the `polarPlot` function is used as a demonstration of their use.

```
# use default colours - no need to specify
polarPlot(mydata)

# use pre-defined "jet" colours
polarPlot(mydata, cols = "jet")

# define own colours going from yellow to green
polarPlot(mydata, cols = c("yellow", "green"))

# define own colours going from red to white to blue
polarPlot(mydata, cols = c("red", "white", "blue"))
```

[9] This is given for interest, the user does not need to know this to use the colours.

```
library(openair)
## small function for plotting
printCols <- function(col, y) {
    rect((0:200) / 200, y, (1:201) / 200, y + 0.1, col = openColours(col, n = 201),
         border = NA)
    text(0.5, y + 0.15, deparse(substitute(col)))
}

## plot an empty plot
plot(1, xlim = c(0, 1), ylim = c(0, 1.6), type = "n", xlab = "", ylab = "",
     axes = FALSE)
printCols("default", 0)
printCols("increment", 0.2)
printCols("heat", 0.4)
printCols("jet", 0.6)
printCols("hue", 0.8)
printCols("brewer1", 1.0)
printCols("greyscale", 1.2)
printCols(c("tomato", "white", "forestgreen" ), 1.4)
```

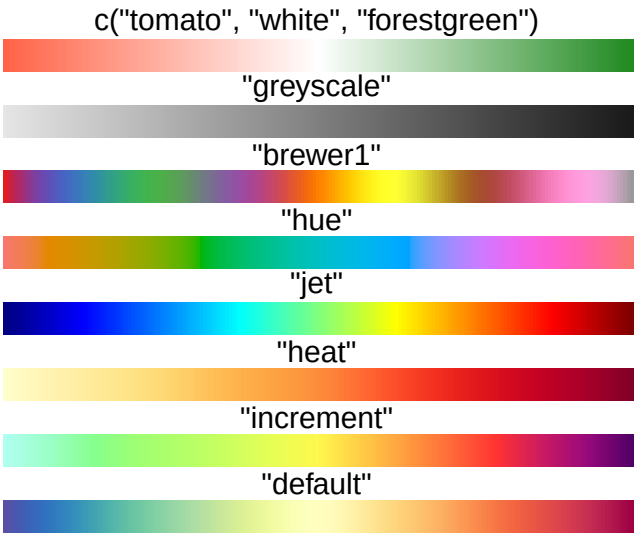


FIGURE 8.1 Pre-defined colour scales in **openair**. The top colour scheme is a user-defined one.

8.6 Automatic text formatting

openair will increasingly try to automate the process of annotating plots. It can be time consuming (and tricky) to repetitively type in text to represent µg m^{-3} or PM_{10} (µg m^{-3}) etc. in R. For this reason, an attempt is made to automatically detect strings such as 'nox' or 'NOx' and format them correctly. Where a user needs a y-axis label such as NO_x (µg m^{-3}) it will only be necessary to type `ylab = "nox (ug/m3)"`. The same is also true for plot titles.

Over time we will add to the number of text strings that could be automatically formatted. It is suggested that users get in touch if they have a specific request that is not yet covered. Most functions have an option called `auto.text` that is set to TRUE by default. Users can override this option by setting it to FALSE.

Note that there will be occasions when the user will want to format the text themselves, as shown by the examples in Table 5.1. In this case the option `auto.text = FALSE` should be set when using a function and the user should supply their own expression.

8.7 Multiple plots on a page

We often get asked how to combine multiple plots on one page. Recent changes to **openair** makes this a bit easier. Note that because **openair** uses `lattice` graphics the base graphics `par` settings will not work.

It is possible to arrange plots based on a column × row layout. Let's put two plots side by side (2 columns, 1 row). First it is necessary to assign the plots to a variable:

```
a <- windRose(mydata)
b <- polarPlot(mydata)
```

Now we can plot them using the split option:

```
print(a, split = c(1, 1, 2, 1))
print(b, split = c(2, 1, 2, 1), newpage = FALSE)
```

In the code above for the 'split' option, the last two numbers give the overall layout (2, 1) — 2 columns, 1 row. The first two numbers give the column/row index for that particular plot. The last two numbers remain constant across the series of plots being plotted.

There is one difficulty with plots that already contain sub-plots such as `timeVariation` where it is necessary to identify the particular plot of interest (see the `timeVariation` help for details). However, say we want a polar plot (`b` above) and a diurnal plot:

```
c <- timeVariation(mydata)
print(b, split = c(1, 1, 2, 1))
print(c, split = c(2, 1, 2, 1), subset = "hour", newpage = FALSE)
```

For more control it is possible to use the `position` argument. `position` is a vector of 4 numbers, c(xmin, ymin, xmax, ymax) that give the lower-left and upper-right corners of a rectangle in which the plot is to be positioned. The coordinate system for this rectangle is [0–1] in both the x and y directions.

As an example, consider plotting the first plot in the lower left quadrant and the second plot in the upper right quadrant:

```
print(a, position = c(0, 0, 0.5, 0.5), more = TRUE)
print(b, position = c(0.5, 0.5, 1, 1))
```

The position argument gives more fine control over the plot location.

8.8 Annotating openair plots

A frequently asked question about **openair** and requested feature is how to annotate plots. While all **openair** functions could have options to allow annotations to be made, this would make the functions cumbersome and reduce flexibility. Nevertheless it is useful to be able to annotate plots in lots of different ways. Fortunately there are existing functions in packages such as `lattice` and `latticeExtra` that allow for plots to be updated. An example of the sorts of annotation that are possible is shown in Figure 8.2, which is an enhanced version of Figure 17.1. These annotations have been subsequently added to Figure 17.1 and built up in layers. This section considers how to annotate **openair** plots more generally and uses Figure 8.2 as an example of the types of annotation possible. Also considered specifically is the annotation of plots that are in polar coordinates, as these can sometimes benefit from different types of annotation.

```
## Loading required package: RColorBrewer
```

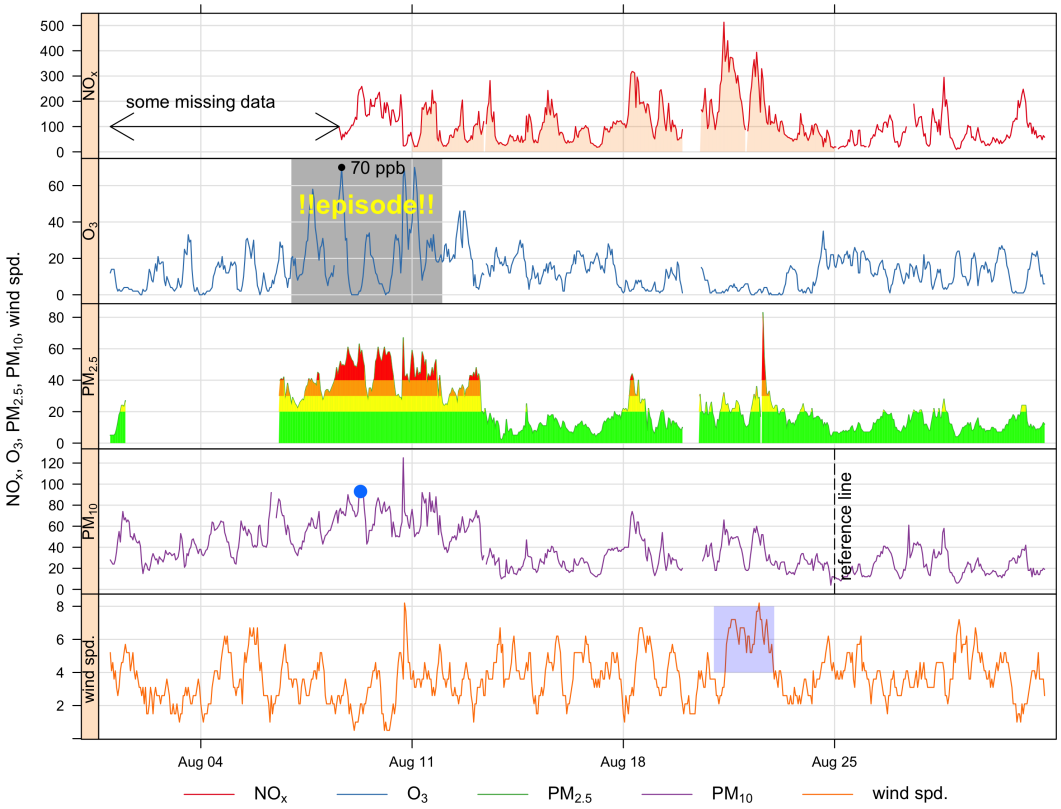


FIGURE 8.2 Examples of different ways of annotating a plot in **openair**.

There are several different types of objects that can be useful to add to plots including text, shapes, lines and other shading. Given that many **openair** plots can consist of multiple panels, it is also useful to think about how to annotate specific panels. The examples given in this section will apply to all **openair** plot, the only difference being the coordinate system used in each case.

The basis of **openair** annotations is through the use of the `latticeExtra` package, which should already be installed as part of **openair**. In that package there is a function called `layer` that effectively allows annotations to be built up 'layer by layer'.

Adding text

To add text (or other annotations) it is necessary to know the coordinates on a plot for where the text will go, which will depend on the data plotted. In this extended example using the `timePlot` function, the y-axis will be in ordinary numerical units, whereas the x-axis will be in a date-time format (`POSIXct`).

There are various ways that annotations can be added, but the method used here is to add to the previous plot using a function called `trellis.last.object()` to which we want to add a later. This may seem complicated, but once a few examples are considered, the method becomes very powerful, flexible and straightforward. In a multi-panel plot such as Figure 8.2 it is also useful to specify which rows/columns should be added to. If they are not specified then the annotation will appear in all panels.

First, a plot should be produced to which we wish to add some text.

```
## make sure latticeExtra is loaded
library(latticeExtra)
timePlot(selectByDate(mydata, year = 2003, month = "aug"),
         pollutant = c("nox", "o3", "pm25", "pm10", "ws"))
```

So, considering Figure 8.2, this is how the text 'some missing data' was added to the top panel.

```
trellis.last.object() +
    layer(ltext(x = as.POSIXct("2003-08-04"), y = 200,
                labels = "some missing data"), rows = 1)
```

So what does this do? First, the `trellis.last.object()` is simply the last plot that was plotted. Next the `layer` function is used to add some text. The text itself is added using the `ltext` (lattice) function. It is worth having a look at the help for `ltext` as that gives an overview of all the common annotations and other options. We have chosen to plot the text at position x = '2003-08-04' and y = 200 and the label itself. A useful option to `ltext` is `pos`. Values can be 1, 2, 3 and 4, and indicate positions below (the default), to the left of, above and to the right of the specified coordinates

Adding text and a shaded area

This time we will highlight an interval in row 2 (O_3) and write some text on top. Note that this time we use the `lpolygon` function and choose to put it under everything else on the plot. For the text, we have chosen a colour (yellow) font type 2 (bold) and made it a bit bigger (`cex = 1.5`). Note also the 'y' values extend beyond the actual limits shown on the plot — just to make sure they cover the whole region.

The polygon could of course be horizontal and more than one producing a series of 'band' e.g. air quality indexes. A more sophisticated approach is shown later for $PM_{2.5}$.

```
## add shaded polygon
trellis.last.object() +
    layer(lpolygon(x = c(as.POSIXct("2003-08-07"),
                  as.POSIXct("2003-08-07"), as.POSIXct("2003-08-12"),
                  as.POSIXct("2003-08-12")), y = c(-20, 600, 600, -20),
                  col = "grey", border = NA), under = TRUE, rows = 2)
## add text
trellis.last.object() +
    layer(ltext(x = as.POSIXct("2003-08-09 12:00"), y = 50,
                labels = "!!episode!!", col = "yellow",
                font = 2, cex = 1.5), rows = 2)
```

The small shaded, semi-transparent area shown in the bottom panel was added as follows:

```
## add shaded polygon
plt <- plt +
    layer(lpolygon(x = c(as.POSIXct("2003-08-21"), as.POSIXct("2003-08-21"),
                  as.POSIXct("2003-08-23"), as.POSIXct("2003-08-23")),
                  y = c(4, 8, 8, 4), col = "blue", border = NA,
                  alpha = 0.2), rows = 5)
```

Adding an arrow

The arrow shown on the first panel of Figure 8.2 was added as follows. Note the `code = 3` placed arrows at both ends. Note that `angle` is the angle from the shaft of the arrow to the edge of the arrow head.

```
trellis.last.object() +
    layer(larrows(as.POSIXct("2003-08-01"), 100,
                  as.POSIXct("2003-08-08 14:00"),
                  100, code = 3, angle = 30), rows = 1)
```

Adding a reference line and text

This code adds a vertical dashed reference line shown in the 4th panel (PM_{10}) along with some text aligned at 90 degrees using the `srt` option of `ltext`.

```
trellis.last.object() +
    layer(panel.abline(v = as.POSIXct("2003-08-25"), lty = 5),
          rows = 4)
trellis.last.object() +
    layer(ltext(x = as.POSIXct("2003-08-25 08:00"), y = 60,
                labels = "reference line", srt = 90), rows = 4)
```

Highlight a specific point

Up until now annotations have been added using arbitrary coordinates in each panel. What if we wanted to highlight a particular point, or more generally work with the actual data that are plotted. Knowing how to refer to existing data greatly extends the power of these functions.

It is possible to refer to a specific point in a panel simply by indexing the point of interest i.e. `x`, `y`. For example, to mark the 200th PM_{10} concentration (without knowing the actual date or value):

```
## add a specfic point
trellis.last.object() +
    layer(lpoints(x[200], y[200], pch = 16, cex = 1.5),
          rows = 4)
```

What if we wanted to highlight the maximum O_3 concentration? It is possible to work out the index first and then use that to refer to that point. Note the '`;`' to allow for the code to span multiple commands.

```
## add a point to the max O3 concentration
trellis.last.object() +
    layer({maxy <- which.max(y);
           lpoints(x[maxy], y[maxy], col = "black", pch = 16)},
          rows = 2)

## label max ozone
trellis.last.object() +
    layer({maxy <- which.max(y);
           ltext(x[maxy], y[maxy], paste(y[maxy], "ppb"),
                 pos = 4)}, rows = 2)
```

Add a filled polygon

It can be seen in the top panel of Figure 8.2 that some of the data are highlighted by filling the area below the line. This approach can be useful more generally in plotting. While it is possible to draw polygons easily and refer to the data itself, there needs to be a way for dealing with gaps in data, otherwise these gaps could be filled in

perhaps unpredictable ways. A function has been written to draw a polygon taking into account gaps (`poly.na`).

```
poly.na <- function(x1, y1, x2, y2, col = "black", alpha = 0.2) {
  for(i in seq(2, length(x1)))
    if (!any(is.na(y2[c(i - 1, i)])))
      lpolygon(c(x1[i - 1], x1[i], x2[i], x2[i - 1]),
               c(y1[i - 1], y1[i], y2[i], y2[i - 1]),
               col = col, border = NA, alpha = alpha)
}
```

This time we work out the ids of the data spanning an area of interest. Then the `poly.na` function is used. Note that the alpha transparency is by default 0.2 but another value can easily be supplied, as shown in the air quality 'bands' example.

```
trellis.last.object() +
    layer({id <- which(x >= as.POSIXct("2003-08-11") &
                       x <= as.POSIXct("2003-08-25"));
           poly.na(x[id], y[id], x[id], rep(0, length(id)),
                   col = "darkorange")}, rows = 1)
```

Add air quality bands as polygons

It is a simple extension to go from using a polygon below the data to polygons at certain intervals e.g. air quality indexes. These are shown for $PM_{2.5}$ and the bands considered are 0–20, 20–30, 30–40 and >40.

```
trellis.last.object() +
    layer(poly.na(x, y, x, rep(0, length(x)),
                  col = "green", alpha = 1), rows = 3)
trellis.last.object() +
    layer(poly.na(x, ifelse(y <20, NA, y), x,
                  rep(20, length(x)), col = "yellow", alpha = 1),
          rows = 3)
trellis.last.object() +
    layer(poly.na(x, ifelse(y <30, NA, y),
                  x, rep(30, length(x)),
                  col = "orange", alpha = 1), rows = 3)
trellis.last.object() +
    layer(poly.na(x, ifelse(y <40, NA, y),
                  x, rep(40, length(x)),
                  col = "red", alpha = 1), rows = 3)
```

Polar plot examples

Many of the examples considered above are relevant to all other functions e.g. how to add text, choosing rows and columns to plot in. Polar coordinate plots are different because of the coordinate system used and this section considers a few examples.

One useful approach is to be able to draw an arc, perhaps highlighting an area of interest. A simple, but flexible function has been written to do this. It takes arguments `theta1` and `theta2` that define the angular area of interest and `lower` and `upper` to set the lower and upper wind speed, respectively. It also has additional arguments `theta3` and `theta4` which optionally set the angles for the 'upper' wind speed.

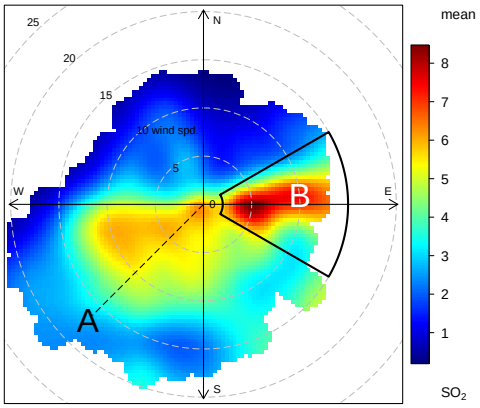


FIGURE 8.3 `polarPlot` for SO_2 with annotations.

```
arc <- function(theta1 = 30, theta2 = 60, theta3 = theta1, theta4 = theta2,
                lower = 1, upper = 10){
  ## function to work out coordinates for an arc sector
  if (theta2 < theta1) {
    ang1 <- seq(theta1, 360, length = abs(theta2 - theta1))
    ang2 <- seq(0, theta2, length = abs(theta2 - theta1))
    angles.low <- c(ang1, ang2)

    ## for upper angles
    ang1 <- seq(theta1, 360, length = abs(theta4 - theta3))
    ang2 <- seq(0, theta2, length = abs(theta4 - theta3))
    angles.high <- c(ang1, ang2)

  } else {
    angles.low <- seq(theta1, theta2, length = abs(theta2 - theta1))
    angles.high <- seq(theta3, theta4, length = abs(theta4 - theta3))
  }
  x1 <- lower * sin(pi * angles.low / 180)
  y1 <- lower * cos(pi * angles.low / 180)
  x2 <- rev(upper * sin(pi * angles.high / 180))
  y2 <- rev(upper * cos(pi * angles.high / 180))
  data.frame(x = c(x1, x2), y = c(y1, y2))

}
```

Following on from the previous examples, some annotations have been added to the basic polar plor for SO_2 as shown in Figure 8.3. Note that in these plots (0, 0) is the middle of the plot and the radial distance will be determined by the wind speed — or whatever the radial variable is. This way of plotting arcs can also be applied to other functions that show directional data.

```
polarPlot(mydata, pollutant = "so2", col = "jet")
trellis.last.object() + layer(ltext(-12, -12, "A", cex = 2))
trellis.last.object() + layer(ltext(10, 2, "B", cex = 2, col = "white"))
trellis.last.object() + layer(lsegments(0, 0, -11.5, -11.5, lty = 5))
## add and arc to highlight area of interest
trellis.last.object() +
    layer(lpolygon(x = arc(theta1 = 60, theta2 = 120, lower = 2,
                  upper = 15)$x, y = arc(theta1 = 60,
                                         theta2 = 120, lower = 2,
                                         upper = 15)$y, lty = 1, lwd = 2))
```

Using grid graphics — identify locations interactively

The examples above provide a precise way of annotating plots for single or multi-panels **openair** displays. However, these methods won't work for plots that consist of completely separate plots such as the four plots in `timeVariation`. There are however other methods that can be used to annotate such plots using the package `grid`, which forms the basis of `lattice` graphics. There is enormous capability for annotating plots using the `grid` package and only a few simple examples are given here.

Given a plot such as Figure 21.1, how could texts be added at any location — say in the middle monthly plot? One very useful function for this type of annotation that allows the user to interactively choose a location is the `grid.locator()` function in the `grid` package. That function can be called with different coordinate systems — but the one we want defines the bottom-left corner as (0, 0) and the top right as (1, 1).

First of all, make a `timeVariation` plot like Figure 21.1.

```
timeVariation(mydata)
```

Now let's choose a location on the plot interactively using the mouse and selecting somewhere in the middle of the monthly plot.

```
library(grid)
## bring up the interactive location chooser
grid.locator(unit="npc")
```

What should happen is that in the R console the coordinates are given for that point. In my case these were x = 0.503 and y = 0.338. These coordinates can now be used as the basis of adding some text or other annotation. In the example below, the `grid.text` function is used to add some text for these coordinates making the font bigger (`cex = 2`), bold (`font = 2`) and blue (`col = "blue"`).

```
grid.text(x = 0.503, y = 0.338, label = "here!",
          gp = gpar(cex = 2, font = 2, col = "blue"))
```

Even with this basic approach, some sophisticated annotation is possible with *any* **openair** plot. There are many other functions that can be used from the grid package that would allow for polygons, segments and other features to be drawn is a similar way to the examples earlier in this section. Continuing with the same example, here is how to add an arrow pointing to the maximum NO_x concentration shown on the top plot for Saturday (again using the `grid.locator` function).

```
grid.lines(x = c(0.736, 0.760), y = c(0.560, 0.778),
           arrow = arrow())
grid.text(x = 0.736, y = 0.560, label = "maximum", just = "left")
```

8.9 Getting help

The principal place for seeking help with **openair** functions is through the software itself. The document you are reading will increasingly give the background to the ideas and wider information. Also, the package itself will always contain the most up to date help. Furthermore, the process of building and checking packages is strict. For example, it is checked to see if all the options in a function match with descriptions in the help files, and all examples given in the help (and there are many) are run to ensure they all work. Nevertheless, the options shown for each function in this document

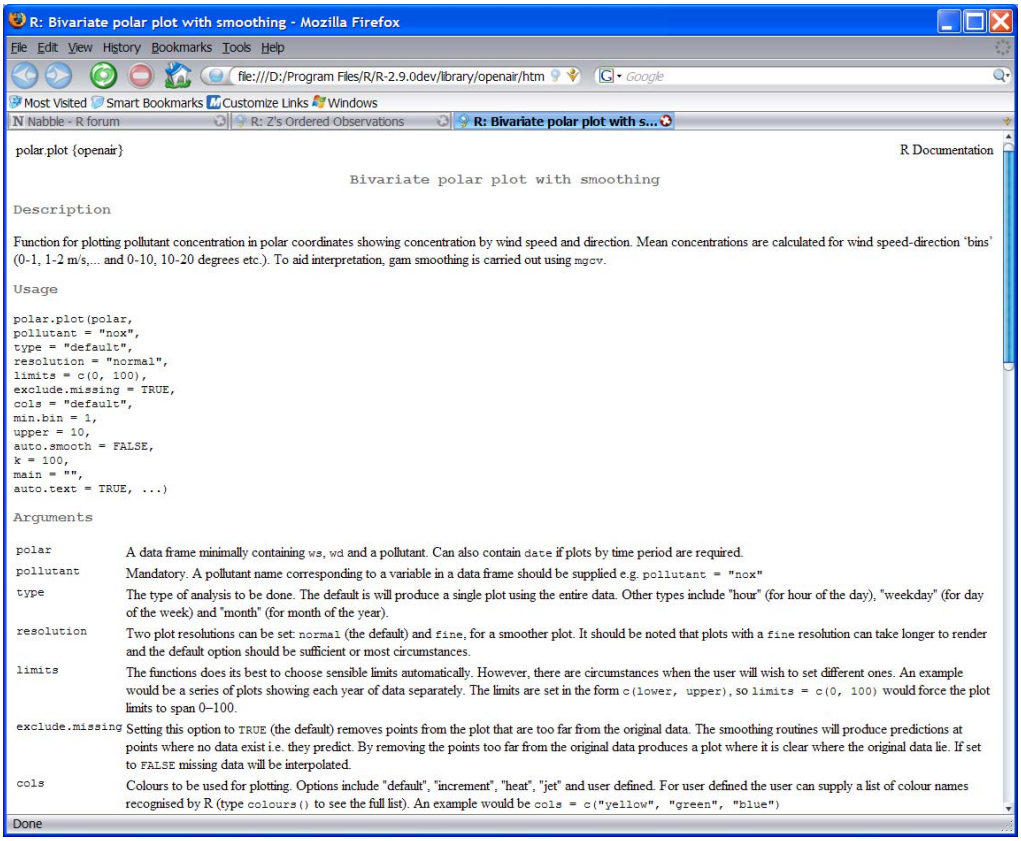


FIGURE 8.4 The help screen for the function `polarPlot`.

are parsed directly from the **openair** package ensuring consistency between this document and the package help. To bring up the general help page (assuming you have loaded **openair**), type `?openair`, which will bring up the main **openair** page, from which there are links to all functions. Similarly, if you want help with a specific function more directly, type something like `?polarPlot`.

The help screen will provide the most up to date information on the function including: a short description, a description of all the options, a more detailed description and links to other similar functions. Importantly, each function help will have several examples given of its use, which are easily reproducible; just copy them into R. These examples use the data set 'example data long.csv' mentioned previously.

Handy tip: use TAB for word completion

If you are typing directly into R you do not always need to type the whole word of a function or option. Taking the `calendarPlot` function as an example, type 'calen' then press TAB and it will complete the whole string 'calendarPlot'. Similarly, when typing the function options such as 'pollutant', just type the first few lines 'poll', press TAB and it will complete as 'pollutant='. This makes R much quicker to work with. It takes a bit of experimentation to get a feel for how many letters are required before a unique function name or option can be completed.

9 Getting data into openair

Importing data is usually the first step involved in data analysis using **openair**. As has been stressed before, the key issue is ensuring the data are in a simple format avoiding any unnecessary formatting. For this reason data are best stored either in a database or a .csv file. R itself has lots of capabilities for importing data and these will be useful in many situations e.g. `read.table` and `read.csv`. However, **openair** has several dedicated functions to make data import easier for users, as well as some more specific functions for particular data types. These are described below.

9.1 Issues related to time zones

Time zones can very difficult to work with in R and other software. Issues related to Daylight Savings Time (DST) can be particularly challenging (see `http://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones` for information in World time zones). In summary. **openair** assumes times that do not include DST. To do this assumptions are made about times when data are imported to R using the **openair** `import` function and users should take care if importing data in other ways.

Ensuring certainty in time zones used

It is recommended that when importing data into **openair** using either the `import` function or by other means that the time zone of the original data are set to GMT or a fixed offset from GMT as outlined in this section. This way of importing data and setting the time zone should avoid any complexities related to Daylight Savings Time.

If the **openair** `import` function is used then the user can set the time zone for the original data. By default 'GMT' is assumed. For non-GMT time zones a GMT offset can be assumed. The option `tzone` (see below) can be used to control the time zone. 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.*

Similarly the time zone should be set in the same way if the data are imported manually. For example:

```
dat <- read.csv("~/openair/Data/example data long.csv", header = TRUE)
## set time zone to equivalent of CET but with no DST:
dat$date <- as.POSIXct(strptime(mydata$date, format = "%d/%m/%Y %H:%M",
                                tz = "Etc/GMT-1"))
```

For most functions these issues will not matter. However, if users are including data from other sources or are working with data that are GMT/UTC (Hysplit back trajectories for example), then it will be important to understand how to deal with these issues.

It may also be easier to set your R session to a non-DST time zone in a similar way to make sure everything is displayed in a consistent format e.g.

```
Sys.setenv(TZ = "Etc/GMT-1")
```

9.2 The import function

A flexible function import has been written to import .csv or .txt file data and format the date/time correctly. The main purpose of this function is to help format dates etc. for use in **openair** and R. This is the principal means by which most users should import data unless the data are from UK networks. It is simple to use with its default assumptions e.g. header on the first line and a column 'date' in the format dd/mm/yyyy HH:MM:

```
mydata <- import()
```

Typing this into R will bring up an 'open file' dialog box, from which you can choose a .csv file. Try importing the 'example data long.csv' file in this was to see how it works. Used without any options like this, it assumes that the date field is in the format dd/mm/yyyy HH:MM and is called 'date'. By default the time zone is set to be GMT. However, users can apply a GMT offset as described in Section 9.1.

Often it is better to supply the file path because this makes the analysis more reproducible e.g.

```
mydata <- import("d:/temp/my interesting data.csv")
```

The import function is actually very flexible and can take account of different date formats, header lines etc. See the options below. For most users, few if any of these options will need to be used, but for 'difficult' data, the flexibility should be helpful. One option that is often useful is to tell R how missing data are represented. If the fields are left blank, they will automatically be set to NA. However, it may be that the file identifies missing data by 'NoData', or '-999'. In this case, import should be called like:

```
mydata <- import(na.strings = "NoData")
```

or

```
mydata <- import(na.strings = "-999")
```

In the case of missing data being represented by several strings e.g. '-99.0' and '-999', it should be called like

```
mydata <- import(na.strings = c("-99.0", "-999"))
```

It is essential to supply the import function with details of how missing data are represented if they are not represented by either a blank cell or NA. This is because if text is present in a column that should be numeric, then R will consider that the column is a character and not numeric. When using the import function, details of the format of each field are printed in R. The user can check that fields that should be numeric appear as either 'numeric' or 'integer' and not 'character' or 'factor'.

Another example is a file that has separate date and time fields e.g. a column called 'mydate' and a separate column called 'mytime'. Further, assume that date is in the format dd.mm.YYYY e.g. 25.12.2010, and time is in the format HH:MM. Then the file could be imported as:

```
import("c:/temp/test.csv", date = "mydate", date.format = "%/d.%m.%Y",
       time = "mytime", time.format = "%H:%M")
```

What if the date was in the format mm.dd.YYYY?:

```
import("c:/temp/test.csv", date = "mydate", date.format = "%/m.%d.%Y",
       time = "mytime", time.format = "%H:%M")
```

...and the time was just the hour as an integer (0–23):

```
import("c:/temp/test.csv", date = "mydate", date.format = "%/m.%d.%Y",
       time = "mytime", time.format = "%H")
```

Another common situation is that hour is represented as 1–24 in a date-time field. In this case it is necessary to correct for this. R stores `POSIXct` format as seconds, so 3600 need to be subtracted to ensure the time is correct. Note that if there is a separate column for hour then `import` will correct that automatically. So, for the date-time situation:

```
import("c:/temp/test.csv", date = "mydate", date.format = "%/m.%d.%Y %H",
       correct.time = -3600)
```

Note if time was expressed as HH:MM:ss, then the option `time.format = "%H:%M:%S"` should be used.

There are other options for ignoring the first *n* lines i.e. due to header information and so on. The user can specify the header line row (`header.at`) and the row the data starts at (`data.at`).

Note also that `import` assumes there are no daylight saving time (DST) issues in the original data i.e. a missing hour in spring and a duplicate hour in autumn. Dealing with these issues in R rapidly gets too complicated ... users should therefore ensure the original data do not consider DST.

The options for the `import` function are:

`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`.

9.3 The `importAURN` function

While `import` is a useful function for ad-hoc data import, much of the data stored in the UK and beyond resides on central repositories that are available over the Internet. The UK AURN archive and King's College London's London Air Quality Network (LAQN) are two important and large databases of information that allow free public access. Storing and managing data in this way has many advantages including consistent data format, and underlying high quality methods to process and store the data. We are working with AEA and KCL to make things easier to link with **openair** functions.

Many users download hourly data from the air quality archive at `http://www.airquality.co.uk`. Most commonly, the data are emailed to the user as .csv files and have a fixed format as shown below. This is a useful facility but does have some limitations and frustrations, many of which have been overcome using a new way of storing and downloading the data described below.

The `importAURN` function has been written to make it easy to import data from the UK AURN. AEA have provided .RData files (R workspaces) of all individual sites and years for the AURN. These files are updated on a daily basis. This approach requires a link to the Internet to work.

There are several advantages over the web portal approach where .csv files are downloaded. First, it is quick to select a range of sites, pollutants and periods (see examples below). Second, storing the data as .RData objects is very efficient as they are about four times smaller than .csv files (which are already small) — 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 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. Type ?importAURN for a full listing of sites and their codes.

Note that currently there is no meteorological data associated with the archive. To use the fill flexibility of the functions it is recommended that the AURN data are combined with appropriate local meteorological data. See (§5.3) for more details on how to combine separate air pollution and meteorological files.

The function has the following options.

`site` Site code of the AURN 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 specfic 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")`.

`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.

Some examples of usage are shown below.

```
## import all pollutants from Marylebone Rd from 1990:2009
mary <- importAURN(site = "my1", year = 2000:2009)

## import nox, no2, o3 from Marylebone Road and Nottingham Centre for 2000
thedata <- importAURN(site = c("my1", "nott"), year = 2000,
                      pollutant = c("nox", "no2", "o3"))

## import over 20 years of Mace Head O3 data!
o3 <- importAURN(site = "mh", year = 1987:2009)
## import hydrocarbon data from Marylebone Road
hc <- importAURN(site = "my1", year = 2008, hc = TRUE)
```

In future, **openair** functions will recognise AURN data and capture units, thus enabling plots to be automatically annotated. Furthermore, there is the potential to include lots of other 'meta data' such as site location, site type etc., which will be added to the function in due couse.

9.4 The `importKCL` function

King's College London also make available their data in a similar way to the `importAURN`. One difference compared to the `importAURN` is the availability of meteorological data in London, which is accessible through the option `met`. We have provided a 'typical' meteorological data set representing London, which is a composite of data from several instruments co-located with air pollution monitoring sites. Access to reliable

meteorological data can be difficult and expensive, and this is an issue we hope to improve in time.

The options for `importKCL` are:

`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.

Examples of importing data are given in the help files as part of **openair**. However, the example below shows how to import data from the Bexley 1 site (code BX1), together with the meteorological data.

Like the `importAURN` function the selection is only possible by site code (all the site codes and full site descriptions are shown in the help file). In time we will include other information such as site location and type and develop the functions to make it easy to use these other fields.

Below is an example of importing data from the Bexley 1 site.

```
bx1 <- importKCL(site = "bx1", year = 2000:2009, met = TRUE)
```

in which case a dialog box will appear prompting the user for a file location. The data frame `mydata` is now ready for use in **openair**.

9.5 Importing and working with data from the European Environment Agency *airbase* database

The European Environment Agency (EEA) makes available hourly and pre-calculated annual statistics for air pollution data from across Europe (see `http://acm.eionet.europa.eu/databases/airbase/`) for approximately 8000 sites. The EEA go to great lengths to compile, check and make available a huge amount of air quality data. The EEA provide a range of interactive maps and make all data available as csv files. These csv files are split by country and can be very large. The database currently used in **openair** is version 8.

The aim of the `importAirbase` function is to provide an alternative and hopefully complementary approach to accessing *airbase* data with a specific focus on integration

with R and the **openair** package. There are two main sets of data that can be imported using **openair**. First, there is detailed hourly data by site and for all available years (accessible using the `importAirbase` function). Second, there are pre-calculated *airbase* annual type statistics (accessible though the `airbaseStats` function). The former represents potentially very large detailed files that are useful for analysing data in detail for relatively few sites. The pre-calculated statistics are much more compact (but still surprisingly large; the 2012 data file is > 3 million rows ...). The pre-calculated statistics are better for considering air pollution at a country or European scale and are useful for mapping annual mean concentrations for example.

Similar to other import functions in openair (such as `importAURN` and `importKCL`), the `importAirbase` function works with sites and combines all species into one data frame. Rather than having year-specific files there is only one file (data frame) per site covering all years. However, users can select the years they require at the import stage.

There are many potential issues that need to be dealt with, although for the most part everything should be compiled in a straightforward way. One of the key issues is the use of different instrument techniques measuring the same species at a site, or an instrument that was replaced at some point. The EEA usefully record this information. Rather than attempt to combine several potential time series for the same pollutant, they have been kept separate. Examples include these use of different methods to measure PM_{10} e.g. TEOM and FDMS. Because different instruments can provide very different concentrations it is probably wise to keep them separate and analyse them as separate species. In other cases e.g. ozone or NO_2, if an instrument was replaced half way through a time series it would be reasonable to combine the time series into a single set. There is a function `airbaseSplice` that will combine pollutants once imported using `importAirbase`.

The `importAirbase` function

The primary function for importing hourly *airbase* data is the `importAirbase` function. In common with many other **openair** import functions a connection to the Internet is required for these functions to work. Rather than save either the data or meta data in the package itself all data are stored on a web server. The advantage if this approach is it keeps the package small and enables the data to be updated centrally. The key requirement is to provide a site code or codes to identify the sites of interest. The `airbaseFindCode` function described below is useful for finding sites of interest. Users may also pre-select the pollutant(s) of interest. The help file for `airbaseStats` lists all the pollutant names.

The user can also elect to add other fields to the returned results. By default, the country and site type are returned but other fields can be chosen such as `city`, `site` (site name), `lat` and `lon` (the latitude and longitude of the sites). There is also an option `splice` the will simplify the returned results when more than one measurement exists for a species (see the `airbaseSplice`) function below for details of what this option does.

The `importAirbase` function has the following arguments:

`site` Site code(s) of the sites to be imported. Can be upper or lower case.

`year` The year or years of interest. For example to select 2010 to 2012 use `year = 2010:2012`.

`pollutant` The pollutant(s) to be selected. See the list in `airbaseStats`.

`add` Additional fields to add to the returned data frame. By default the country and site type are returned. Other useful options include "city", "site" (site name), "EMEP_station", "lat", "lon" and "altitude".

`splice` Should the pollutant fields be consolidated when multiple measurements of individual pollutants are available? See `airbaseSplice` for details.

`local` Used for tesing local imports.

To import data for the North Kensington site for example (code GB0620A):

```
kc1.airbase <- importAirbase(site = "gb0620a")
head(kc1.airbase)
```

```
##      code                date                  site BS  CO NO NO2 NOX O3
## 1 GB0620A 1996-04-01 00:00:00 LONDON N. KENSINGTON NA 0.7 NA  80 120  4
## 2 GB0620A 1996-04-01 01:00:00 LONDON N. KENSINGTON NA 0.4 NA  63  69  8
## 3 GB0620A 1996-04-01 02:00:00 LONDON N. KENSINGTON NA 0.4 NA  NA  NA 14
## 4 GB0620A 1996-04-01 03:00:00 LONDON N. KENSINGTON NA 0.2 NA  42  42 24
## 5 GB0620A 1996-04-01 04:00:00 LONDON N. KENSINGTON NA 0.2 NA  36  38 32
## 6 GB0620A 1996-04-01 05:00:00 LONDON N. KENSINGTON NA 0.4 NA  52  61 24
##   PM10|100 PM2.5 SO2 PM10|101 PM10|102        country  site.type
## 1       30    NA  27       NA       NA United Kingdom Background
## 2       25    NA  NA       NA       NA United Kingdom Background
## 3       23    NA  16       NA       NA United Kingdom Background
## 4       20    NA  13       NA       NA United Kingdom Background
## 5       17    NA  NA       NA       NA United Kingdom Background
## 6       20    NA  NA       NA       NA United Kingdom Background
```

The `airbaseFindCode` function

To make it easier to find different *airbase* sites the `airbaseFindCode` has been written. This function returns the *airbase* site code(s) as a string that can be used directly in `importAirbase` or to help refine selections in `airbaseStats`. The function allows the user to select sites based on common selection criteria (if there are other criteria you would like added then please get in touch). These criteria include the country code (e.g. "GB", "DE", "DK"), the site type (background, traffic, industrial or unknown), the area type (rural, urban, suburban or unknown), the latitude and longitude range and local site code. The options are explained below.

`country` A character or vector of characters representing country code.

`site.type` One of "background", "traffic", "industrial", "unknown" representing the type of site.

`area.type` The type of area in which the site is located. Can be "rural", "urban", "suburban", "unknown".

`local.code` A character or vector of characters representing the local site code. For example "MY1" is the UK code for Marylebone Road.

`city` A city name to search — using character matching (`grep`). The search string can be upper or lower case e.g. `city = "london"`. To extract several cities e.g. Copenhagen and Barcelona use `city = c("copenhagen", "barcelona")`. Note that by default any matching characters are returned, so `city = "london"` would also return Londonderry (Northern Ireland).

Regular expression searches are very powerful and potentially complicated. However there are a few useful tips. To match the *beginning* of a name use '^'. So `city = "^london"` would return London and Londonderry (both begin with 'london'). To match the end of a name use '$', so `city = "london$"` would just return London but not Londonderry.

The cities chosen are printed to screen to make it easy to check (and refine the search string) of the selected sites.

`site` The name of the site or sites to search, see `city` for details of how to search.

`emep` Select an EMEP station. Can be "yes", "no" or `NA` (the default, selects everything).

`lat` The latitude range to select in the form c(lower, upper).

`lon` The longitude range to select in the form c(lower, upper).

To list the site codes of traffic sites in Denmark for example:

```
sites <- airbaseFindCode(country = "dk", site.type = "traffic")
sites
```

```
##  [1] "DK0002A" "DK0004A" "DK0008A" "DK0009A" "DK0010A" "DK0021A" "DK0037A"
##  [8] "DK0034A" "DK0030A" "DK0016A" "DK0036A" "DK0007A" "DK0003A" "DK0017A"
## [15] "DK0006A" "DK0013A" "DK0022A" "DK0031A" "DK0018A" "DK0001A" "DK0012A"
## [22] "DK0014A" "DK0015A" "DK0011A" "DK0005A" "DK0051A"
```

If one wanted to import all hourly data for these sites (lots of data) then:

```
dk.traffic <- importAirbase(site = sites)
```

Another example is to find background sites in the UK and Germany (about 1500 sites ...):

```
sites <- airbaseFindCode(country = c("DE", "GB"), site.type = "background")
```

The `city` option is a little different because it can be trickier to select specific cities. However, it is possible to construct sophisticated text searches. The search string can be upper or lower case e.g. `city = "london"`. It is also possible to use regular expressions. For example, to extract sites in Copenhagen and Barcelona use `city = c("copenhagen", "barcelona")`. Note that by default any matching characters are returned, so `city = "london"` would also return Londonderry (Northern Ireland).

Regular expression searches are very powerful and potentially difficult to understand. However there are a few useful tips. To match the beginning of a name use `^`. So `city = "^london"` would return London and Londonderry (both begin with 'london'). To match the end of a name use `$`, so `city = "london$"` would just return London but not Londonderry. These two examples alone will probably be sufficient to isolate almost any city of interest.

It is easy to quickly find sites by name. For example, Marylebone Road can be searched even with a partial string. Note that `airbaseFindCode` will print the site name to screen so that it is possible to check the results. In this way sites can be chosen in an interactive, iterative manner where search strings can be refined as necessary.

```
airbaseFindCode(site = "maryle")

##      code                   site
## 1 GB0682A LONDON MARYLEBONE ROAD
## [1] "GB0682A"
```

The `airbaseInfo` function

Another useful utility function is `airbaseInfo`. This function takes an *airbase* site code(s) and returns detailed information concerning the site and instruments. Of particular use is information on the instrument type used. Because the returned data set can be quite large (particularly if several sites are selected) it is best to read the data info a variable. By default the function only returns one row per site with basic information such as site type, latitude and longitude.

`code` Site code(s) of the sites to be imported. Can be upper or lower case.

`instrument` Should species/instrument details also be returned. When `FALSE` only one row per site is returned with other information such as site type, latitude and logitude. When `TRUE` details of the individual species and instruments used is also returned.

For example, to find information on the London North Kensington site:

```
airbaseInfo(code = "gb0620a")

##          code                 site        country   city  site.type      lat
## 37508 GB0620A LONDON N. KENSINGTON United Kingdom LONDON Background 51.52106
##             lon altitude
## 37508 -0.213431        5
```

For detailed information on the species measured and instruments used:

```
kcInfo <- airbaseInfo(code = "gb0620a", instrument = TRUE)
head(kcInfo)
```

```
##          code pollutant                            poll.name
## 37508 GB0620A       SO2                Sulphur dioxide (air)
## 37509 GB0620A      PM10 Particulate matter < 10 um (aerosol)
## 37510 GB0620A      PM10 Particulate matter < 10 um (aerosol)
## 37511 GB0620A      PM10 Particulate matter < 10 um (aerosol)
## 37512 GB0620A      PM10 Particulate matter < 10 um (aerosol)
## 37513 GB0620A        BS                    Black smoke (air)
##       measurement_european_group_code
## 37508                             100
## 37509                             100
## 37510                             101
## 37511                             102
## 37512                             110
## 37513                             100
##                                      technique
## 37508                        UV fluorescence
## 37509 Tapered Element Oscillating Microbalance (TEOM)
## 37510 Tapered Element Oscillating Microbalance (TEOM)
## 37511          Volatile Correction Model (VCM)
## 37512                    Gravimetric analysis
## 37513                            Aethalometry
##                              equipment sam.height  unit                  site
## 37508                          UNKNOWN          0 ug/m3 LONDON N. KENSINGTON
## 37509                        TEOM 1400          0 ug/m3 LONDON N. KENSINGTON
## 37510 TEOM 1400AB with FDMS module (8500)          0 ug/m3 LONDON N. KENSINGTON
## 37511                        TEOM 1400          0 ug/m3 LONDON N. KENSINGTON
## 37512                         Partisol          0 ug/m3 LONDON N. KENSINGTON
## 37513                     Aethalometer          0 ug/m3 LONDON N. KENSINGTON
##              country   city  site.type      lat       lon altitude
## 37508 United Kingdom LONDON Background 51.52106 -0.213431        5
## 37509 United Kingdom LONDON Background 51.52106 -0.213431        5
## 37510 United Kingdom LONDON Background 51.52106 -0.213431        5
## 37511 United Kingdom LONDON Background 51.52106 -0.213431        5
## 37512 United Kingdom LONDON Background 51.52106 -0.213431        5
## 37513 United Kingdom LONDON Background 51.52106 -0.213431        5
```

The `airbaseSplice` function

Airbase keeps a detailed account of the instruments used for measuring different species. For the same species several different instruments can be used over time. For example, for NO_x a broken instrument could be replaced by a new one. In *airbase* these two instruments will results in two separate time series identified through the *airbase* measurement_european_group_code (with numbers like 100, 101, 102). When data are imported using the `importAirbase` function the different measurement codes are retained in the pollutant names if more than one code is used, resulting in names like `NOX|101`, `NOX|102` and so on.

For 'simple' species such as NO_x there is generally no reason to keep two separate time series and it would seem reasonable to combine them into a single species `NOX`. This is what the `airbaseSplice` function does. It will combine multiple measurements of the same species into a new single species. In cases where there is overlap, more recent instrument measurements take precedence over older instrument measurements.

In other cases it can be sensible to keep the different measurements of the same species separate. An example of such a case is for PM_{10} where, for example, moving from TEOM to FDMS measurements produces time series that cannot (or should not)

be easily be merged.

Rather than combine all pollutant measurements of the same species by default it was thought important to retain this information to allow users more flexibility.

If `drop = TRUE` then the original are columns not retained. For example, given original columns `NOX|101`, `NOX|102` the new data frame will have only one column NOX. Conversely, if `drop = FALSE` then the final data frame will have three species: `NOX|101`, `NOX|102` and a new combined field `NOX`.

`dat` Data (a data frame) that has been imported using the `importAirbase` function.

`drop` Should the original columns be dropped or kept. The default is to remove (drop) them.

The `airbaseStats` function

The `airbaseStats` function imports pre-calculated (by the EEA) summary statistics from the *airbase* database. The data have been split by the statistic in question to make the file sizes more manageable. The main aim of this function is to work with single statistics and pollutants. There are several options to refine the selection criteria including pollutant (listed in the help file), the *airbase* site code (e.g. perhaps generated from `airbaseFindCode`), the site type, year(s) and data capture threshold percentage.

In the summary *airbase* statistics there are different averaging periods; the most common being 'day' and 'hour'. Often several averaging times will be available for a pollutant. For example, for PM_{10} the annual mean concentrations can be derived from averaging hourly data or daily data. Furthermore, hourly means can be used to derive an annual mean in two ways: average all the hours in the year or first produce daily means and average those (the results will be near identical apart from rounding errors). In some cases though e.g. PM_{10}, many instruments will only report daily means, which will not allow annual means to be derived based on hourly values. By default **openair** will use the statistic that has the most data associated with it.

`statistic` A *single* choice from "P50", "Mean", "P95", "P98", "Max", "Max36", "Max8", "Days.c.50.", "Max26", "Days.c.120.","SOMO35", "AOT40", "Max19", "Hours.c.200.", "Max4", "Days.c.125.", "P99_9", "Max25", "Hours.c.350.".

`pollutant` The *single* name of a pollutant to extract; can be upper or lower case but must match one of those below.

`avg.time` The averaging time on which the measurements are based. By default the averaging period with most data associated with it is used. For most pollutants the averaging period will be "day". Other common options are "hour" and "week".

`code` The airbase site code(s) to select. Can be upper or lower case.

`site.type` The type of site(s) to select.

`year` The year(s) to select e.g. `year = 2000:2012` to select years 2010, 2011 and 2012.

`data.cap` The data capture threshold to use (%). By default all data are selected.

`add` Additional fields to add to the returned data frame. By default the country, site type, latitude and logitude are returned. Other useful options include "city", "site" (site name), "EMEP_station" and "altitude".

To import annual mean NO_2 concentrations for all sites in 2012 with a data capture rate of ≥90%:

```
no2 <- airbaseStats(statistic = "Mean", pollutant = "no2", year = 2012,
                    data.cap = 90)

## Using averging time based on day

## how many sites is this?
nrow(no2)

## [1] 2484

head(no2)

##       code component_code         component_name pollutant measurement_unit
## 2  AL0201A              8 Nitrogen dioxide (air)       NO2             ug/m3
## 14 AT0ENK1              8 Nitrogen dioxide (air)       NO2             ug/m3
## 27 AT0ILL1              8 Nitrogen dioxide (air)       NO2             ug/m3
## 51 AT0PIL1              8 Nitrogen dioxide (air)       NO2             ug/m3
## 82 AT0VOR1              8 Nitrogen dioxide (air)       NO2             ug/m3
## 92 AT0ZOE2              8 Nitrogen dioxide (air)       NO2             ug/m3
##    measurement_european_group_code year statistics_average_group stat
## 2                              100 2012                      day Mean
## 14                             100 2012                      day Mean
## 27                             500 2012                      day Mean
## 51                             500 2012                      day Mean
## 82                             100 2012                      day Mean
## 92                             100 2012                      day Mean
##    statistic_name    no2 data.cap       date country      lat      lon
## 2     annual mean 41.411   95.355 2012-01-01 Albania 41.33027 19.82177
## 14    annual mean 11.391   97.268 2012-01-01 Austria 48.39167 13.67111
## 27    annual mean  8.791   99.454 2012-01-01 Austria 47.77000 16.76640
## 51    annual mean  8.836   98.087 2012-01-01 Austria 48.72111 15.94223
## 82    annual mean  3.216   96.721 2012-01-01 Austria 46.67972 12.97195
## 92    annual mean  4.510   99.180 2012-01-01 Austria 47.83861 14.44139
##     site.type                                      site   city
## 2     Traffic                          Tirana Center TIRANA
## 14 Background                 Enzenkirchen im Sauwald   <NA>
## 27 Background                                Illmitz   <NA>
## 51 Background                   Pillersdorf bei Retz   <NA>
## 82 Background            Vorhegg bei K<U+00F6>tschach-Mauthen   <NA>
## 92 Background Z<U+00F6>belboden - Reichraminger Hintergebirge   <NA>
```

9.6 Importing data from the CERC ADMS modelling systems

The ADMS suite of models is widely used in the UK and beyond. These models are used for a wide range of purposes and one of the benefits of **openair** is that many of the functions are potentially useful for model evaluation. One of the principal benefits of linking **openair** with the ADMS models is the access to meteorological data that is possible. In the UK, the Met Office provides meteorological data in a specific format for use in ADMS models.[10] It is useful to be able to easily import the meteorological data into **openair** because analyses are often limited by the availability of representative meteorological data. However, the use of directly measured input data is only one possibility. When ADMS models run they use a sophisticated meteorological pre-processor to calculate other quantities that are not directly measured, but are important to dispersion modelling. Examples of these other variables are boundary

[10] Specifically hourly sequential data and not statistical summaries of data.

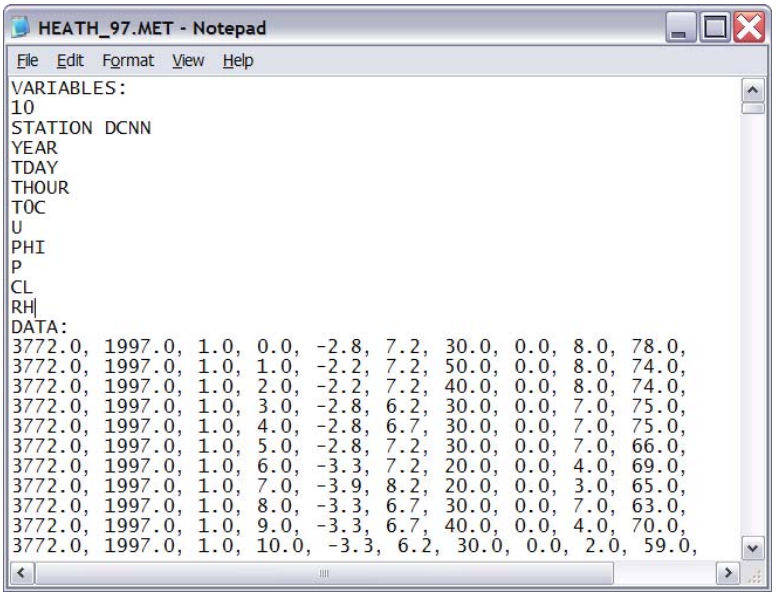


FIGURE 9.1 Typical format of an hourly ADMS met file.

layer height and surface sensible heat flux. These and many other quantities are calculated by the met pre-processor and output to a .MOP file. Access to these other quantities greatly increases the potential for model evaluation and in general provides a much richer source of information for analysis.

Many users may have meteorological data in the ADMS format. This is the format provided by the UK Met Office for the ADMS model. A an example of the format is shown in Figure 9.1, which is a simple text file. The `importADMSMet` function imports such data into R in a format suitable for **openair**.

This can be done, for example by:

```
met <- importADMS("d:/temp/heathrow01.met")
```

If no file name is supplied, the user will be prompted for one.

Sometimes it may be necessary to import several years. Here's one approach for doing so assuming the files are in a folder d:/metdata and all have a file extension.met:

```
all.met <- lapply(list.files(path = "d:/metdata", pattern =".met", full.names = TRUE),
                  function(.file) importADMS(.file))
all.met <- do.call(rbind.fill, all.met)
```

`all.met` will then contain met data for all years in one data frame.

9.6.1 An example considering atmospheric stability

One of the significant benefits of working with ADMS output files is having access to the outputs from the meteorological pre-processor. ADMS uses readily available meteorological variables such as wind speed, temperature and cloud cover and calculates parameters that are used in the dispersion algorithms. When ADMS is run it produces a .MOP file with all these inputs and processed quantities in. Access to parameters such as boundary layer height, Monin-Obukov length and so on can *greatly* increase the opportunities for insightful data analysis using existing **openair**

functions. This is almost certainly an area we will cover in more depth later; but for now, here are a few examples.

We are going to use a .MOP file from 2001 following some dispersion modelling of stacks in London. The interest here is to use the results from the met pre-processor to better understand sources in the east of London at the Thurrock background site. First, we can import the Thurrock data (type `?importKCL` for site code listing) using the `importKCL` function:

```
tk1 <- importKCL(site = "tk1", year = 2001)


## show first few lines of tk1
head(tk1)
```

```
##                      date  nox  no2 o3      so2    co pm10_raw pm10
## 41862 2001-01-01 00:00:00   NA   NA NA       NA    NA      5.2  5.2
## 41863 2001-01-01 01:00:00 7.68 5.76 50 46.40133 0.232     11.7 11.7
## 41864 2001-01-01 02:00:00 5.76 3.84 52 57.55550 0.232      7.8  7.8
## 41865 2001-01-01 03:00:00 5.76 3.84 56 14.72350 0.232      5.2  5.2
## 41866 2001-01-01 04:00:00 1.92 1.92 54 10.70800 0.232     10.4 10.4
## 41867 2001-01-01 05:00:00 3.84 1.92 54 10.70800 0.232     14.3 14.3
##                                  site code
## 41862 Thurrock - London Road (Grays)  TK1
## 41863 Thurrock - London Road (Grays)  TK1
## 41864 Thurrock - London Road (Grays)  TK1
## 41865 Thurrock - London Road (Grays)  TK1
## 41866 Thurrock - London Road (Grays)  TK1
## 41867 Thurrock - London Road (Grays)  TK1
```

Next we will import the .MOP file. The function automatically lists all the variables imported:

```
met <- importADMS("~/openair/Data/met01.MOP")
```

```
##              date1              date2               line                run
##          "POSIXct"          "POSIXt"          "integer"             "factor"
##                 fr                 ws           ws.gstar                 wd
##          "numeric"          "numeric"          "numeric"          "numeric"
##           delta.wd             ftheta0                  k          recip.lmo
##          "numeric"          "numeric"          "numeric"          "numeric"
##                  h                 nu        delta.theta               temp
##          "numeric"          "numeric"          "numeric"          "numeric"
##                  p                 cl         albedo.met        albedo.disp
##          "numeric"          "numeric"          "numeric"          "numeric"
##          alpha.met         alpha.disp               tsea            delta.t
##          "numeric"          "numeric"          "numeric"          "numeric"
##        sigma.theta                rhu                 q0            lambdae
##          "numeric"          "numeric"          "numeric"          "numeric"
##              rhu.1              drhdzu     process.ws.star        process.ws.g
##          "numeric"          "numeric"          "numeric"          "numeric"
##    process.ws.gstar       process.wd.0       process.wd.g    process.delta.wd
##          "numeric"          "numeric"          "numeric"          "numeric"
##      process.wd.sec       process.wstar     process.ftheta0          process.k
##          "numeric"          "numeric"          "numeric"          "numeric"
##   process.recip.lmo          process.h          process.nu process.delta.theta
##          "numeric"          "numeric"          "numeric"          "numeric"
##        process.temp          process.p     process.delta.t process.sigma.theta
##          "numeric"          "numeric"          "numeric"          "numeric"
##          process.q0     process.lambdae         process.rhu      process.drhdzu
##          "numeric"          "numeric"          "numeric"          "numeric"
##      process.z0.met     process.z0.disp
##          "numeric"          "numeric"
```

Now we need to merge these two files using 'date' as the common field using the `merge` function, which is part of the base R system:

```
tk1 <- merge(tk1, met, by = "date")
```

Now we have a data frame with all the pollution measurements and meteorological variables matched up. A nice first example is to make use of variables that are not readily available. In particular, those representing atmospheric stability are very useful. So, let's see what a polar plot looks like split by different levels of the atmospheric stability parameter the reciprocal of the Monin-Obukov length, $\frac{1}{L_{MO}}$. This has the name `process.recip.lmo`. Note we also set the option `min.bin = 2`, to ensure the output is not overly affected by a single high concentration.

The results are shown in Figure 9.2. So what does this tell us? Well, first $\frac{1}{L_{MO}}$ has been split into three different levels (broadly speaking the more negative the value of $\frac{1}{L_{MO}}$ the more unstable the atmosphere is and the more positive $\frac{1}{L_{MO}}$ is, the more stable the atmosphere is). In Figure 9.2 the plot shows what we might think of unstable, neutral and stable atmospheric conditions.

The first thing to note from Figure 9.2 is that lower wind speeds are associated with stable and unstable atmospheric conditions — shown by the smaller plot areas for these conditions (neutral conditions have a larger 'blob' extending to higher wind speeds). This is entirely expected. Starting with the unstable conditions (top left panel), SO_2 concentrations are dominated by easterly and south-easterly winds. These concentrations are likely dominated by tall stack emissions from those wind directions. For stable conditions (plot at the bottom), three sources seem to be important. There is a source to the north-east, the south-east and higher concentrations for very low wind speeds. The latter is likely due to road vehicle emissions of SO_2. The neutral conditions

```
polarPlot(tk1, pollutant = "so2", type = "process.recip.lmo",
          min.bin = 2)
```

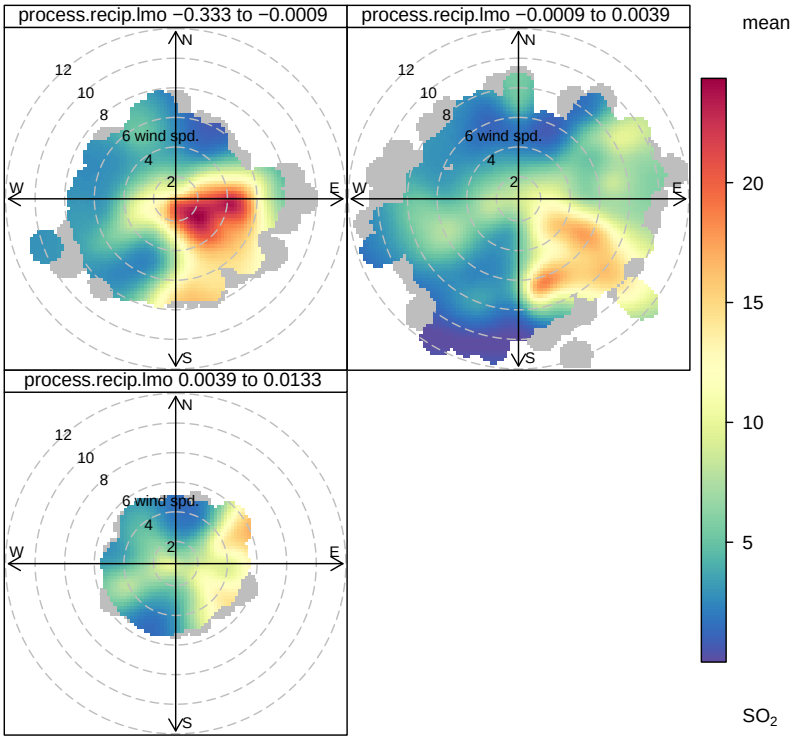


FIGURE 9.2 Use of the `importADMS` function to access atmospheric stability parameters for use in a polar plot. In this case $\frac{1}{L_{MO}}$ is split by three different levels, approximately corresponding to unstable, neutral and stable atmospheric conditions.

are perhaps revealing two sources to the south-east. Taken together, plotting the data in this way is beginning to reveal a potentially large number of sources in the area. Combined with the results from a dispersion model, or knowledge of local stacks, there is a good chance that these sources can be identified.

A polar plot on its own does not reveal such detailed information. Try it:

```
polarPlot(tk1, pollutant = "so2")
```

Of course care does need to be exercised when interpreting these outputs, but the availability of wider range of meteorological data can only improve inference.

Here are some other analyses (not plotted, but easily run). For NO_x:

```
## dominated by stable conditions and low
## wind speeds (traffic sources)
polarPlot(tk1, pollutant = "nox", type = "process.recip.lmo",
          min.bin = 2)
```

PM_{10}:

```
## complex, but dominated by stable/unstable easterly conditions
polarPlot(tk1, pollutant = "pm10", type = "process.recip.lmo",
          min.bin = 2)
```

How about the ratio of two pollutants, say the ratio of SO_2/NO_x? First calculate the

ratio:

```
tk1 <- transform(tk1, ratio = so2 / nox)
## evidence of a source with high so2/nox ratio ro the SSE
polarPlot(tk1, pollutant = "ratio", type = "process.recip.lmo",
          min.bin = 2)
```

And don't forget all the other parameters available such as boundary layer height etc. — and all the other functions in **openair** that can be used.

10 The summaryPlot function

The summaryPlot function is a way of rapidly summarising important aspects of data. While many statistical summaries are possible to calculate with R, the summaryPlot function has been written specifically for monitoring data. The function provides key graphical and statistical summaries. summaryPlot has the following options:

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 to 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.
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")`.

the `summaryPlot` function should be used for checking input data before applying other functions

It is called in a very simple way:[11]

An example of using `summaryPlot` shown in Figure 10.1. For each numerical variable in a data frame, a plot is made, shown in the left panel, showing where data exist (blue) and missing data (red). For clarity, only running sequences of $\geq$ 24 hours of missing data are shown. It is easy to see therefore that the beginning part of the time series for $PM_{2.5}$ is missing and the end part of SO_2. It is also clear that the time series stops half way through 2005. Also shown in each panel are statistical summaries, which include: number of missing points (with percentage shown in parentheses), minimum, maximum, mean, median and the 95th percentile. For each year, the data capture (%) is shown in green font. So, for example, the data capture for NO_x in 2000 was 96.3 %.

The pale yellow line gives an indication of the variation in values over time expressed as a daily mean. It is in indication because no numerical scale is given. The data are formatted so that 0 is placed at the lower part of the scale (top of the data indicator strip) and the maximum value at the top of the graph. The intention is to give the user a feel for how the data vary over the length of the time series.

The plots shown in the right panel are histograms. It is also possible to show *density plots*. A density plot is somewhat similar to a histogram but avoids having to arbitrarily select a 'bin' size. The choice of bin size in histograms can often lead to a misleading impression of how data are distributed — simply because of the bin size chosen. The default behaviour of this function 'clips' the data and excludes the highest 1 % of values. This is done to help highlight the shape of the bulk of the data and has the effect of removing the long tail, typical of air pollution concentration distributions.

It is possible, however, not to clip the histogram or density plot data and select various other options:

[11] Note that a data frame `mydata` is automatically loaded when loading the **openair** package. The data set consists of several years of pollutant data from Marylebone Road in London.

```
library(openair) # load openair
data(mydata) ## make sure data that comes with openair is loaded
summaryPlot(mydata)
```

```
##     date1     date2        ws        wd       nox       no2        o3      pm10
##  "POSIXt" "POSIXct" "numeric" "integer" "integer" "integer" "integer" "integer"
##       so2        co      pm25
## "numeric" "numeric" "integer"
```

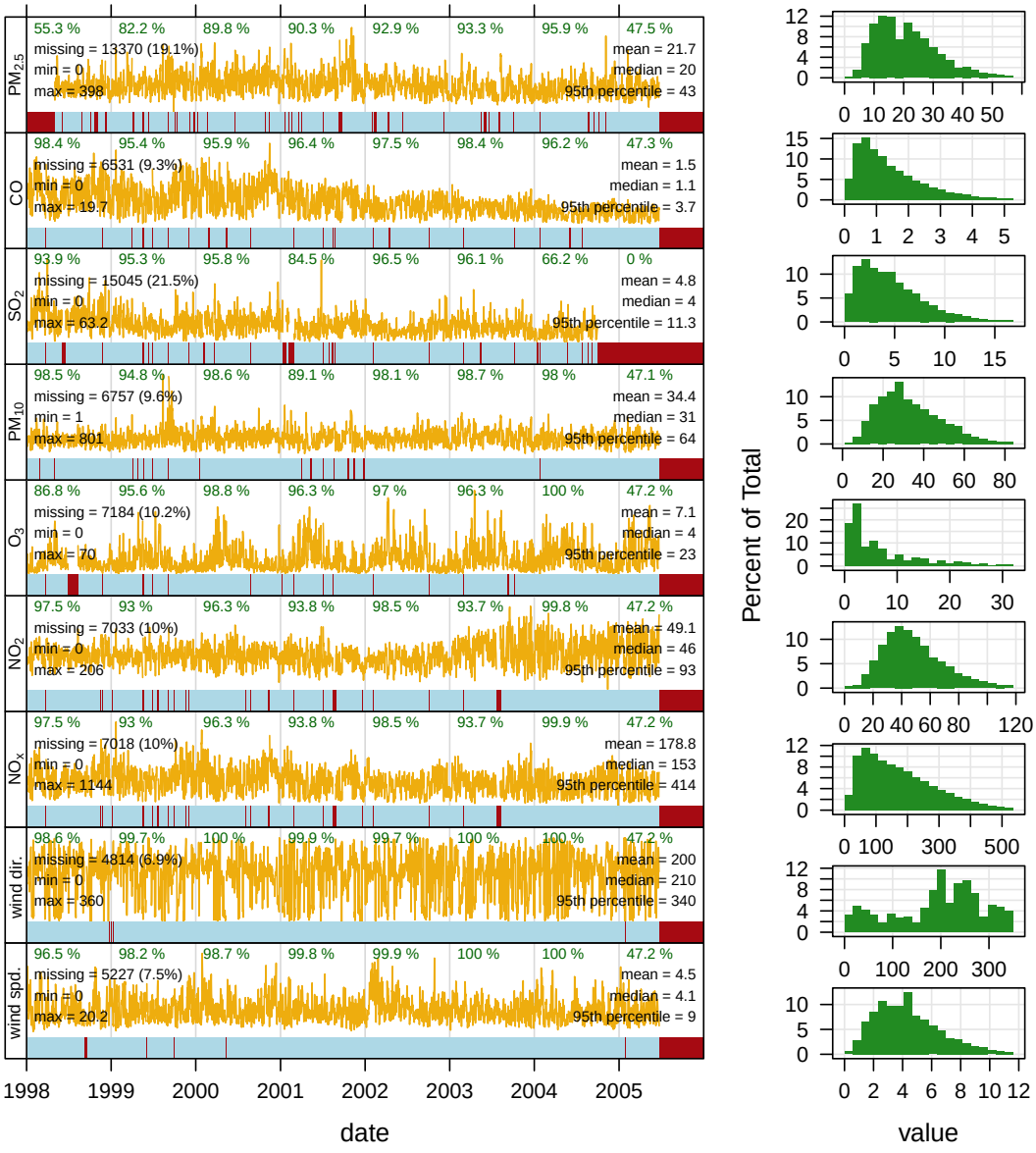


FIGURE 10.1 Use of `summaryPlot` function applied to the `mydata` data frame. The plots in the left panel show the time series data, where blue shows the presence of data and red missing data. The daily mean values are also shown in pale yellow scaled to cover the range in the data from zero to the maximum daily value. As such, the daily values are indicative of an overall trend rather than conveying quantitative information. For each pollutant, the overall summary statistics are given. For each year the percentage data capture is shown in green font. The panels on the right show the distribution of each species using a histogram plot.

```
summaryPlot(mydata, clip = FALSE) # do not clip density plot data

summaryPlot(mydata, percentile = 0.95) # exclude highest 5 % of data etc.

# show missing data where there are at least 10 continuous missing values
summaryPlot(mydata, na.len = 10)

summaryPlot(mydata, col.data = "green") # show data in green

summaryPlot(mydata, col.mis = "yellow") # show missing data in yellow

summaryPlot(mydata, col.dens = "black") # show density plot line in black
```

Depending on the data available, there may be too many plots shown on one page, making it difficult to see the detail. Currently, the simplest way to reduce what is shown is to limit the data to be plotted. In the code below, for example, only columns 2 and 5 to 7 are plotted (column 1 in this case is the date and must always be supplied). Alternatively, the `subset` function could be used:

```
summaryPlot(mydata[, c(1, 2, 5:7)]) # only plot columns 2 and 5-7
summaryPlot(subset(mydata, select = c(date, nox, no2, co))) # alternative selecting
```

So far the `summaryPlot` function has been described and used in terms of plotting many variables from a single site. What happens if there is more than one site? Because the plot already produces a dense amount of information it does not seem sensible to plot several variables across several sites at the same time. Therefore, if there is a `site` field, `summaryPlot` will provide summary data for *a single pollutant* across all sites. See `?summaryPlot` for more details.

Use `summaryPlot` first

It is recommended that the `summaryPlot` function is used before moving on to using the other functions detailed below. One of the reasons (apart from getting to know your data) is that it also acts as a way of ensuring that other functions should work. For example, if wind speed was missing, or was formatted as a character rather than a number, it will not show up in the summary plot. In time we intend to use this function to carry out many data checks and issue warnings if problems are detected.

11 The `cutData` function

The `cutData` function is a utility function that is called by most other functions but is useful in its own right. It's main use is to partition data in many ways, many of which are built-in to **openair**.

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.

For example, to cut data into seasons:

```
mydata <- cutData(mydata, type = "season")
head(mydata)
```

```
##                  date   ws  wd nox no2 o3 pm10    so2      co pm25       season
## 1 1998-01-01 00:00:00 0.60 280 285  39  1   29 4.7225  3.3725   NA winter (DJF)
## 2 1998-01-01 01:00:00 2.16 230  NA  NA NA   37     NA      NA   NA winter (DJF)
## 3 1998-01-01 02:00:00 2.76 190  NA  NA  3   34 6.8300  9.6025   NA winter (DJF)
## 4 1998-01-01 03:00:00 2.16 170 493  52  3   35 7.6625 10.2175   NA winter (DJF)
## 5 1998-01-01 04:00:00 2.40 180 468  78  2   34 8.0700  8.9125   NA winter (DJF)
## 6 1998-01-01 05:00:00 3.00 190 264  42  0   16 5.5050  3.0525   NA winter (DJF)
```

This adds a new field 'season' that is split into four seasons. There is an option `hemisphere` that can be used to use southern hemisphere seasons when set as `hemisphere = "southern"`.

The `type` can also be another field in a data frame e.g.

```
mydata <- cutData(mydata, type = "pm10")
head(mydata)
```

```
##                  date   ws  wd nox no2 o3          pm10    so2      co pm25
## 1 1998-01-01 00:00:00 0.60 280 285  39  1 pm10 22 to 31 4.7225  3.3725   NA
## 2 1998-01-01 01:00:00 2.16 230  NA  NA NA pm10 31 to 44     NA      NA   NA
## 3 1998-01-01 02:00:00 2.76 190  NA  NA  3 pm10 31 to 44 6.8300  9.6025   NA
## 4 1998-01-01 03:00:00 2.16 170 493  52  3 pm10 31 to 44 7.6625 10.2175   NA
## 5 1998-01-01 04:00:00 2.40 180 468  78  2 pm10 31 to 44 8.0700  8.9125   NA
## 6 1998-01-01 05:00:00 3.00 190 264  42  0  pm10 1 to 22 5.5050  3.0525   NA
```

```
data(mydata) ## re-load mydata fresh
```

This divides PM_{10} concentrations into four *quantiles* — roughly equal numbers of PM_{10} concentrations in four levels.

Most of the time users do not have to call `cutData` directly because most functions have a `type` option that is used to call `cutData` directly e.g.

```
polarPlot(mydata, pollutant = "so2", type = "season")
```

However, it can be useful to call `cutData` *before* supplying the data to a function in a few cases. First, if one wants to set seasons to the southern hemisphere as above. Second, it is possible to override the division of a numeric variable into four quantiles by using the option `n.levels`. More details can be found in the `cutData` help file.

The `cutData` function has the following options:

`x` A data frame containing a field `date`.

`type` A string giving the way in which the data frame should be split. Predefined 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.

`...` All additional parameters are passed on to next function(s). For example, with `cutData` all additional parameters are passed on to `cutDaylight` allowing direct access to `cutDaylight` via either `cutData` or any `openair` using `cutData` for `type` conditioning.

`local.hour.offset,latitude,longitude` Parameters used by `cutDaylight` to estimate if the measurement was collected during daylight or nighttime hours. `local.hour.offset` gives the measurement timezone and `latitude` and `longitude` give the measurement location. NOTE: The default settings for these three parameters are the London Marylebone Road AURN site associated with the `mydata` example data set. See `...{}` and Details below for further information.

12 The windRose and pollutionRose functions

12.1 Purpose

see also polarFreq percentileRose

The wind rose is a very useful way of summarising meteorological data. It is particularly useful for showing how wind speed and wind direction conditions vary by year. The `windRose` function can plot wind roses in a variety of ways: summarising all available wind speed and wind direction data, plotting individual wind roses by year, and also by month. The latter is useful for considering how meteorological conditions vary by season.

Data are summarised by direction, typically by 45 or 30° 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.

The `windRose` function also calculates the percentage of 'calms' i.e. when the wind speed is zero. UK Met Office data assigns these periods to 0 degrees wind direction with valid northerly winds being assigned to 360 degrees.

The `windRose` function will also correct for bias when wind directions are rounded to the nearest 10 degrees but are displayed at angles that 10 degrees is not exactly divisible into e.g. 22.5 degrees. When such data are binned, some angles i.e. N, E, S, W will comprise of three intervals whereas others will comprise of two, which can lead to significant bias. This issue and its solution is discussed by Droppo and Napier (2008) and Applequist (2012).[12] **openair** uses a simple method to correct for the bias by globally rescaling the count in each wind direction bin by the number of directions it

[12] Thanks to Philippe Barnéoud of Environment Canada for pointing this issue out.

represents relative to the average. Thus, the primary four directions are each reduced by a factor of 0.75 and the remaining 12 directions are multiplied by 1.125.

12.2 Options available

The `windRose` function has the following options:

`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`.

`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.

`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.

```
windRose(mydata)
```

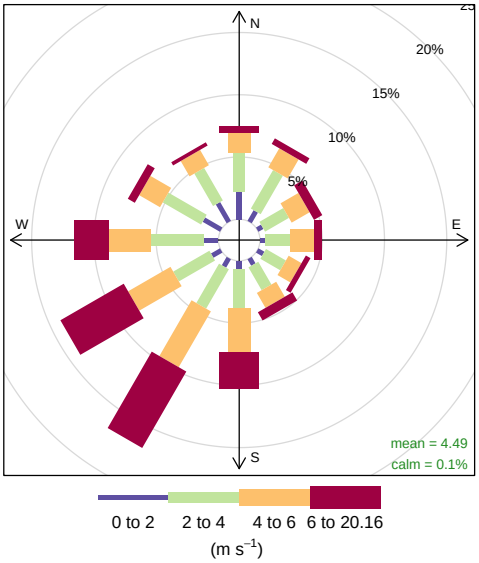


FIGURE 12.1 Use of `windRose` function to plot wind speed/direction frequencies. Wind speeds are split into the intervals shown by the scale in each panel. The grey circles show the % frequencies.

`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.

`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.

12.3 Example of use

The function is very simply called as shown for Figure 12.1.

Figure 12.2 highlights some interesting differences between the years. In 2000, for example, there were a large number of occasions when the wind was from the SSW and 2003 clearly had more occasions when the wind was easterly. It can also be useful to use `type = "month"` to get an idea of how wind speed and direction vary seasonally.

The `type` option is very flexible in **openair** and can be used to quickly consider the dependencies between variables. Section 11 describes the basis of this option in **openair** plot. As an example, consider the question: what are the meteorological conditions that control high and low concentrations of PM_{10}? By setting `type = "pm10"`, **openair** will split the PM_{10} concentrations into four *quantiles* i.e. roughly equal numbers of points in each level. The plot will then show four different wind roses for each quantile level, although the default number of levels can be set by the user — see

```
windRose(mydata, type = "year", layout = c(4, 2))
```

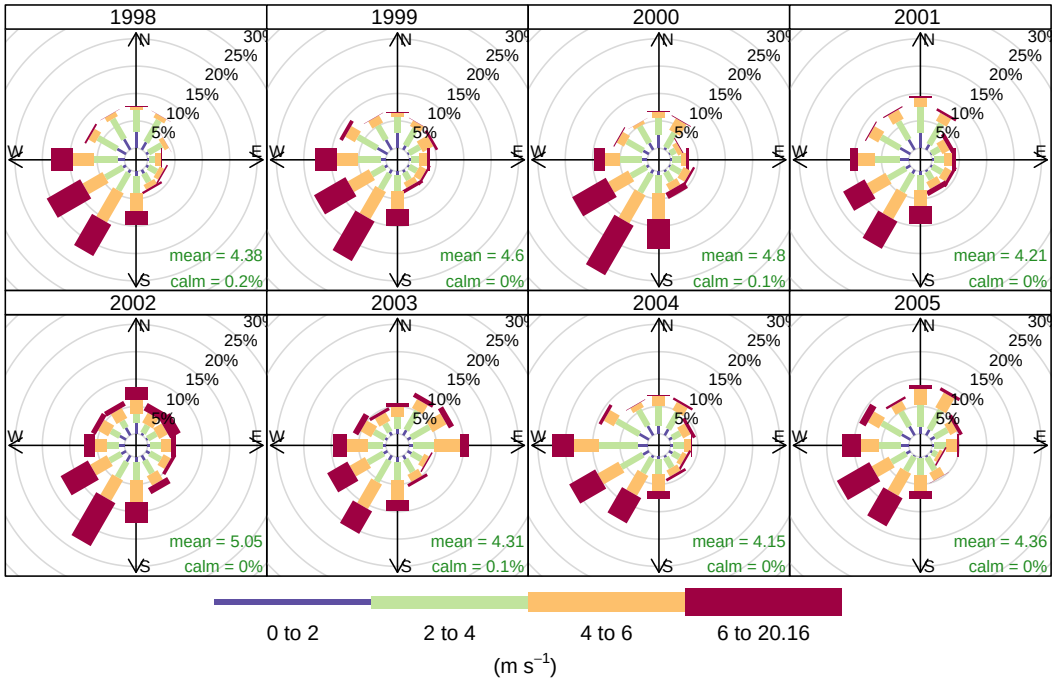


FIGURE 12.2 Use of `windRose` function to plot wind speed/direction frequencies by year. Wind speeds are split into the intervals shown by the scale in each panel. The grey circles show the 10 and 20 % frequencies.

`?cutData` for more details. Figure 12.3 shows the results of setting `type = "pm10"`. For the lowest concentrations of PM_{10} the wind direction is dominated by northerly winds, and relatively low wind speeds. By contrast, the highest concentrations (plot furthest right) are dominated by relatively strong winds from the south-west. It is therefore very easy to obtain a good idea about the conditions that tend to lead to high (or low) concentrations of a pollutant. Furthermore, the `type` option is available in almost all **openair** functions.

A comparison of the effect that bias has can be seen by plotting the following. Note the prominent frequencies for W, E and N in particular that are due to the bias issue discussed by Applequist (2012).

```
## no bias correction
windRose(mydata, angle = 22.5, bias.corr = FALSE)

## bias correction (the default)
windRose(mydata, angle = 22.5)
```

`pollutionRose` is a variant of `windRose` that is useful for considering pollutant concentrations by wind direction, or more specifically the percentage time the concentration is in a particular range. This type of approach can be very informative for air pollutant species, as demonstrated by Ronald Henry and co-authors in a recent paper (Henry et al. 2009).

You can produce similar pollution roses using the `pollutionRose` function in recent versions of **openair**, e.g. as in Figure 12.4:

`pollutionRose` is *wrapper* for `windRose`. It simply replaces the wind speed data

```
windRose(mydata, type = "pm10", layout = c(4, 1))
```

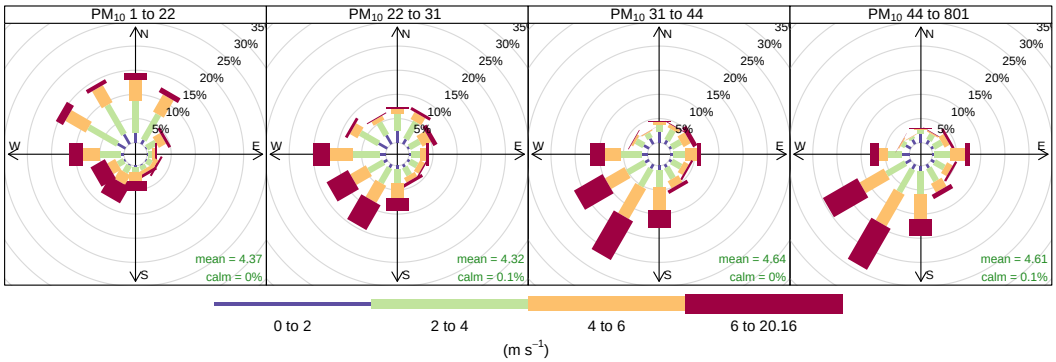


FIGURE 12.3 Wind rose for four different levels of PM_{10} concentration. The levels are defined as the four *quantiles* of PM_{10} concentration and the ranges are shown on each of the plot labels.

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

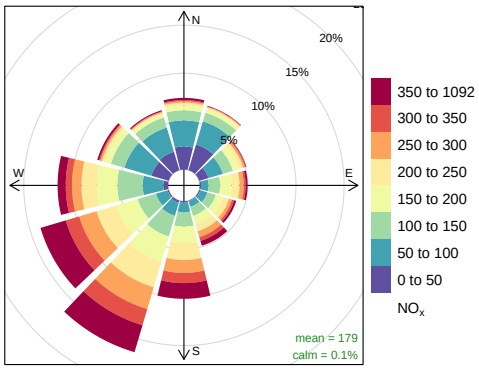


FIGURE 12.4 NO_x pollution rose produced using `pollutionRose` and default `pollutionRose` settings.

series in the supplied data set with another variable using the argument `pollutant` before passing that on to `windRose`. It also modifies `breaks` to estimate a sensible set of break points for that pollutant and uses a slightly different set of default options (key to right, wedge style plot) but otherwise handles arguments just like the parent `windRose` function.

While Figure 12.4 indicates that most higher NO_x concentrations are also associated with the SW, conditioning allows you to be much informative. For example, conditioning by SO_2 (Figure 12.5) demonstrates that higher NO_x concentrations are associated with the SW and much of the higher SO_2 concentrations. However, it also highlights a notable NO_x contribution from the E, most apparent at highest SO_2 concentrations that is obscured in Figure 12.4 by a relatively high NO_x background (Figure 12.5).

`pollutionRose` can also usefully be used to show which wind directions dominate the overall concentrations. By supplying the option `statistic = "prop.mean"` (proportion contribution to the mean), a good idea can be gained as to which wind directions contribute most to overall concentrations, as well as providing information on the different concentration levels. A simple plot is shown in Figure 12.6, which

```
pollutionRose(mydata, pollutant = "nox", type = "so2", layout = c(4, 1))
```

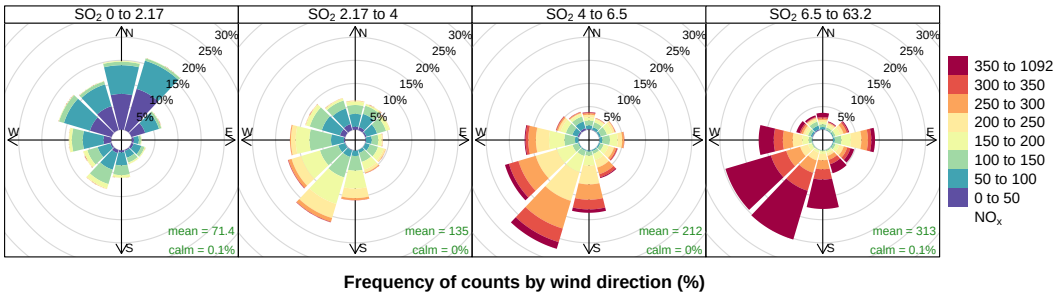


FIGURE 12.5 NO_x pollution rose conditioned by SO_2 concentration.

```
pollutionRose(mydata, pollutant = "nox", statistic = "prop.mean")
```

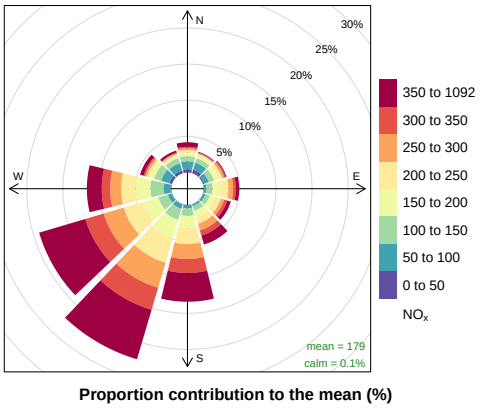


FIGURE 12.6 Pollution rose showing which wind directions contribute most to overall mean concentrations.

clearly shows the dominance of south-westerly winds controlling the overall mean NO_x concentrations at this site. Indeed, almost half the overall NO_x concentration is contributed by two wind sectors to the south-west. The `polarFreq` function can also show this sort of information, but the pollution rose is more effective because both length and colour are used to show the contribution. These plots are very useful for understanding which wind directions control the overall mean concentrations.

Comparing two meteorological data sets

The `pollutionRose` function is also useful for comparing two meteorological data sets. In this case a 'reference' data set is compared with a second data set. There are many reasons for doing so e.g. to see how one site compares with another or for meteorological model evaluation (more on that in later sections). 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.

```
## $example of comparing 2 met sites
## first we will make some new ws/wd data with a postive bias
mydata <- transform(mydata,
                    ws2 = ws + 2 * rnorm(nrow(mydata)) + 1,
                    wd2 = 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", grid.line = 5)
```

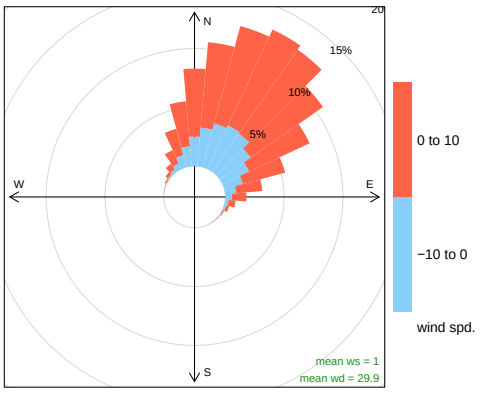


FIGURE 12.7 Pollution rose showing the difference between two meteorolgical data sets. The colours are used to show whether data tend to be postively or negatively biased with respect to the reference data set.

Note that this plot is mostly aimed at showing wind direction biases. It does also show the wind speed bias *but only if there is a wind direction bias also*. However, in most practical situations the plot should show both wind speed and direction biases together. An example of a situation where no wind speed bias would be shown would be for westerly winds where there was absolutely no bias between two data sets in terms of westerly wind direction but there was a difference in wind speed. Users should be aware of this limitation.

In the next example, some artificial wind direction data are generated by adding a positive bias of 30 degrees with some normally distributed scatter. Also, the wind speed data are given a positive bias. The results are shown in Figure 12.7. The Figure clearly shows the mean positive bias in wind direction i.e. the direction is displaced from north (no bias). The colour scale also shows the extent to which wind speeds are biased i.e. there is a higher proportion of positively biased wind speeds shown by the red colour compared with the negatively biased shown in blue. Also shown in Figure 12.7 is the mean wind speed and direction bias as numerical values.

Note that the `type` option can be used in Figure 12.7 e.g. `type = "month"` to split the analysis in useful ways. This is useful if one wanted to see whether a site or the output from a model was biased for different periods. For example, `type = "daylight"` would show whether there are biases between nighttime and daytime conditions.

An example of using user-supplied breaks is shown in Figure 12.8. In this case six intervals are chosen including one that spans −0.5 to +0.5 that is useful to show wind speeds that do not change.

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

pollutionRose(mydata, ws = "ws", wd = "wd", ws2 = "ws2", wd2 = "wd2",
              breaks = c(-11, -2, -1, -0.5, 0.5, 1, 2, 11),
              cols = c("dodgerblue4", "white", "firebrick"),
              grid.line = 5, type = "daylight")
```

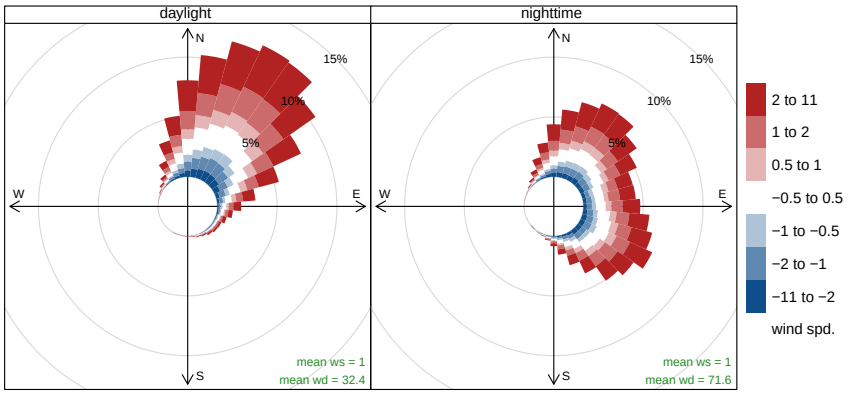


FIGURE 12.8 Pollution rose showing the difference between two meteorolgical data sets. The colours are used to show whether data tend to be postively or negatively biased with respect to the reference data set. In this case the example shows how to use user-defined breaks and split the data by day/night for a latitude assumed to be London.

13 The percentileRose function

13.1 Purpose

see also windRose, polarPlot pollutionRose polarAnnulus

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.

By default the function plots percentile concentrations in 10 degree segments. Alternatively, the levels by wind direction are calculated using a cyclic smooth cubic spline. 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 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.

13.2 Options available

The percentileRose function has the following options:

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")`

`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.

13.3 Example of use

The first example is a basic plot of percentiles of O_3 shown in Figure 13.1.

A slightly more interesting plot is shown in Figure 13.2 for SO_2 concentrations. We also take the opportunity of changing some of the default options. In this case it can be clearly seen that the highest concentrations of SO_2 are dominated by east and south-easterly winds; likely reflecting the influence of stack emissions in those directions.

A smoothed version of Figure 13.2 can be plotted by:

```
percentileRose(mydata, pollutant = "so2",
               percentile = c(25, 50, 75, 90, 95, 99, 99.9),
               col = "brewer1", key.position = "right",
               smooth = TRUEs)
```

Lots more insight can be gained by considering how percentile values vary by other factors i.e. conditioning. For example, what do O_3 concentrations look like split by season and whether it is daylight or nighttime hours? We can set the type to consider season *and* whether it is daylight or nighttime.[13] This Figure reveals some interesting

[13] In choosing `type = "daylight"` the default is to consider a latitude of central London (or close to). Users can set the latitude in the function call if working in other parts of the world.

```
percentileRose(mydata, pollutant = "o3")
```

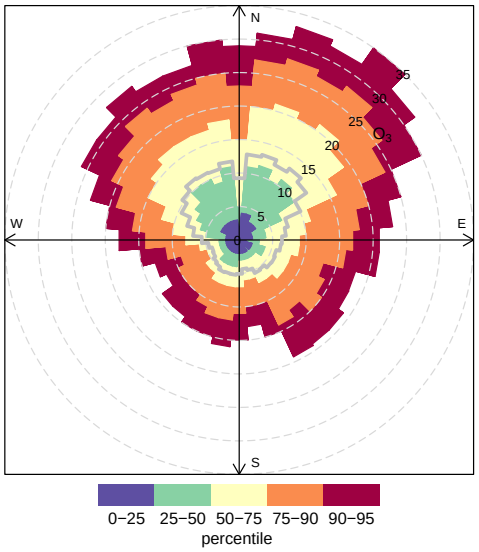


FIGURE 13.1 A `percentileRose` plot of O_3 concentrations at Marylebone Road. The percentile intervals are shaded and are shown by wind direction. It shows for example that higher concentrations occur for northerly winds, as expected at this location. However, it also shows, for example the actual value of the 95th percentile O_3 concentration.

```
percentileRose(mydata, pollutant = "so2",
               percentile = c(25, 50, 75, 90, 95, 99, 99.9),
               col = "brewer1", key.position = "right", smooth = TRUE)
```

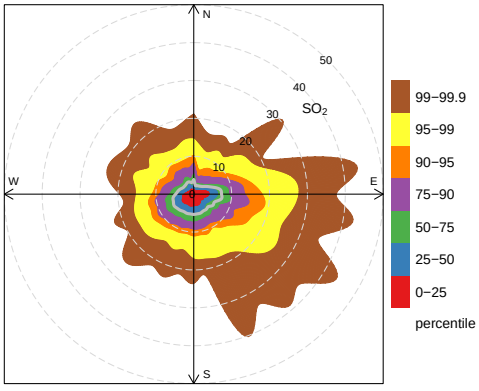


FIGURE 13.2 A `percentileRose` plot of SO_2 concentrations at Marylebone Road. The percentile intervals are shaded and are shown by wind direction. This plot sets some user-defined percentile levels to consider the higher SO_2 concentrations, moves the key to the right and uses an alternative colour scheme.

```
percentileRose(mydata, type = c("season", "daylight"), pollutant = "o3",
               col = "Set3", mean.col = "black")
```

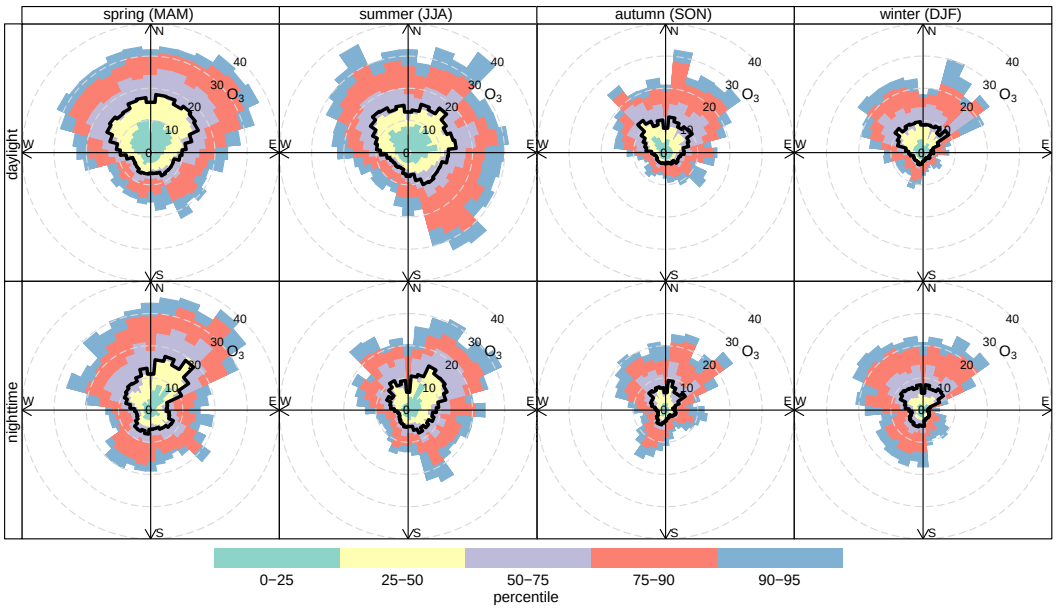


FIGURE 13.3 A `percentileRose` plot of O_3 concentrations at Marylebone Road. The percentile intervals are shaded and are shown by wind direction.The plot show the variation by season and whether it is nighttime or daylight hours.

features. First, O_3 concentrations are higher in the spring and summer and when the wind is from the north. O_3 concentrations are higher on average at this site in spring due to the peak of northern hemispheric O_3 and to some extent local production. This may also explain why O_3 concentrations are somewhat higher at nighttime in spring compared with summer. Second, peak O_3 concentrations are higher during daylight hours in summertime when the wind is from the south-east. This will be due to more local (UK/European) production that is photochemically driven — and hence more important during daylight hours.

The `percentileRose` function can also plot conditional probability functions (CPF) (Ashbaugh et al. 1985). The CPF is defined as CPF = m_θ/n_θ, where m_θ is the number of samples in the wind sector θ with mixing ratios greater than some 'high' concentration, and n_θ is the total number of samples in the same wind sector. CPF analysis is very useful for showing which wind directions are dominated by high concentrations and give the probability of doing so. In **openair**, a CPF plot can be produced as shown in Figure 13.4. Note that in these plots only one percentile is provided and the method must be supplied. In Figure 13.4 it is clear that the high concentrations (greater than the 95th percentile of *all* observations) is dominated by easterly wind directions. There are very low conditional probabilities of these concentrations being experienced for other wind directions.

It is easy to plot several species on the same plot and this works well because they all have the same probability scale (i.e. 0 to 1). In the example below (not shown) it is easy to see for each pollutant the wind directions that dominate the contributions to the highest (95th percentile) concentrations. For example, the highest CO and NO_x concentrations are totally dominated by south/south-westerly winds and the probability of their being such high concentrations from other wind directions is effectively zero.

```
percentileRose(mydata, poll="so2", percentile = 95, method = "cpf",
               col = "darkorange", smooth = TRUE)
```

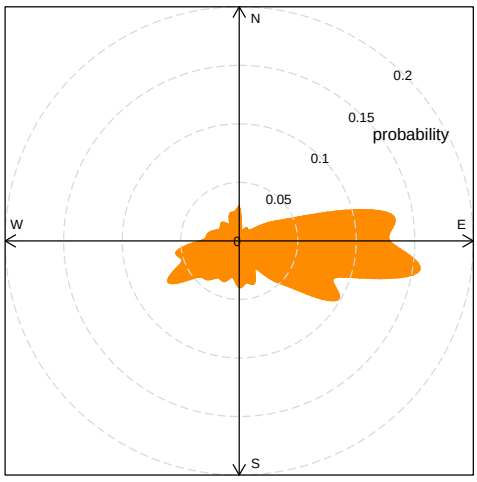


FIGURE 13.4 A CPF plot of SO_2 concentrations at Marylebone Road.

```
percentileRose(mydata, poll=c("nox", "so2", "o3", "co", "pm10", "pm25"),
               percentile = 95, method = "cpf", col = "darkorange",
               layout = c(2, 3))
```

14 The polarFreq function

14.1 Purpose

see also windRose percentileRose polarPlot

This is a custom-made plot to compactly show the distribution of wind speeds and directions from meteorological measurements. It is similar to the traditional wind rose, but includes a number of enhancements to also show how concentrations of pollutants and other variables vary. It can summarise all available data, or show it by different time periods e.g. by year, month, day of the week. It can also consider a wide range of statistics.

14.2 Options available

The polarFreq function has the following options:

`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.

`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.

For `type = "site"`, it is necessary to format the input data into columns `date`, `ws`, `wd`, `site` (and maybe `pollutant`). This means that `date`, for example is repeated a number of times equal to the number of sites.

14.3 Example of use

This section shows an example output and use, using our data frame `mydata`. The function is very simply run as shown in Figure 14.1. This produces the plot shown in Figure 14.1.

By setting `type = "year"`, the frequencies are shown separately by year as shown in Figure 14.2, which shows that most of the time the wind is from a south-westerly direction with wind speeds most commonly between 2–6 m s^{-1}. In 2000 there seemed to be a lot of conditions where the wind was from the south-west (leading to high pollutant concentrations at this location). The data for 2003 also stand out due to the relatively large number of occasions where the wind was from the east. Note the default colour scale, which has had a square-root transform applied, is used to provide a better visual distribution of the data.

The `polarFreq` function can also usefully consider pollutant concentrations. Figure 14.3 shows the mean concentration of SO_2 by wind speed and wind direction and clearly highlights that SO_2 concentrations tend to be highest for easterly winds and for 1998 in particular.

By weighting the concentrations by the frequency of occasions the wind is from a certain direction and has a certain speed, gives a better indication of the conditions

```
polarFreq(mydata)
```

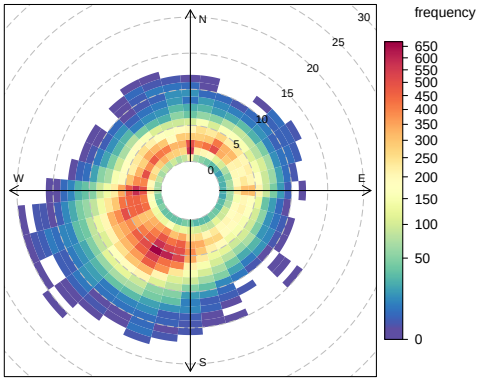


FIGURE 14.1 Use of `polarFreq` function to plot wind speed/directions. Each cell gives the total number of hours the wind was from that wind speed/direction in a particular year. The number of hours is coded as a colour scale shown to the right. The scale itself is non-linear to help show the overall distribution. The dashed circular grey lines show the wind speed scale. The date range covered by the data is shown in the strip.

that dominate the overall mean concentrations. Figure 14.4 shows the weighted mean concentration of SO_2 and highlights that annual mean concentrations are dominated by south-westerly winds i.e. contributions from the road itself — and not by the fewer higher hours of concentrations when the wind is easterly. However, 2003 looks interesting because for that year, significant contributions to the overall mean were due to easterly wind conditions.

These plots when applied to other locations can reveal some useful features about different sources. For example, it may be that the highest concentrations measured only occur infrequently, and the weighted mean plot can help show this.

The code required to make Figure 14.3 and 14.4 is shown below.

Users are encouraged to try out other plot options. However, one potentially useful plot is to select a few specific years of user interest. For example, what if you just wanted to compare two years e.g. 2000 and 2003? This is easy to do by sending a subset of data to the function. Use here can be made of the **openair** `selectByDate` function.

```
# wind rose for just 2000 and 2003
polarFreq(selectByDate(mydata, year = c(2000, 2003)), cols = "jet",
          type = "year")
```

The `polarFreq` function can also be used to gain an idea about the wind directions that contribute most to the overall mean concentrations. As already shown, use of the option `statistic = "weighted.mean"` will show the percentage contribution by wind direction and wind speed bin. However, often it is unnecessary to consider different wind speed intervals. To make the plot more effective, a few options are set as shown in Figure 14.5. First, the `statistic = "weighted.mean"` is chosen to ensure that the plot shows concentrations weighted by their frequency of occurrence of wind direction. For this plot, we are mostly interested in just the contribution by wind direction and not wind speed. Setting the `ws.int` to be above the maximum wind speed in the data set ensures that all data are shown in one interval. Rather than having a square-root transform applied to the colour scale, we choose to have a linear scale by setting `trans = FALSE`. Finally, to show a 'disk', the `offset` is set at

```
polarFreq(mydata, type = "year")
```

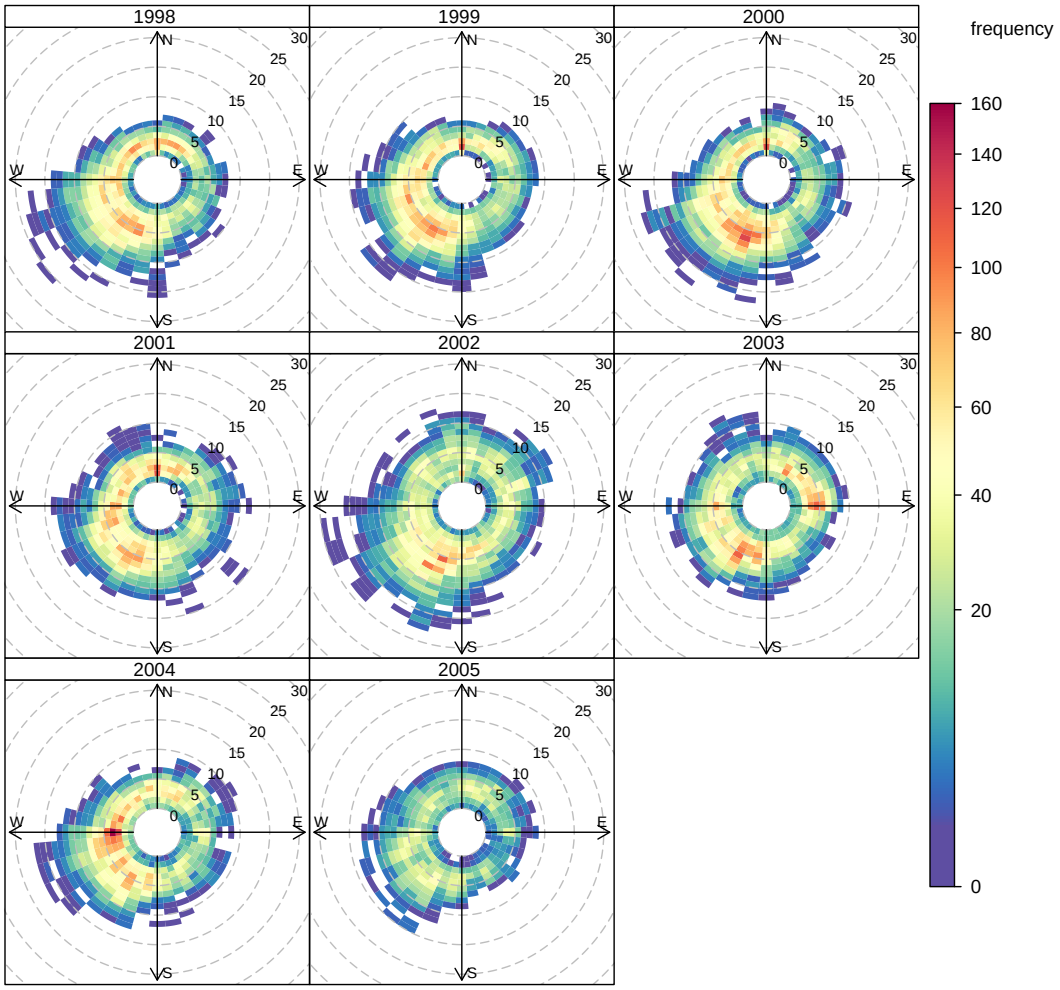


FIGURE 14.2 Use of `polarFreq` function to plot wind speed/directions by year. Each cell gives the total number of hours the wind was from that wind speed/direction in a particular year. The number of hours is coded as a colour scale shown to the right. The scale itself is non-linear to help show the overall distribution. The dashed circular grey lines show the wind speed scale.

80. Increasing the value of the offset will narrow the disk.

While Figure 14.5 is useful — e.g. it clearly shows that concentrations of NO_x at this site are totally dominated by south-westerly winds, the use of `pollutionRose` for this type of plot is more effective, as shown in Section 12.

```
polarFreq(mydata, pollutant = "so2", type = "year",
          statistic = "mean", min.bin = 2)
```

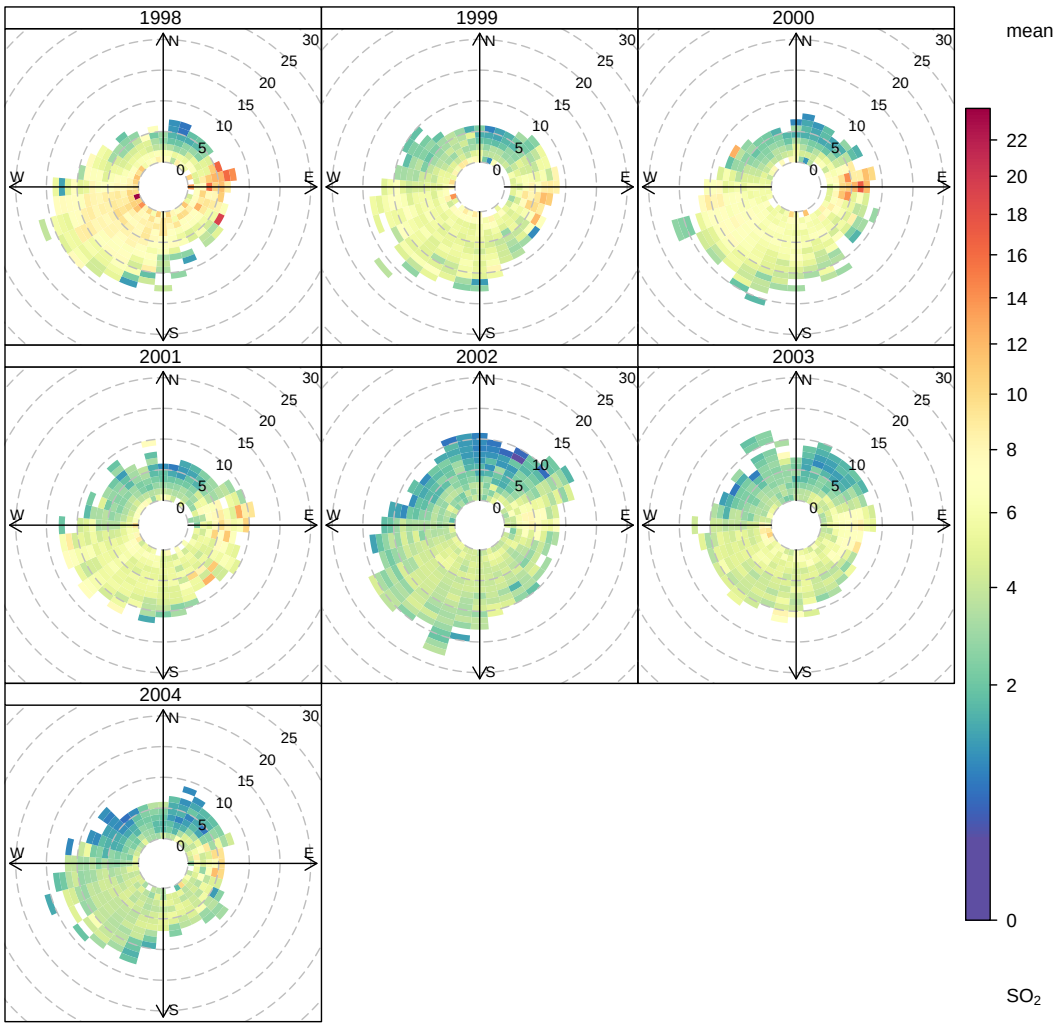


FIGURE 14.3 Use of polarFreq function to plot mean SO_2 concentrations (ppb) by wind speed/directions and year.

```
# weighted mean SO2 concentrations
polarFreq(mydata, pollutant = "so2", type = "year",
          statistic = "weighted.mean", min.bin = 2)
```

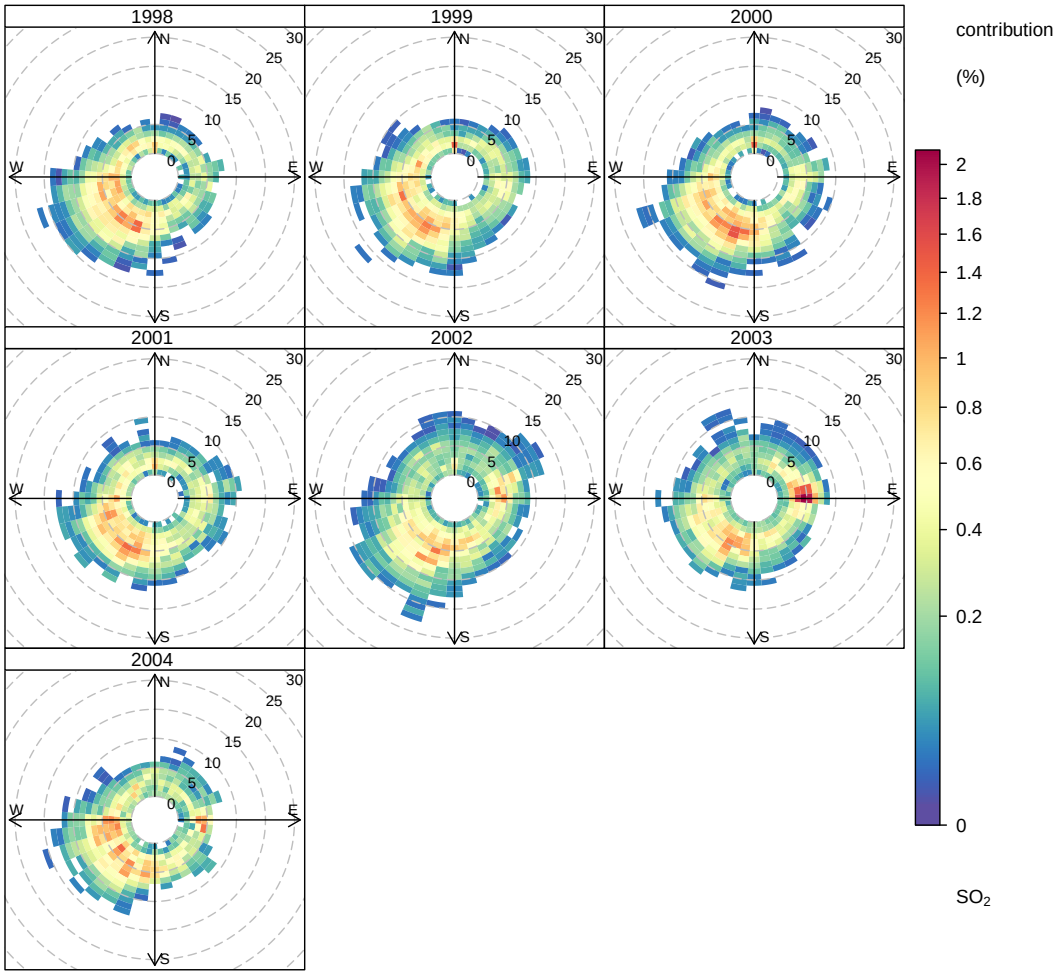


FIGURE 14.4 Use of polarFreq function to plot weighted mean SO_2 concentrations (ppb) by wind speed/directions and year.

```
polarFreq(mydata, pollutant = "nox", ws.int = 30, statistic = "weighted.mean",
          offset = 80, trans = FALSE, col = "heat")
```

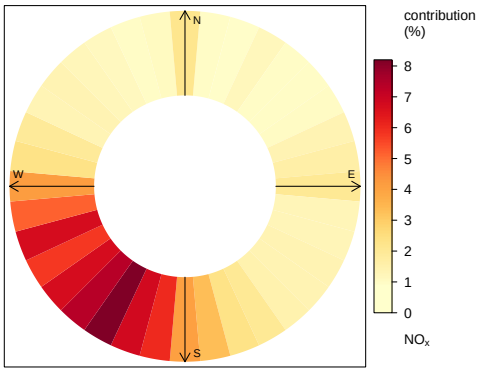


FIGURE 14.5 The percentage contribution to overall mean concentrations of NO_x at Marylebone Road.

15 The polarPlot and polarCluster functions

15.1 Purpose

see also
polarFreq
polarAnnulus
percentileRose
pollutionRose

The polarPlot function plots a *bivariate polar plot* of concentrations. Concentrations are shown to vary by wind speed and wind direction. In many respects they are similar to the plots shown in (§14) but include some additional enhancements. These enhancements include: plots are shown as a continuous surface and surfaces are calculated through modelling using smoothing techniques. These plots are not entirely new as others have considered the joint wind speed-direction dependence of concentrations (see for example Yu et al. (2004)). However, plotting the data in polar coordinates and for the purposes of source identification is new. Furthermore, the basic polar plot is since been enhanced in many ways as described below. Recent publications that describe or use the technique are Carslaw et al. (2006) and Westmoreland et al. (2007). These plots have proved to be useful for quickly gaining a graphical impression of potential sources influences at a location.

The polarPlot function is described in more detail in Carslaw et al. (2006) where it is used to triangulate sources in an airport setting, Carslaw and Beevers (2013) where it is used with a clustering technique to identify features in a polar plot with similar characteristics and Uria-Tellaetxe and Carslaw (2014) where it is extended to include a conditional probability function to extract more information from the plots. The latter paper is Open Access and can be downloaded from the **openair** website.

For many, maybe most situations, increasing wind speed generally results in lower concentrations due to increased dilution through advection and increased mechanical turbulence. There are, however, many processes that can lead to interesting concentration-wind speed dependencies and we will provide a more theoretical treatment of this in due course. However, below are a few reasons why concentrations can change with increasing wind speeds.

- Buoyant plumes from tall stacks can be brought down to ground-level resulting in high concentrations under high wind speed conditions.
- Particle suspension increases with increasing wind speeds e.g. PM_{10} from spoil heaps and the like.
- 'Particle' suspension can be important close to coastal areas where higher wind speeds generate more sea spray.
- The wind speed dependence of concentrations in a street canyon can be very complex: higher wind speeds do not always results in lower concentrations due to re-circulation. Bivariate polar plots are very good at revealing these complexities.
- As Carslaw et al. (2006) showed, aircraft emissions have an unusual wind speed dependence and this can help distinguish them from other sources. If several measurement sites are available, polar plots can be used to triangulate different sources.
- Concentrations of NO_2 can increase with increasing wind speed — or at least not decline steeply due to increased mixing. This mixing can result in O_3-rich air converting NO to NO_2.

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. For research applications where many more variables could be available, discriminating sources by these other variables could be very insightful.

Bivariate polar plots are constructed in the following way. First, wind speed, wind direction and concentration data are partitioned into wind speed-direction bins and the mean concentration calculated for each bin. Testing on a wide range of data suggests that wind direction intervals at 10 degrees and 30 wind speed intervals capture sufficient detail of the concentration distribution. The wind direction data typically available are generally rounded to 10 degrees and for typical surface measurements of wind speed in the range 0 to 20 to 30 m s^{-1}, intervals greater than 30 would be difficult to justify based on a consideration of the accuracy of the instruments. Binning the data in this way is not strictly necessary but acts as an effective data reduction technique without affecting the fidelity of the plot itself. Furthermore, because of the inherent wind direction variability in the atmosphere, data from several weeks, months or years typically used to construct a bivariate polar plot tends to be diffuse and does not vary abruptly with either wind direction or speed and more finely resolved bin sizes or working with the raw data directly does not yield more information.

The wind components, u and v are calculated i.e.

$$u = \overline{u}.sin\left(\frac{2\pi}{\theta}\right), v = \overline{u}.cos\left(\frac{2\pi}{\theta}\right) \quad (1)$$

with $\overline{u}$ is the mean hourly wind speed and θ is the mean wind direction in degrees with 90 degrees as being from the east.

The calculations above provides a u, v, concentration (C) surface. While it would be possible to work with this surface data directly a better approach is to apply a model to the surface to describe the concentration as a function of the wind components u and v to extract real source features rather than noise. A flexible framework for fitting a surface is to use a Generalized Additive Model (GAM) e.g. (Hastie and Tibshirani 1990; Wood 2006). GAMs are a useful modelling framework with respect to air pollution prediction because typically the relationships between variables are non-linear and variable interactions are important, both of which issues can be addressed in a GAM framework. GAMs can be expressed as shown in Equation 2:

$$\sqrt{C_i} = \beta_0 + \sum_{j=1}^{n} s_j(x_{ij}) + e_i \quad (2)$$

where C_i is the ith pollutant concentration, β_0 is the overall mean of the response, $s_j(x_{ij})$ is the smooth function of ith value of covariate j, n is the total number of covariates, and e_i is the ith residual. Note that C_i is square-root transformed as the transformation generally produces better model diagnostics e.g. normally distributed residuals.

The model chosen for the estimate of the concentration surface is given by Equation 3. In this model the square root-transformed concentration is a smooth function of the bivariate wind components u and v. Note that the smooth function used is *isotropic* because u and v are on the same scales. The isotropic smooth avoids the potential difficulty of smoothing two variables on different scales e.g. wind speed and direction, which introduces further complexities.

$$\sqrt{C_i} = s(u, v) + e_i \quad (3)$$

15.2 Options available

The polarPlot function has the following options:

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).

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. 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.

When 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.

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

`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 interval. Calculating the uncertainties is useful to understand whether features are real or not. For example, at high wind speeds where there are few data there is greater uncertainty over the predicted values. The uncertainties are calculated using the GAM and weighting is done by the frequency of measurements in each wind speed-direction bin. Note that if uncertainties are calculated then the type is set to "default".

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

`percentile` is also used for the Conditional Probability Function (CPF) plots. `percentile` can be of length two, in which case the percentile *interval* is considered for use with CPF. For example, `percentile = c(90, 100)` will plot the CPF for concentrations between the 90 and 100th percentiles. Percentile intervals can be useful for identifying specific sources. In addition, `percentile` can also be of length 3. The third value is the 'trim' value to be applied. When calculating percentile intervals many can cover very low values where there is no useful information. The trim value ensures that values greater than or equal to the trim * mean value are considered *before* the percentile intervals are calculated. The effect is to extract more detail from many source signatures. See the manual for examples. Finally, if the trim value is less than zero the

percentile range is interpreted as absolute concentration values and subsetting is carried out directly.

`cols` Colours to 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")`

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

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

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

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

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

`angle.scale` 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.

`...` 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.

15.3 Example of use

We first use the function in its simplest form to make a polar plot of NO_x. The code is very simple as shown in Figure 15.1.

This produces Figure 15.1. The scale is automatically set using whatever units the original data are in. This plot clearly shows highest NO_x concentrations when the wind is from the south-west. Given that the monitor is on the *south* side of the street and the highest concentrations are recorded when the wind is blowing *away* from the monitor is strong evidence of street canyon recirculation.

Figure 15.2 and Figure 15.3 shows polar plots using different defaults and for other pollutants. In the first (Figure 15.2, a different colour scheme is used and some adjustments are made to the key. In Figure 15.3, SO_2 concentrations are shown. What is interesting about this plot compared with either Figure 15.2 or Figure 15.1 is that the

```
polarPlot(mydata, pollutant = "nox")
```

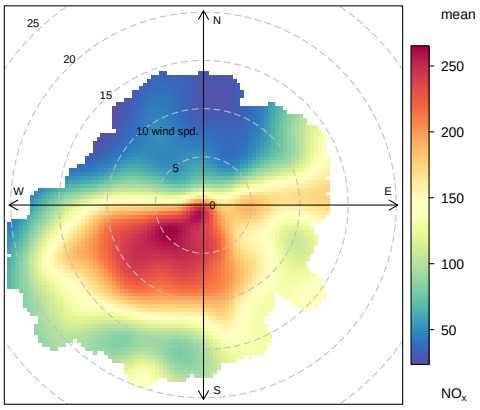


FIGURE 15.1 Default use of the polarPlot function applied to Marylebone Road NO_x concentrations.

concentration pattern is very different i.e. high concentrations with high wind speeds from the east. The most likely source of this SO_2 are industrial sources to the east of London. The plot does still however show evidence of a source to the south-west, similar to the plot for NO_x, which implies that road traffic sources of SO_2 can also be detected.

These plots often show interesting features at higher wind speeds. 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`. The problem with this approach is that it is difficult to know how best to bin wind speed and direction: the choice of interval tends to be arbitrary. 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. This is a useful strategy for testing how sensitive the plotted surface is to the number of measurements available. While this is a useful strategy to get a feel for how the surface changes with different `min.bin` settings, it is still difficult to know how many points should be used as a minimum. Third, consider setting `uncertainty = TRUE`. This option will show the predicted surface together with upper and lower 95% confidence intervals, which take account of the frequency of measurements. The uncertainty approach ought to be the most robust and removes any arbitrary setting of other options. There is a close relationship between the amount of smoothing an the uncertainty: more smoothing will tend to reveal less detail and lower uncertainties in the fitted surface and vice-versa.

The default however is to down-weight the bins with few data points when fitting a surface. Weights of 0.25, 0.5 and 0.75 are used for bins containing 1, 2 and 3 data points respectively. The advantage of this approach is that no data are actually removed (which is what happens when using `min.bin`). This approach should be robust in a very wide range of situations and is also similar to the approaches used when trying to locate sources when using back trajectories as described in Section 26. Users can ignore the automatic weighting by supplying the option `weights = c(1, 1, 1)`.

polarFreq provides an un-smoothed surface

```
## NOx plot
polarPlot(mydata, pollutant = "nox", col = "jet", key.position = "bottom",
          key.header = "mean nox (ug/m3)", key.footer = NULL)
```

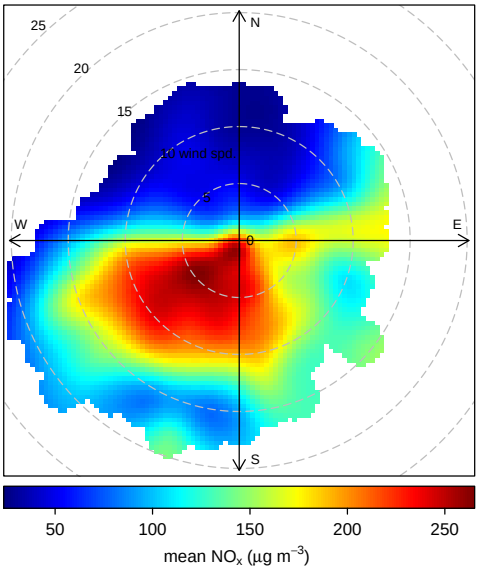


FIGURE 15.2 Example plots using the polarPlot function with different options for the mean concentration of NO_x.

```
polarPlot(mydata, pollutant = "so2")
```

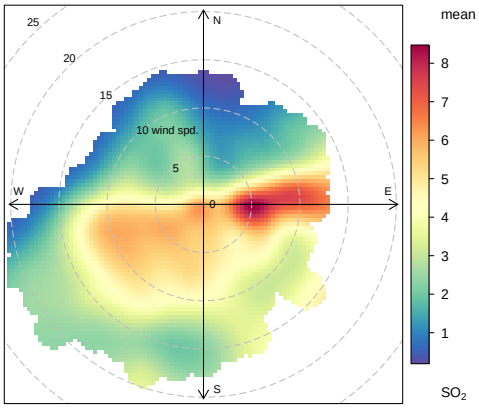


FIGURE 15.3 Example plots using the polarPlot function for the mean concentration of SO_2.

```
polarPlot(mydata, pollutant = "ratio", main = "so2/nox ratio")
```

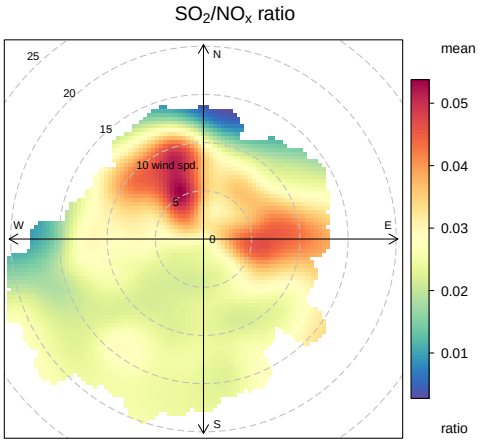


FIGURE 15.4 Bivariate polar plot of the ratio of SO_2/NO_x.

A very useful approach for understanding air pollution is to consider ratios of pollutants. One reason is that pollutant ratios can be largely independent of meteorological variation. In many circumstances it is possible to gain a lot of insight into sources if pollutant ratios are considered. First, it is necessary to calculate a ratio, which is easy in R. In this example we consider the ratio of SO_2/NO_x:

```
mydata <- transform(mydata, ratio = so2 / nox)
```

Working with ratios of pollutants

This makes a new variable called `ratio`. Sometimes it can be problematic if there are values equal to zero on the denominator, as is the case here. The mean and maximum value of the ratio is infinite, as shown by the `Inf` in the statistics below. Luckily, R can deal with infinity and the **openair** functions will remove these values before performing calculations. It is very simple therefore to calculate ratios.

```
summary(mydata[, "ratio"])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       0       0       0     Inf       0     Inf   11782
```

A polar plot of the SO_2/NO_x ratio is shown in Figure 15.4. The plot highlights some new features not seen before. First, to the north there seems to be evidence that the air tends to have a higher SO_2/NO_x ratio. Also, the source to the east has a higher SO_2/NO_x ratio compared with that when the wind is from the south-west i.e. dominated by road sources. It seems therefore that the easterly source(s), which are believed to be industrial sources have a different SO_2/NO_x ratio compared with road sources. This is a very simple analysis, but ratios can be used effectively in many functions and are particularly useful in the presence of high source complexity.

Sometimes when considering ratios it might be necessary to limit the values in some way; perhaps due to some unusually low value denominator data resulting in a few very high values for the ratio. This is easy to do with the `subset` command. The code below selects ratios less than 0.1.

```
polarPlot(subset(mydata, ratio < 0.1), pollutant = "ratio")
```

The uncertainties in the surface can be calculated by setting the option `uncertainty`

```
polarPlot(mydata, pollutant = "so2", uncertainty = TRUE)
```

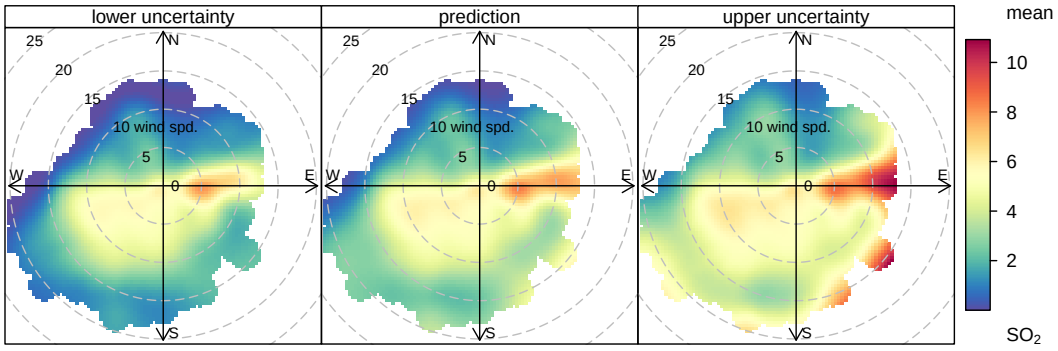


FIGURE 15.5 Bivariate polar plot of SO_2 concentrations at Marylebone Road. Three surfaces are shown: the central prediction (middle) and the lower and upper 95% estimated uncertainties. These plots help to show that in this particular case, some of the concentrations for strong easterly and south-easterly winds are rather uncertain. However, the central feature to the east remains, suggesting this feature is 'real' and not an artifact of there being too few data.

= TRUE. The details are described above and here we show the example of SO_2 concentrations (Figure 15.5). In general the uncertainties are higher at high wind speeds i.e. at the 'fringes' of a plot where there are fewer data. However, the magnitude depends on both the frequency and magnitude of the concentration close to the points of interest. The pattern of uncertainty is not always obvious and it can differ markedly for different pollutants.

The `polarPlot` function can also produce plots dependent on another variable (see the `type` option). For example, the variation of SO_2 concentrations at Marylebone Road by hour of the day in the code below. The function was called as shown in in this case the minimum number of points in each wind speed/direction was set to 2.

```
polarPlot(mydata, pollutant = "so2", type = "hour", min.bin = 2)
```

This plot shows that concentrations of SO_2 tend to be highest from the east (as also shown in Figure 15.3) and for hours in the morning. Together these plots can help better understand different source types. For example, does a source only seem to be present during weekdays, or winter months etc. In the case of `type = "hour"`, the more obvious presence during the morning hours could be due to meteorological factors and this possibility should be investigated also. In other settings where there are many sources that vary in their source emission and temporal characteristics, the `polarPlot` function should prove to be very useful.

One issue to be aware of is the amount of data required to generate some of these plots; particularly the hourly plots. If only a relatively short time series is available there may not be sufficient information to produce useful plots. Whether this is important or not will depend on the specific circumstances e.g. the prevalence of wind speeds and directions from the direction of interest. When used to produce many plots (e.g. type = "hour"), the run time can be quite long.

15.3.1 Conditional Probability Function (CPF) plot

The conditional probability functions (CPF) was described on page 116 in the context of the `percentileRose` function. The CPF was originally used to show the wind directions that dominate a (specified) high concentration of a pollutant; showing the probability

of such concentrations occurring by wind direction (Ashbaugh et al. 1985). However, these ideas can very usefully be applied to bivariate polar plots. In this case the CPF is defined as CPF = $m_{\theta_j}/n_{\theta_j}$, where m_{θ_j} is the number of samples in the wind sector θ and wind speed interval j with mixing ratios greater than some 'high' concentration, and n_{θ_j} is the total number of samples in the same wind direction-speed interval. Note that j does not have to be wind speed but could be any numeric variable e.g. ambient temperature. CPF analysis is very useful for showing which wind direction, wind speed intervals are dominated by high concentrations and give the probability of doing so. A full explanation of the development and use of the bivariate case of the CPF is described in Uria-Tellaetxe and Carslaw (2014) where it is applied to monitoring data close to a steelworks.

```
polarPlot(mydata, pollutant = "so2", statistic = "cpf", percentile = 90)
```

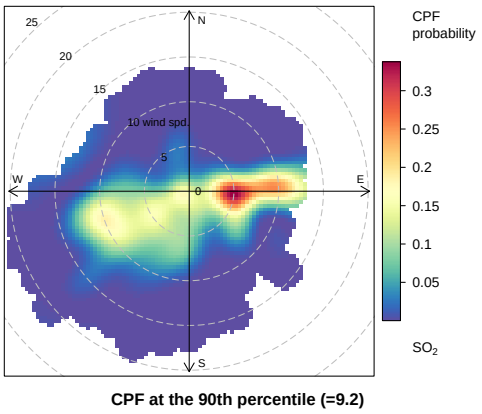


FIGURE 15.6 polarPlot of SO_2 concentrations at Marylebone Road based on the CPF function.

An example of a CPF polar plot is shown in Figure 15.6 for the 90th percentile concentration of SO_2. This plot shows that for most wind speed-directions the probability of SO_2 concentrations being greater than the 90th percentile is zero. The clearest areas where the probability is higher is to the east. Indeed, the plot now clearly reveals two potential sources of SO_2, which are not as apparent in the 'standard' plot shown in Figure 15.3. Note that Figure 15.6 also gives the calculated percentile at the bottom of the plot (9.2 ppb in this case). Figure 15.6 can also be compared with the CPF plot based only on wind direction shown in Figure 13.4. While Figure 13.4 very clearly shows that easterly wind dominate high concentrations of SO_2, Figure 15.6 provides additional valuable information by also considering wind speed, which in this case is able to discriminate between two sources (or groups of sources) to the east.

The polar CPF plot is therefore potentially very useful for source identification and characterisation. It is, for example, worth also considering other percentile levels and other pollutants. For example, considering the 95th percentile for SO_2 'removes' one of the sources (the one at highest wind speed). This helps to show some maybe important differences between the sources that could easily have been missed. Similarly, considering other pollutants can help build up a good understanding of these sources. A CPF plot for NO_2 at the 90th percentile shows the single dominance of the road source. However, a CPF plot at the 75th percentile level indicates source contributions from the east (likely tall stacks), which again are not as clear in the standard bivariate polar plot. Considering a range of percentile values can therefore help to build up a more complete understanding of source contributions.

However, even more useful information can be gained by considering *intervals* of

```
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(0, 10))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(10, 20))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(20, 30))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(30, 40))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(40, 50))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(50, 60))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(60, 70))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(70, 80))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(80, 90))
polarPlot(mydata, poll= "so2", stati="cpf", percentile = c(90, 100))
```

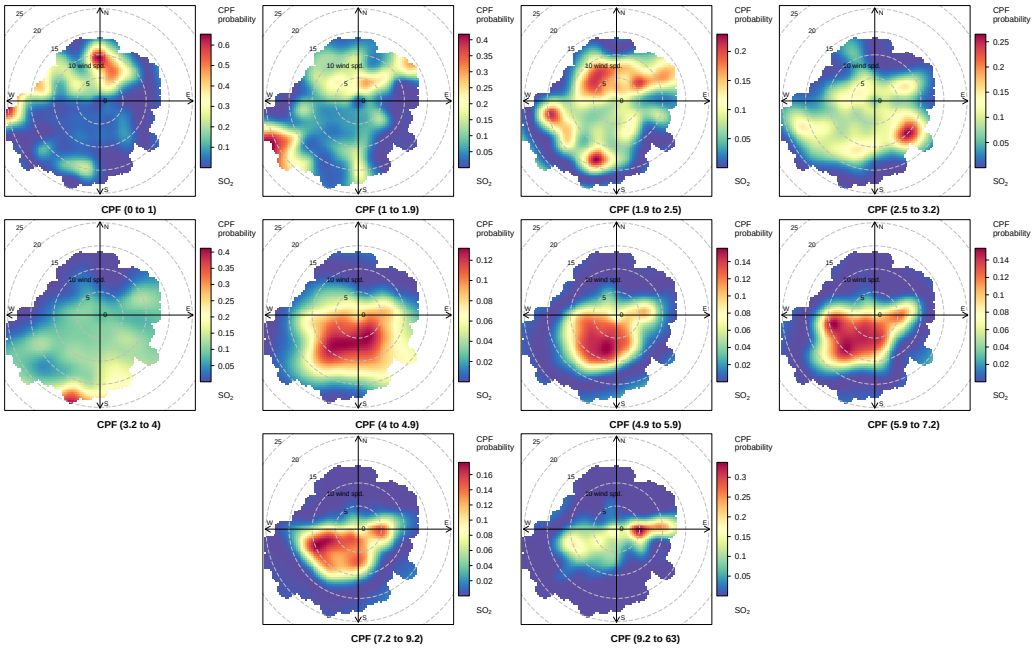


FIGURE 15.7 `polarPlot` of SO_2 concentrations at Marylebone Road based on the CPF function for a range of percentile intervals from 0–10, 10–20, ..., 90-100.

percentiles e.g. 50–60, 60–70 etc. By considering intervals of percentiles it becomes clear that some sources only affect a limited percentile range. `polarPlot` can accept a `percentile` argument of length two e.g. `percentile = c(80, 90)`. In this case concentrations in the range from the lower to upper percentiles will be considered. In Figure 15.7 for example, it is apparent that the road source to the south west is only important between the 60 to 90th percentiles. As mentioned previously, the chimney stacks to the east are important for the higher percentiles (90 to 100). What is interesting though is the emergence of what appears to be other sources at the lower percentile intervals. These potential sources are not apparent in Figure 15.3. The other interesting aspect is that it does seem that specific sources tend to be prominent for specific percentile ranges. If this characteristic is shown to be the case more generally, then CPF intervals could be a powerful way in which to identify many sources. Whether these particular sources are important or not is questionable and depends on the aims of the analysis. However, there is no reason to believe that the potential sources shown in the percentile ranges 0 to 50 are artifacts. They could for example be signals from more distant point sources whose plumes have diluted more over longer distances. Such sources would be 'washed out' in an ordinary polar plot. For a fuller example of this approach see Uria-Tellaetxe and Carslaw (2014).

Note that it is easy to work out what the concentration intervals are for the percentiles

shown in Figure 15.7:

```
quantile(mydata$so2, probs = seq(0, 1, by = 0.1), na.rm = TRUE)
```

```
##      0%     10%     20%     30%     40%     50%     60%     70%     80%     90%
##  0.0000  1.0125  1.8825  2.5000  3.2500  4.0000  4.9375  5.9100  7.2375  9.2500
##    100%
## 63.2050
```

To plot the Figures on one page it is necessary to make the plot objects first and then decide how to plot them. To plot the Figures in a particular layout see page 74.

15.3.2 The polarCluster function for feature identification and extraction

The polarPlot function will often identify interesting features that would be useful to analyse further. It is possible to select areas of interest based only on a consideration of a plot. Such a selection could be based on wind direction and wind speed intervals for example e.g.

```
subdata <- subset(mydata, ws >3 & wd >= 180 & wd <=270)
```

which would select wind speeds >3 m s^{-1} and wind directions from 180 to 270 degrees from mydata. That subset of data, subdata, could then be analysed using other functions. While this approach may be useful in many circumstances it is rather arbitrary. In fact, the choice of 'interesting feature' in the first place can even depend on the colour scale used, which is not very robust. Furthermore, many interesting patterns can be difficult to select and won't always fall into convenient intervals of other variables such as wind speed and direction.

A better approach is to use a method that can select group similar features together. One such approach is to use *cluster analysis*. **openair** uses k-means clustering as a way in which bivariate polar plot features can be identified and grouped. The main purpose of grouping data in this way is to identify records in the original time series data by cluster to enable post-processing to better understand potential source characteristics. The process of grouping data in k-means clustering proceeds as follows. First, k points are randomly chosen form the space represented by the objects that are being clustered into k groups. These points represent initial group centroids. Each object is assigned to the group that has the closest centroid. When all objects have been assigned, the positions of the k centroids is re-calculated. The previous two steps are repeated until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimised can be calculated.

Central to the idea of clustering data is the concept of distance i.e. some measure of similarity or dissimilarity between points. Clusters should be comprised of points separated by small distances relative to the distance between the clusters. Careful consideration is required to define the distance measure used because the effectiveness of clustering itself fundamentally depends on its choice. The similarity of concentrations shown in Figure 15.1 for example is determined by three variables: the u and v wind components and the concentration. All three variables are equally important in characterising the concentration-location information, but they exist on different scales i.e. a wind speed-direction measure and a concentration. Let $X = \{x_i\}, i = 1, \ldots, n$ be a set of n points to be clustered into K clusters, $C = \{c_k, k = 1, \ldots, K\}$. The basic k-means algorithm for K clusters is obtained by minimising:

$$\sum_{k=1}^{K}\sum_{x_i \in c_k} ||x_i - \mu_k||^2 \quad (4)$$

where $||x_i - \mu_k||^2$ is a chosen distance measure, μ_k is the mean of cluster c_k. The distance measure is defined as the Euclidean distance:

$$d_{x,y} = \left(\sum_{j=1}^{J}(x_j - y_j)^2\right)^{1/2} \quad (5)$$

Where **x** and **y** are two J-dimensional vectors, which have been standardized by subtracting the mean and dividing by the standard deviation. In the current case J is of length three i.e. the wind components u and v and the concentration C, each of which is standardized e.g.:

$$x_j = \left(\frac{x_j - \overline{x}}{\sigma_x}\right) \quad (6)$$

Standardization is necessary because the wind components u and v are on different scales to the concentration. In principle, more weight could be given to the concentration rather than the u and v components, although this would tend to identify clusters with similar concentrations but different source origins.

`polarCluster` can be thought of as the 'local' version of clustering of back trajectories. Rather than using air mass origins, wind speed, wind direction and concentration are used to group similar conditions together. Section 26.3 provides the details of clustering back trajectories in **openair**. A fuller description of the clustering approach is described in Carslaw and Beevers (2013).

The `polarCluster` function has the following options.

`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 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")`.

`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.

The use of the `polarCluster` is very similar to the use of all **openair** functions. While there are many techniques available to try and find the optimum number of clusters, it is difficult for these approaches to work in a consistent way for identifying features in bivariate polar plots. For this reason it is best to consider a range of solutions that covers a number of clusters.

Cluster analysis is computationally intensive and the `polarCluster` function can take a comparatively long time to run. The basic idea is to calculate the solution to several cluster levels and then choose one that offers the most appropriate solution for post-processing.

The example given below is for concentrations of SO_2, shown in Figure 15.3 and the aim is to identify features in that plot. A range of numbers of clusters will be calculated — in this case from two to ten.

The real benefit of `polarCluster` is being able to identify clusters in the original data frame. To do this, the results from the analysis must be read into a new variable, as in Figure 15.9, where the results are read into a data frame `results`. Now it is possible to use this new information. In the 8-cluster solution to Figure 15.9, cluster 6 seems to capture the elevated SO_2 concentrations to the east well (see Figure 15.3 for comparison), while cluster 5 will strongly represent the road contribution.

The results are here:

```
head(results[["data"]])
```

```
##                  date   ws  wd nox no2 o3 pm10    so2      co pm25      ws2
## 1 1998-01-01 00:00:00 0.60 280 285  39  1   29 4.7225  3.3725   NA 2.787295
## 2 1998-01-01 02:00:00 2.76 190  NA  NA  3   34 6.8300  9.6025   NA 5.788196
## 3 1998-01-01 03:00:00 2.16 170 493  52  3   35 7.6625 10.2175   NA 2.712619
## 4 1998-01-01 04:00:00 2.40 180 468  78  2   34 8.0700  8.9125   NA 2.197152
## 5 1998-01-01 05:00:00 3.00 190 264  42  0   16 5.5050  3.0525   NA 3.705192
## 6 1998-01-01 06:00:00 3.00 140 171  38  0   11 4.2300  2.2650   NA 6.440763
##        wd2      ratio cluster
## 1 304.5741 0.01657018       4
## 2 303.9971         NA       4
## 3 348.8439 0.01554260       5
## 4 295.2694 0.01724359       5
## 5 286.5633 0.02085227       4
## 6 151.9821 0.02473684       5
```

Note that there is an additional column `cluster` that gives the cluster a particular row belongs to and that this is a *character* variable. It might be easier to read these results into a new data frame:

```
polarCluster(mydata, pollutant="so2", n.clusters=2:10, cols= "Set2")
```

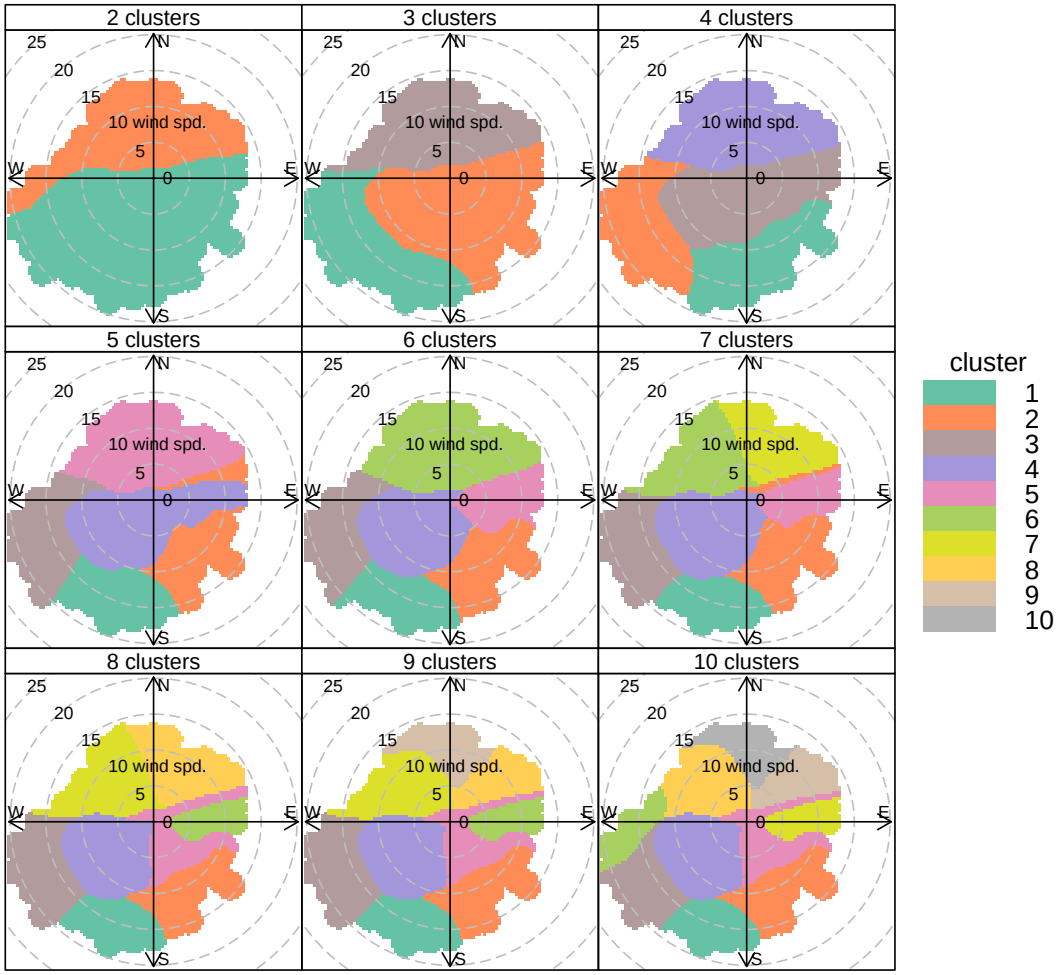


FIGURE 15.8 Use of the polarCluster function applied to SO_2 concentrations at Marylebone Road. In this case 2 to 10 clusters have been chosen.

```
results <- polarCluster(mydata, pollutant="so2", n.clusters=8, cols= "Set2")
```

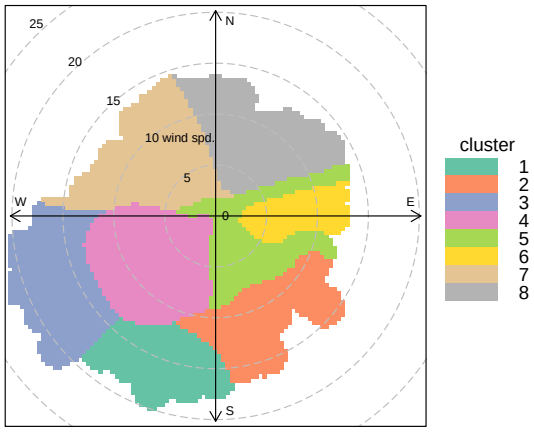


FIGURE 15.9 Use of the polarCluster function applied to SO_2 concentrations at Marylebone Road. In this case 8 clusters have been chosen.

```
timeProp(selectByDate(results, year = 2003), pollutant = "so2", avg.time = "day",
         proportion= "cluster", col = "Set3", key.position = "top",
         key.columns = 8, date.breaks = 10, ylab = "so2 (ug/m3)")
```

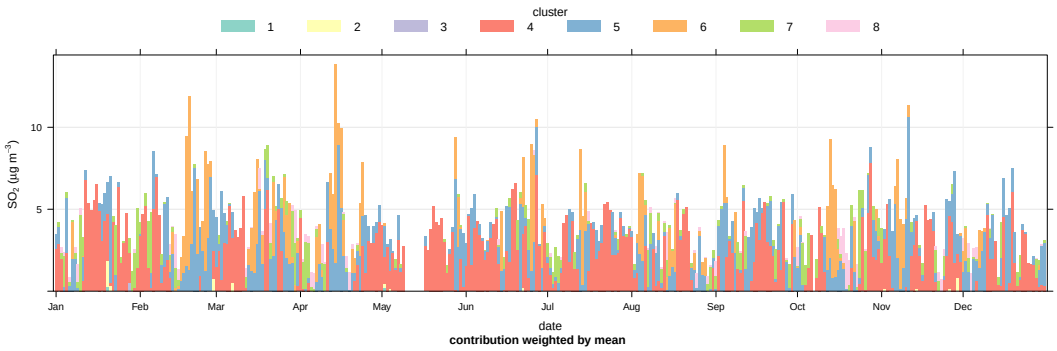


FIGURE 15.10 Temporal variation in daily SO_2 concentration at the Marylebone Road site show by contribution of each cluster for 2003.

```
results <- results[["data"]]
```

It is easy to find out how many points are in each cluster:

```
table(results[, "cluster"])
```

```
##
##     1     2     3     4     5     6     7     8
##   206   412   160 24133 16049  2590  7839  2918
```

Now other **openair** analysis functions can be used to analyse the results. For example, to consider the temporal variations by cluster:

```
timeVariation(results, pollutant="so2", group = "cluster",
              col = "Set2", ci = FALSE, lwd = 3)
```

Or if we just want to plot a couple of clusters (5 and 6) *using the same colours* as in Figure 15.9:

```
timeVariation(subset(results, cluster %in% c("C5", "C6")), pollutant="so2",
              group = "cluster",  col = openColours("Set2", 8)[5:6], lwd = 3)
```

`polarCluster` will work on any surface that can be plotted by `polarPlot` e.g. the radial variable does not have to be wind speed but could be another variable such as temperature. While it is not always possible for `polarCluster` to identify all features in a surface it certainly makes it easier to post-process `polarPlots` using other **openair** functions or indeed other analyses altogether.

Another useful way of understanding the clusters is to use the `timeProp` function, which can display a time series as a bar chart split by a categorical variable (in this case the cluster). In this case it is useful to plot the time series of SO_2 *and* show how much of the concentration is contributed to by each cluster. Such a plot is shown in Figure 15.10. It is now easy to see for example that many of the peaks in SO_2 are associated with cluster 6 (power station sources from the east), seen in Figure 15.9. Cluster 6 is particularly prominent during springtime, but those sources also make important contributions through the whole year.

16 The polarAnnulus function

16.1 Purpose

see also polarFreq polarPlot percentileRose pollutionRose

The polarAnnulus function provides a way in which to consider the temporal aspects of a pollutant concentration by wind direction. This is another means of visualising diurnal, day of week, seasonal and trend variations. Plotting as an annulus, rather than a circle avoids to some extent the difficulty in interpreting values close to the origin. These plots have the capacity to display potentially important information regarding sources; particularly if more than one pollutant is available.

16.2 Options available

The polarAnnulus function has the following options:

`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.

16.3 Example of use

We apply the four variations of the `polarAnnulus` plot to PM_{10} concentrations at Marylebone Road. Figure 16.1 shows the different temporal components. Similar to other analyses for PM_{10}, the trend plot show that concentrations are dominated by southerly winds and there is little overall change in concentrations from 1998 to 2005, as shown by the red colouring over the period. The seasonal plot shows that February/March is important for easterly winds, while the summer/late summer period is more important for southerly and south-westerly winds. The day of the week plot clearly shows concentrations to be elevated for during weekdays but not weekends — for all wind directions. Finally, the diurnal plot highlights that higher concentrations are observed from 6 am to 6 pm.

Interestingly, the plot for NO_x and CO (not shown, but easily produced) did not show such a strong contribution for south-easterly winds. This raises the question as to whether the higher particle concentrations seen for these wind directions are dominated by different sources (i.e. not the road itself). One explanation is that during easterly flow, concentrations are strongly affected by long-range transport. However, as shown in the diurnal plot in Figure 16.1, the contribution from the south-east also has a sharply defined profile — showing very low concentrations at night, similar to the likely contribution from the road. This type of profile might not be expected from a long-range source where emissions are well-mixed and secondary particle formation has had time to occur. The same is also true for the day of the week plot, where there is little evidence of 'smeared-out' long-range transport sources. These findings may suggest a different, local source of PM_{10} that is not the road itself. Clearly, a more detailed analysis would be required to confirm the patterns shown; but it does highlight the benefit of being able to analyse data in different ways.

Where there is interest in considering the wind direction dependence of concentrations, it can be worth filtering for wind speeds. At low wind speed with wind direction becomes highly variable (and is often associated with high pollutant concentrations). Therefore, for some situations it might be worth considering removing the very low wind speeds. The code below provides two ways of doing this using the `subset` function. The first selects data where the wind speed is > 2 m s^{-1}. The second part shows how to select wind speeds greater than the 10th percentile, using the `quantile` function. The latter way of selecting is quite useful, because it is known how much data are selected i.e. in this case 90 %. It is worth experimenting with different values because it is also important not to lose information by ignoring wind speeds that provide useful information.

```
## wind speed >2
polarAnnulus(subset(mydata, ws > 2), poll="pm10", type = "hour")
## wind speed > 10th percentile
polarAnnulus(subset(mydata, ws > quantile(ws, probs = 0.1, na.rm = TRUE)),
             poll="pm10", type = "hour")
```

```
data(mydata)
polarAnnulus(mydata, poll = "pm10", period = "trend", main = "Trend")
polarAnnulus(mydata, poll = "pm10", period = "season", main = "Season")
polarAnnulus(mydata, poll = "pm10", period = "weekday", main = "Weekday")
polarAnnulus(mydata, poll = "pm10",period = "hour", main = "Hour")
```

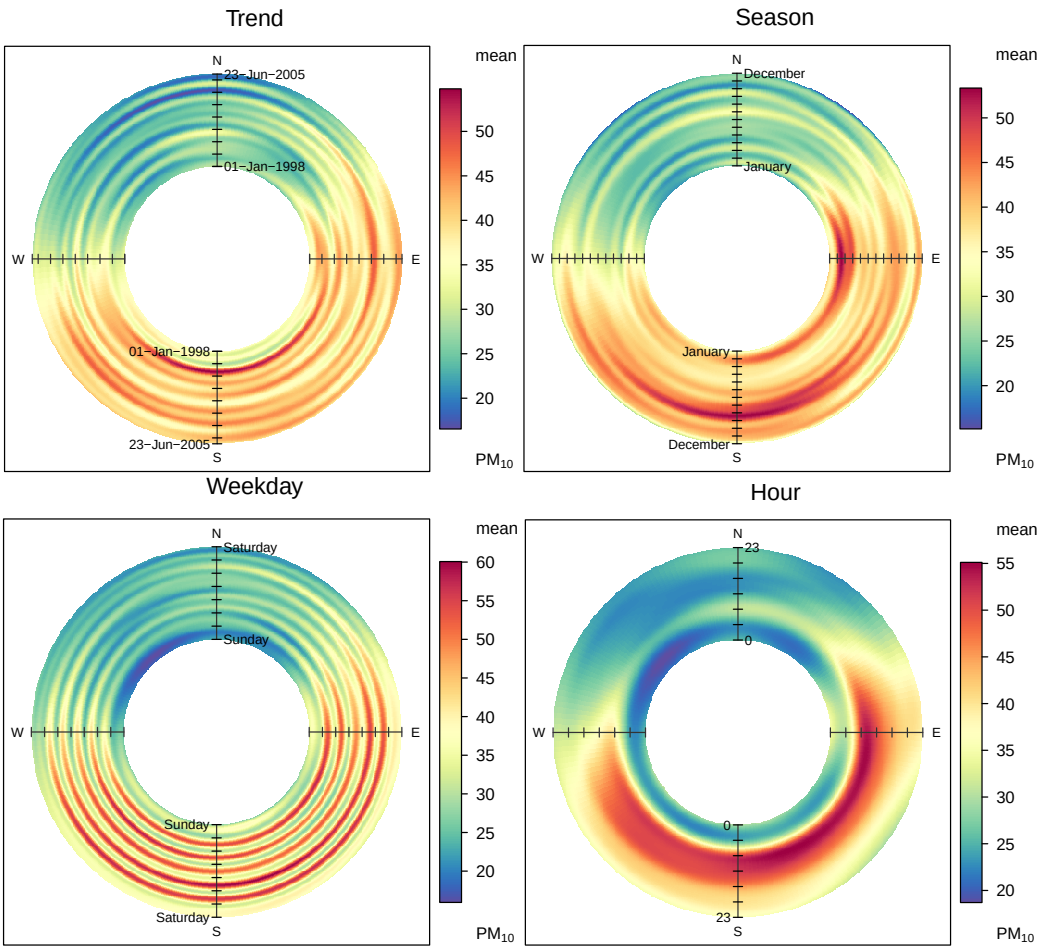


FIGURE 16.1 Examples of the `polarAnnulus` function applied to Marylebone Road

17 The timePlot and timeProp functions

17.1 Purpose

see also
smoothTrend
TheilSen
timeVariation
scatterPlot

The `timePlot` function is designed to quickly plot time series of data, perhaps for several pollutants or variables. This is, or should be, a very common task in the analysis of air pollution. In doing so, it is helpful to be able to plot several pollutants at the same time (and maybe other variables) and quickly choose the time periods of interest. It will plot time series of type `Date` and hourly and high time resolution data.

The function offers fine control over many of the plot settings such as line type, colour and width. If more than one pollutant is selected, then the time series are shown in a compact way in different panels with different scales. Sometimes it is useful to get and idea of whether different variables 'go up and down' together. Such comparisons in `timePlot` are made easy by setting `group = TRUE`, and maybe also `normalise = "mean"`. The latter setting divides each variable by its mean value, thus enabling several variables to be plotted together using *the same scale*. The `normalise` option

will also take a date as a string (in British format dd/mm/YYYY), in which case all data are normalise to equal 100 at that time. Normalising data like this makes it easy to compare time series on different scales e.g. emissions and ambient measurements.

`timePlot` works very well in conjunction with `selectByDate`, which makes it easy to select specific time series intervals. See (§31.1) for examples of how to select parts of a data frame based on the date.

Another useful feature of `timePlot` is the ability to average the data in several ways. This makes it easy, for example, to plot daily or monthly means from hourly data, or hourly means from 15-minute data. See the option `avg.time` for more details and (§31.4) where a full description of time averaging of data frames is given.

17.2 Options available

`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`.

`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 aer 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-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.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.

`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.

17.3 Example of use

A full set of examples is shown in the help pages — see ?timePlot for details. At the basic level, concentrations are shown using a simple call e.g. to plot time series of

```
data(mydata)
timePlot(selectByDate(mydata, year = 2003, month = "aug"),
         pollutant = c("nox", "o3", "pm25", "pm10", "ws"))
```

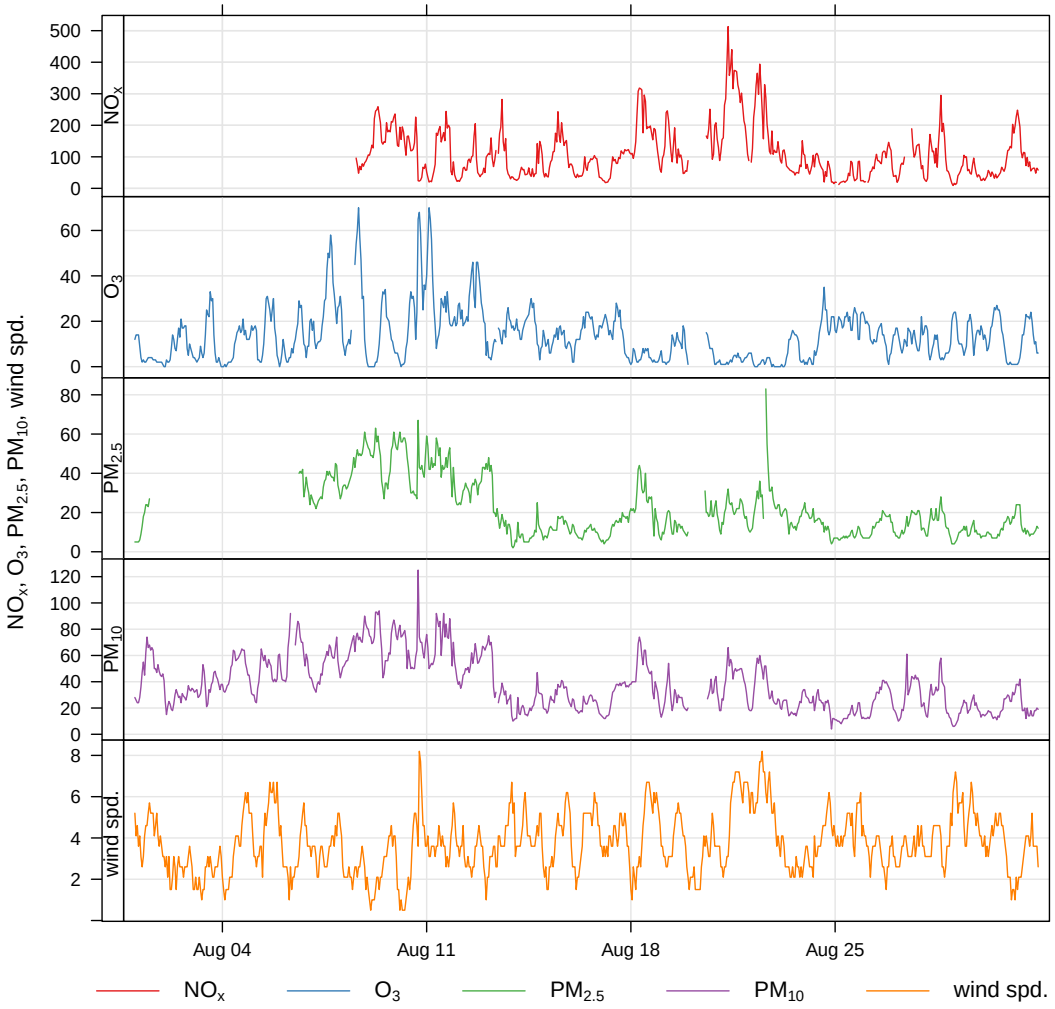


FIGURE 17.1 Time series for several variables using the `timePlot` and the `selectByDate` functions. The data shown are for August 2003.

NO_x and O_3 in separate panels with their own scales.

```
timePlot(mydata, pollutant = c("nox", "o3"))
```

Often it is necessary to only consider part of a time series and using the **openair** function `selectByDate` makes it easy to do this. Some examples are shown below.

To plot data only for 2003:

```
timePlot(selectByDate(mydata, year = 2003), pollutant = c("nox", "o3"))
```

Plots for several pollutants for August 2003, are shown in Figure 17.1.

Some other examples (not plotted) are:

```
timePlot(mydata, pollutant = c("nox", "no2", "co", "so2", "pm10"),
         avg.time = "year", normalise = "1/1/1998", lwd = 4, lty = 1,
         group = TRUE, ylim = c(0, 120))
```

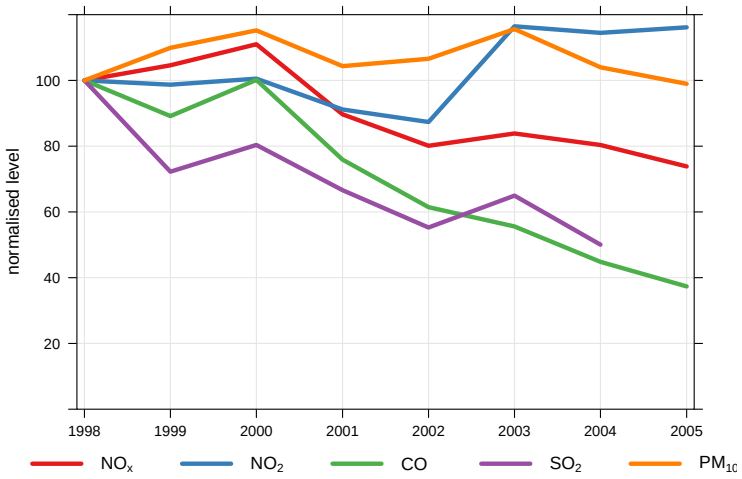


FIGURE 17.2 An example of normalising time series data to fix values to equal 100 at the beginning of 1998.

```
## plot monthly means of ozone and no2
timePlot(mydata, pollutant = c("o3", "no2"), avg.time = "month")

## plor 95th percentile monthly concentrations
timePlot(mydata, pollutant = c("o3", "no2"), avg.time = "month",
         statistic = "percentile", percentile = 95)

## plot the number of valid records in each 2-week period
timePlot(mydata, pollutant = c("o3", "no2"), avg.time = "2 week",
         statistic = "frequency")
```

An example of normalising data is shown in Figure 17.2. In this plot we have:

- Averaged the data to annual means;
- Chosen to normalise to the beginning of 2008;
- Set the line width to 4 and the line type to 1 (continuous line);
- Chosen to group the data in one panel.

Figure 17.2 shows that concentrations of NO_2 and O_3 have increased over the period 1998–2005; SO_2 and CO have shown the greatest reductions (by about 60%), whereas NO_x concentrations have decreased by about 20%.

Another example is grouping pollutants from *several sites* on one plot. It is easy to import data from several sites and to plot the data in separate panels e.g.

```
## import data from 3 sites
thedata <- importAURN(site = c("kc1", "my1", "nott"), year = 2005:2010)

## plot it
timePlot(test, pollutant = "nox", type = "site", avg.time = "month")
```

Using the code above it is also possible to include several species. But what if we wanted to plot NO_x concentrations across all sites in one panel? To do this we need

to re-organise the data, as described in Section 5.4. An example of how to do this is shown below. Note, in order to make referring to the columns easier, we will drop the full (long) site name and use the site code instead.

```
## first drop site name
thedata <- subset(thedata, select = -site)
## now reshape the data using the reshape package
thedata <- melt(thedata, id.vars = c("date", "code"))
thedata <- dcast(thedata, ... ~ code + variable)
```

The final step will make columns of each site/pollutant combination e.g. 'KC1_nox', 'KC1_pm10' and so on. It is then easy to use any of these names to make the plot:

```
timePlot(thedata, pollutant = c("KC1_nox", "MY1_nox", "NOTT_nox"),
         avg.time = "month", group = TRUE)
```

An alternative way of selecting all columns containing the character 'nox' is to use the grep command (see page 35). For example:

```
timePlot(thedata, pollutant =  names(thedata)[grep(pattern = "nox", names(thedata))],
         avg.time = "month", group = TRUE)
```

17.3.1 The timeProp function

The timeProp ('time proportion') function shows time series plots as stacked bar charts. For a particular time, proportions of a chosen variable are shown as a stacked bar chart. 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 (see Section 15) and trajCluster (see Section 26.3) that consider local and regional (back trajectory) cluster analysis respectively. However, the function has more general use for understanding time series data. 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.

The timeProp function has the following options:

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".

box.width — The width of the boxes for panel.boxplot. A value of 1 means that there is no gap between the boxes.

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".

```
timeProp(selectByDate(mydata, year = 2003), pollutant = "so2", avg.time = "3 day",
         proportion = "wd", date.breaks = 10, key.position = "top",
         key.columns = 8, ylab = "so2 (ug/m3)")
```

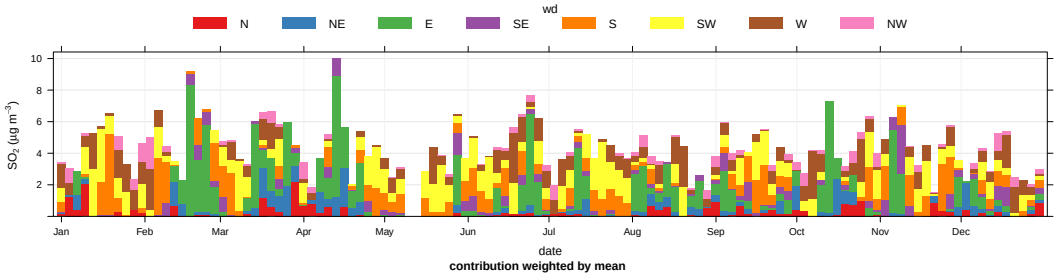


FIGURE 17.3 `timeProp` plot for SO_2 concentrations in 2003. The data are categorised into 8 wind sectors for 3-day averages.

`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.

An example of the `timeProp` function is shown in Figure 17.3. In this example SO_2 concentrations are considered for 2003 (using the `selectByDate` function). The averaging period is set to 3 days and the mean concentration is plotted and the proportion contribution by wind sector is given. Other options are chosen to place the key at the top and choose the number of columns used in the key. It is apparent from Figure 17.3 that the highest SO_2 concentrations are dominated by winds from an easterly sector, but actually occur throughout the year.

Note that `proportion` can be an existing categorical (i.e. factor or character) variable in a data frame. If a numeric variable is supplied, then it is typically cut into four quantile levels. So, for example, the plot below would show four intervals of wind speed, which would help show the wind speed conditions that control high SO_2 concentration — and importantly, when they occur.

One of the key uses of `timeProp` is to post-process cluster analysis data. Users should consider the uses of `timeProp` for cluster analysis shown in Section 15 and Section 26.3. In both these cases the cluster analysis yields a categorical output directly i.e. cluster, which lends itself to analysis using `timeProp`.

```
timeProp(selectByDate(mydata, year = 2003), pollutant = "so2",
         avg.time = "3 day", proportion = "ws", date.breaks = 10,
         key.position = "top", key.columns = 4)
```

18 The calendarPlot function

18.1 Purpose

Sometimes it is useful to visualise data in a familiar way. Calendars are the obvious way to represent data for data on the time scale of days or months. The `calendarPlot` function provides an effective way to visualise data in this way by showing daily concentrations laid out in a calendar format. The concentration of a species is shown by its colour. The data can be shown in different ways. By default `calendarPlot` overlays the day of the month. However, if wind speed and wind direction are available then an arrow can be shown for each day giving the vector-averaged wind direction. In addition, the arrow can be scaled according to the wind speed to highlight both the direction and strength of the wind on a particular day, which can help show the influence of meteorology on pollutant concentrations.

`calendarPlot` can also show the daily mean concentration as a number on each day and can be extended to highlight those conditions where daily mean (or maximum etc.) concentrations are above a particular threshold. This approach is useful for highlighting daily air quality limits e.g. when the daily mean concentration is greater than 50 µg m^{-3}.

The `calendarPlot` function can also be used to plot categorical scales. This is useful for plotting concentrations expressed as an air quality index i.e. intervals of concentrations that are expressed in ways like 'very good', 'good', 'poor' and so on.

18.2 Options available

`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`.

`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`.

`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.

`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.

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

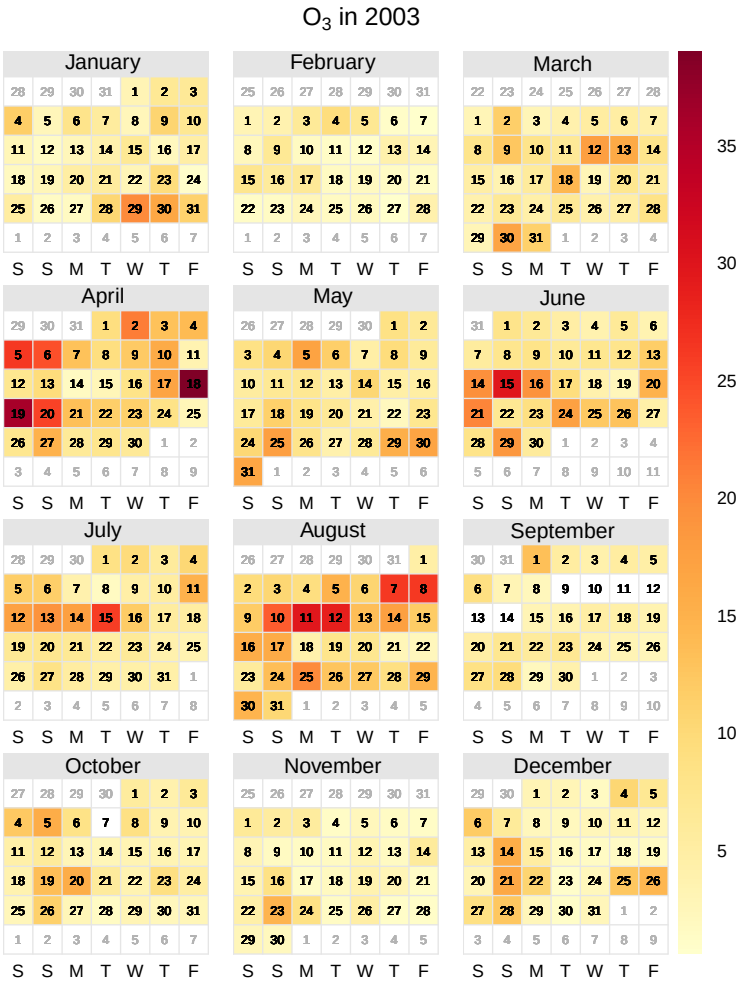


FIGURE 18.1 `calendarPlot` for O_3 concentrations in 2003.

`...` 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.

18.3 Example of use

The function is called in the usual way. As a minimum, a data frame, pollutant and year is required. So to show O_3 concentrations for each day in 2003 (Figure 18.1).

It is sometimes useful to annotate the plots with other information. It is possible to show the daily mean wind angle, which can also be scaled to wind speed. The idea here being to provide some information on meteorological conditions on each day. Another useful option is to set `annotate = "value"` in which case the daily concentration will be shown on each day. Furthermore, it is sometimes useful to highlight particular values more clearly. For example, to highlight daily mean PM_{10} concentrations above 50 µg m^{-3}. This is where setting `lim` (a concentration limit) is useful. In setting `lim` the user can then differentiate the values below and above `lim` by colour of text, size of text and type of text e.g. plain and bold.

```
data(mydata) ## make sure openair 'mydata' loaded fresh
calendarPlot(mydata, pollutant = "pm10", annotate = "value", lim =50,
             cols = "Purples", col.lim = c("black",  "orange"),
             layout = c(4, 3))
```

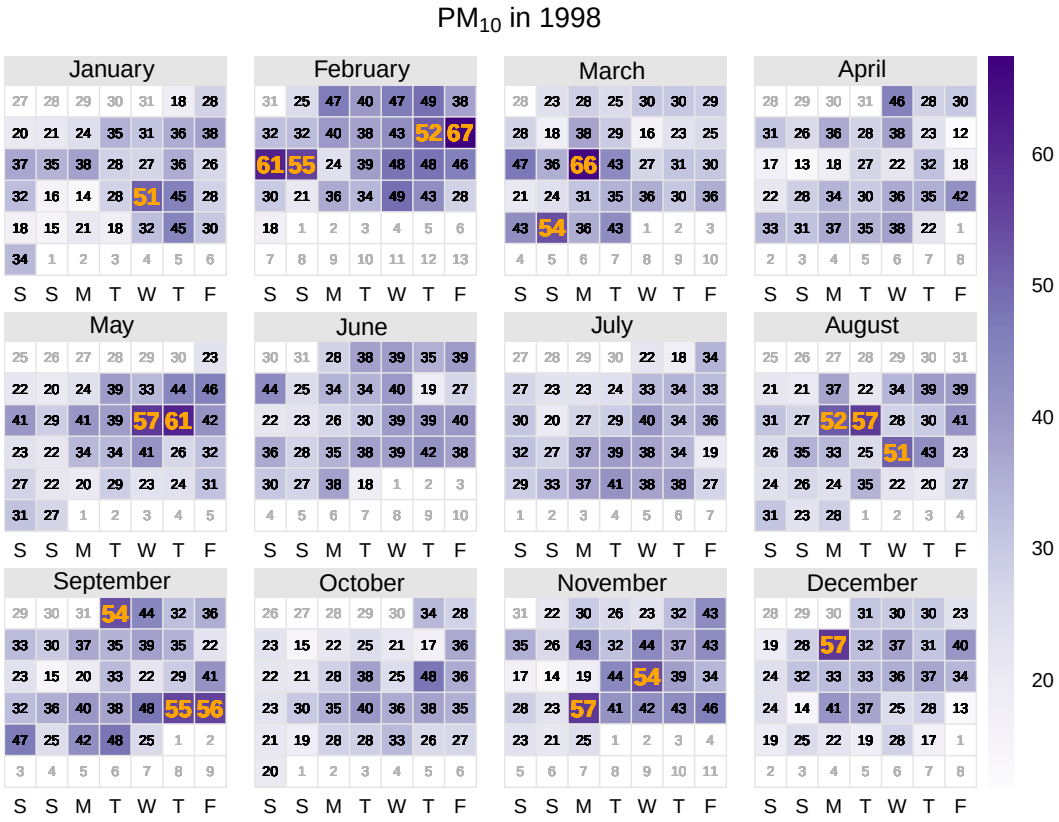


FIGURE 18.2 `calendarPlot` for PM_{10} concentrations in 2003 with annotations highlighting those days where the concentration of PM_{10} >50 µg m^{-3}. The numbers show the PM_{10} concentration in µg m^{-3}.

Figure 18.2 highlights those days where PM_{10} concentrations exceed 50 µg m^{-3} by making the annotation for those days bigger, bold and orange. Plotting the data in this way clearly shows the days where PM_{10} >50 µg m^{-3}.

Other **openair** functions can be used to plot other statistics. For example, `rollingMean` could be used to calculate rolling 8-hour mean O_3 concentrations. Then, `calendarPlot` could be used with `statistic = "max"` to show days where the maximum daily rolling 8-hour mean O_3 concentration is greater than a certain threshold e.g. 100 or 120 µg m^{-3}.

To show wind angle, scaled to wind speed (Figure 18.3).

Note again that `selectByDate` can be useful. For example, to plot select months:

```
calendarPlot(selectByDate(mydata, year = 2003, month = c("jun", "jul", "aug")),
             pollutant = "o3", year = 2003)
```

Figure 18.4 shows an example of plotting data with a categorical scale. In this case the options `labels` and `breaks` have been used to define concentration intervals and their descriptions. Note that `breaks` needs to be one longer than `labels`. In the example in Figure 18.4 the first interval ('Very low') is defined as concentrations from 0 to 50 (ppb), 'Low' is 50 to 100 and so on. Note that the upper value of `breaks` should

```
calendarPlot(mydata, pollutant = "o3", year = 2003, annotate = "ws")
```

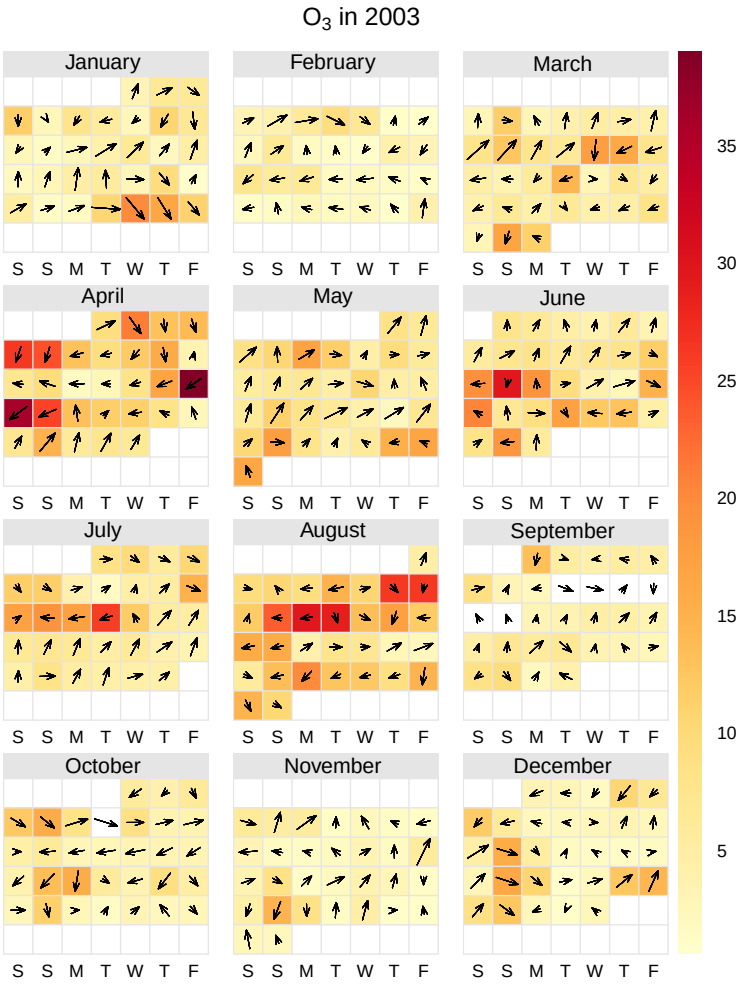


FIGURE 18.3 calendarPlot for O_3 concentrations in 2003 with annotations showing wind angle scaled to wind speed i.e. the longer the arrow, the higher the wind speed. It shows for example high O_3 concentrations on the 17 and 18th of April were associated with strong north-easterly winds.

be a number greater than the maximum value contained in the data to ensure that it is encompassed. In the example given in Figure 18.4 the maximum daily concentration is plotted. These types of plots are very useful for considering national or international air quality indexes.

The user can explicitly set each colour interval:

```
calendarPlot(mydata, pollutant = "no2", breaks = c(0, 50, 100, 150, 1000),
             labels = c("Very low", "Low", "High", "Very High"),
             cols = c("lightblue", "forestgreen", "yellow",  "red"),
             statistic = "max")
```

```
calendarPlot(mydata, pollutant = "no2", breaks = c(0, 50, 100, 150, 1000),
             labels = c("Very low", "Low", "High", "Very High"),
             cols = "increment", statistic = "max")
```

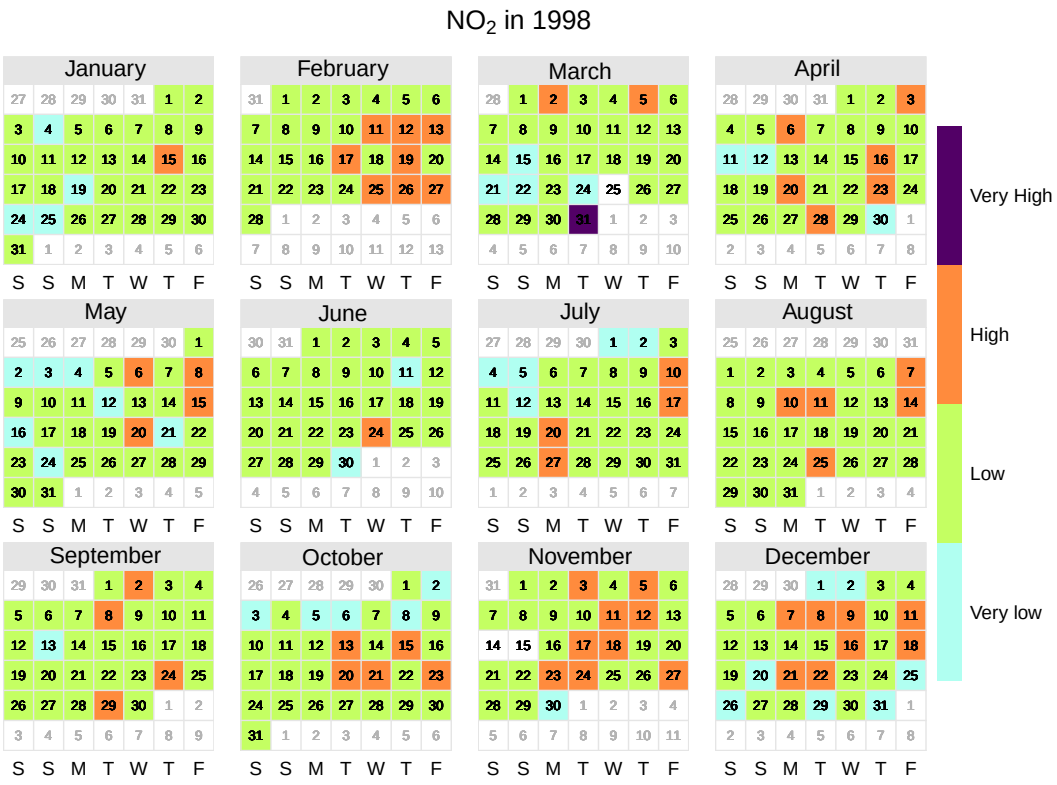


FIGURE 18.4 `calendarPlot` for NO_2 concentrations in 2003 with a user-defined categorical scale.

Note that in the case of categorical scales it is possible to define the breaks and labels first and then make the plot. For example:

```
breaks <- c(0, 34, 66, 100, 121, 141, 160, 188, 214, 240, 500)
labels <- c("Low.1", "Low.2", "Low.3", "Moderate.4", "Moderate.5", "Moderate.6",
            "High.7", "High.8", "High.9", "Very High.10")
calendarPlot(mydata, pollutant = "no2", breaks = breaks, labels = labels,
             cols = "jet", statistic = "max")
```

It is also possible to first use `rollingMean` to calculate statistics. For example, if one was interested in plotting the maximum daily rolling 8-hour mean concentration, the data could be prepared and plotted as follows.

```
## makes a new field 'rolling8o3'
dat <- rollingMean(dat, pollutant = "o3", hours = 8)
breaks <- c(0, 34, 66, 100, 121, 141, 160, 188, 214, 240, 500)
labels <- c("Low.1", "Low.2", "Low.3", "Moderate.4", "Moderate.5", "Moderate.6",
            "High.7", "High.8", "High.9", "Very High.10")
calendarPlot(mydata, pollutant = "rolling8o3", breaks = breaks, labels = labels,
             cols = "jet", statistic = "max")
```

The UK has an air quality index for O_3, NO_2, PM_{10} and $PM_{2.5}$ described in detail at http://uk-air.defra.gov.uk/air-pollution/daqi and COMEAP (2011). The air quality index is shown in Table 18.1. The index is most relevant to air quality forecasting, but is used widely for public information. Most other countries have similar indexes. Note that the indexes are calculated for different averaging times dependent on the pollutant: rolling 8-hour mean for O_3, hourly means for NO_2 and a fixed 24-hour mean for PM_{10} and $PM_{2.5}$.

TABLE 18.1 The UK daily air quality index (values in µg m^{-3}).

Band	Description	NO_2	O_3	PM_{10}	$PM_{2.5}$
1	Low	0–66	0–33	0–16	0–11
2	Low	67–133	34–65	17–33	12–23
3	Low	134–199	66–99	34–49	24–34
4	Moderate	200–267	100–120	50–58	35–41
5	Moderate	268–334	121–140	59–66	42–46
6	Moderate	335–399	141–159	67–74	47–52
7	High	400–467	160–187	75–83	53–58
8	High	468–534	188–213	84–91	59–64
9	High	535–599	214–239	92–99	65–69
10	Very High	600 or more	240 or more	100 or more	70 or more

In the code below the labels and breaks are defined for each pollutant in Table 18.1 to make it easier to use the index in the `calendarPlot` function.

```
## the labels - same for all species
labels <- c("1 - Low", "2 - Low", "3 - Low", "4 - Moderate", "5 - Moderate",
            "6 - Moderate", "7 - High", "8 - High", "9 - High", "10 - Very High")
o3.breaks <-c(0, 34, 66, 100, 121, 141, 160, 188, 214, 240, 500)
no2.breaks <- c(0, 67, 134, 200, 268, 335, 400, 468, 535, 600, 1000)
pm10.breaks <- c(0, 17, 34, 50, 59, 67, 75, 84, 92, 100, 1000)
pm25.breaks <- c(0, 12, 24, 35, 42, 47, 53, 59, 65, 70, 1000)
```

Remember it is necessary to use the correct averaging time. Assuming data are imported using `importAURN` or `importKCL` then the units will be in µg m^{-3}— if not the

user should ensure this is the case. Note that rather than showing the day of the month (the default), `annotate = "value"` can be used to show the actual numeric value on each day. In this way, the colours represent the categorical interval the concentration on a day corresponds to *and* the actual value itself is shown.

```
## import test data
dat <- importAURN(site = "kc1", year = 2010)

## no2 index example
calendarPlot(dat, year = 2010, pollutant = "no2", labels = labels,
             breaks = no2.breaks, statistic = "max", cols = "jet")

## for PM10 or PM2.5 we need the daily mean concentration
calendarPlot(dat, year = 2010, pollutant = "pm10", labels = labels,
             breaks = pm10.breaks, statistic = "mean", cols = "jet")

## for ozone, need the rolling 8-hour mean
dat <- rollingMean(dat, pollutant = "o3", hours = 8)
calendarPlot(dat, year = 2010, pollutant = "rolling8o3", labels = labels,
             breaks = o3.breaks, statistic = "max", cols = "jet")
```

19 The TheilSen function

19.1 Purpose

see also
smoothTrend
timePlot

Calculating trends for air pollutants is one of the most important and common tasks that can be undertaken. Trends are calculated for all sorts of reasons. Sometimes it is useful to have a general idea about how concentrations might have changed. On other occasions a more definitive analysis is required; for example, to establish statistically whether a trend is significant or not. The whole area of trend calculation is a complex one and frequently trends are calculated with little consideration as to their validity. Perhaps the most common approach is to apply linear regression and not think twice about it. However, there can be many pitfalls when using ordinary linear regression, such as the assumption of normality, autocorrelation etc.

One commonly used approach for trend calculation in studies of air pollution is the non-parametric *Mann-Kendall* approach (Hirsch et al. 1982). Wilcox (2010) provides an excellent case for using 'modern methods' for regression including the benefits of non-parametric approaches and bootstrap simulations. Note also that the all the regression parameters are estimated through bootstrap resampling.

The Theil-Sen method dates back to 1950, but the basic idea pre-dates 1950 (Theil 1950; Sen 1968). It is one of those methods that required the invention of fast computers to be practical. The basic idea is as follows. Given a set of n x, y pairs, the slopes between all pairs of points are calculated. Note, the number of slopes can increase by $\approx n^2$ so that the number of slopes can increase rapidly as the length of the data set increases. The Theil-Sen estimate of the slope is the median of all these slopes. The advantage of the using the Theil-Sen estimator is that it tends to yield accurate confidence intervals even with non-normal data and heteroscedasticity (non-constant error variance). It is also resistant to outliers — both characteristics can be important in air pollution. As previously mentioned, the estimates of these parameters can be made more robust through bootstrap-resampling, which further adds to the computational burden, but is not an issue for most time series which are expressed either as monthly or annual means. Bootstrap resampling also provides the estimate of p for the slope.

An issue that can be very important for time series is dependence or *autocorrelation* in the data. Normal (in the statistical sense) statistics assume that data are independent,

but in time series this is rarely the case. The issue is that neighbouring data points are similar to one another (correlated) and therefore not independent. Ignoring this dependence would tend to give an overly optimistic impression of uncertainties. However, taking account of it is far from simple. A discussion of these issues is beyond the aims of this report and readers are referred to standard statistical texts on the issue. In **openair** 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.

There is a temptation when considering trends to use all the available data. Why? Often it is useful to consider specific periods. For example, is there any evidence that concentrations of NO_x have decreased since 2000? Clearly, the time period used depends on both the data and the questions, but it is good to be aware that considering subsets of data can be very insightful.

Another aspect is that almost all trends are shown as mean concentration versus time; typically by year. Such analyses are very useful for understanding how concentrations have changed through time and for comparison with air quality limits and regulations. However, if one is interested in *understanding* why trends are as they are, it can be helpful to consider how concentrations vary in other ways. The trend functions in **openair** do just this. Trends can be plotted by day of the week, month, hour of the day, by wind direction sector and by different wind speed ranges. All these capabilities are easy to use and their effectiveness will depend on the situation in question. One of the reasons that trends are not considered in these many different ways is that there can be a considerable overhead in carrying out the analysis, which is avoided by using these functions. Few, for example, would consider a detailed trend analysis by hour of the day, ensuring that robust statistical methods were used and uncertainties calculated. However, it can be useful to consider how concentrations vary in this way. It may be, for example, that the hours around midday are dominated by heavy vehicle emissions rather than by cars — so is the trend for a pollutant different for those hours compared with say, hours dominated by other vehicle types? Similarly, a much more focussed trend analysis can be done by considering different wind direction, as this can help isolate different source influences.

The `TheilSen` function is typically used to determine trends in pollutant concentrations over several years. However, it can be used to calculate the trend in any numeric variable. It calculates monthly mean values from daily, hourly or higher time resolution data, as well as working directly with monthly means. Whether it is meaningful to calculate trends over shorter periods of time (e.g. 2 years) depends very much on the data. It may well be that statistically significant trends can be detected over relatively short periods but it is another matter whether it matters. Because seasonal effects can be important for monthly data, there is the option to deseasonalise the data first. The `timeVariation` function are both useful to determine whether there is a seasonal cycle that should be removed.

Note also 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 $p < 0.1 = +$.

19.2 Options available

The `TheilSen` function has the following options:

`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 linearly interpolated because `stl` cannot handle missing 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.

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 (aggregating the data using `avg.time`. A value of zero means that all available data will be used in a particular period regardless if of the number of values available. Conversely, a value of 100 will mean that all data will need to be present for the average to be calculated, else it is recorded as `NA`.

`alpha` For the confidence interval calculations of the slope. The default is 0.05. To show 99% 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

`line.col` Colour name for the slope and uncertainty estimates

`text.col` Colour name for the slope/uncertainty numeric estimates

`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 years there will be 75 units but the actual time span will be 6 years i.e. year 1 is used as a reference year. Things are slightly different for monthly values e.g. `avg.time = "month"`, which will use the total number of months as a basis of the time span and is therefore able to deal with partial years. There can be slight differences in the 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.

`...` 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.

```
TheilSen(mydata, pollutant = "o3", ylab = "ozone (ppb)", deseason = TRUE)
```

```
## [1] "Taking bootstrap samples. Please wait."
```

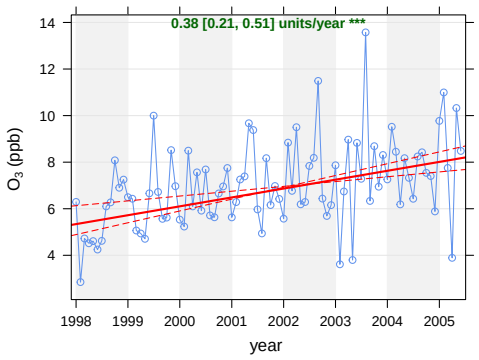


FIGURE 19.1 Trends in ozone at Marylebone Road. The plot shows the deseasonalised monthly mean concentrations of O_3. The solid red line shows the trend estimate and the dashed red lines show the 95 % confidence intervals for the trend based on resampling methods. The overall trend is shown at the top-left as 0.38 (ppb) per year and the 95 % confidence intervals in the slope from 0.21–0.51 ppb/year. The $***$ show that the trend is significant to the 0.001 level.

19.3 Example of use

We first show the use of the `TheilSen` function by applying it to concentrations of O_3. The function is called as shown in Figure 19.1.

Because the function runs simulations to estimate the uncertainty in the slope, it can take a little time for all the calculations to finish. These printed results show that in this case the trend in O_3 was +0.38 units (i.e. ppb) per year as an average over the entire period. It also shows the 95 % confidence intervals in the trend ranged between 0.21 to 0.51 ppb/year. Finally, the significance level in this case is very high; providing very strong evidence that concentrations of O_3 increased over the period. The plot together with the summary results is shown in Figure 19.1. Note that if one wanted to display the confidence intervals in the slope at the 99 % confidence intervals, the code would be Figure 19.2.

```
TheilSen(mydata, pollutant = "o3", ylab = "ozone (ppb)", alpha = 0.01)
```

Sometimes it is useful to consider a subset of data, perhaps by excluding some years. This is easy with the `subset` function. The following code calculates trends for years greater than 1999 i.e. from 2000 onwards.

```
TheilSen(subset(mydata, format(date, "%Y") > 1999), pollutant = "o3",
         ylab = "ozone (ppb)")
```

It is also possible to calculate trends in many other ways e.g. by wind direction. Considering how trends vary by wind direction can be extremely useful because the influence of different sources invariably depends on the direction of the wind. The `TheilSen` function splits the wind direction into 8 sectors i.e. N, NE, E etc. The Theil-Sen slopes are then calculated for each direction in turn. This function takes rather longer to run because the simulations need to be run eight times in total. Considering concentrations of O_3 again, the output is shown in Figure 19.2. Note

that this plot is specifically laid out to assist interpretation, with each panel located at the correct point on the compass. This makes it easy to see immediately that there is essentially no trend in O_3 for southerly winds i.e. where the road itself has the strongest influence. On the other hand the strongest evidence of increasing O_3 are for northerly winds, where the influence of the road is much less. The reason that there is no trend in O_3 for southerly winds is that there is always a great excess of NO, which reacts with O_3 to form NO_2. At this particular location it will probably take many more years before O_3 concentrations start to increase when the wind direction is southerly. Nevertheless, there will always be *some* hours that do not have such high concentrations of NO.

The option `slope.percent` can be set to express slope estimates as a percentage change per year. This is useful for comparing slopes for sites with very different concentration levels and for comparison with emission inventories. The percentage change uses the concentration at the beginning and end months to express the mean slope.

The trend, T is defined as:

$$T[\%.yr^{-1}] = 100.\left(\frac{C_{End}}{C_{Start}} - 1\right)/N_{years} \quad (7)$$

where C_{End} and C_{Start} are the mean concentrations for the end and start date, respectfully. N_{years} is the number of years (or fractions of) the time series spans.

```
TheilSen(mydata, pollutant = "o3", deseason = TRUE,
            slope.percent = TRUE)
```

The `TheilSen` function was written to work with hourly data, which is then averaged into monthly or annual data. However, it is realised that users may already have data that is monthly or annual. The function can therefore accept as input monthly or annual data directly. *However, it is necessary to ensure the date field is in the correct format*. Assuming data in an Excel file in the format dd/mm/YYYY (e.g. 23/11/2008), it is necessary to convert this to a date format understood by R, as shown below. Similarly, if annual data were available, get the dates in formats like '2005-01-01', '2006-01-01'... and make sure the date is again formatted using `as.Date`. Note that if dates are pre-formatted as YYYY-mm-dd, then it is sufficient to use `as.Date` without providing any format information because it is already in the correct format.

```
mydata$date = as.Date(mydata$date, format = "%d/%m/%Y")
```

Finally, the `TheilSen` function can consider trends at different sites, provided the input data are correctly formatted. For input, a data frame with three columns is required: date, pollutant and *site*. The call would then be, for example:

```
TheilSen(mydata, pollutant = "no2", type = "site")
```

19.4 Output

The `TheilSen` function provides lots of output data for further analysis or adding to a report. To obtain it, it is necessary to read it into a variable:

```
TheilSen(mydata, pollutant = "o3", type = "wd", deseason = TRUE,
         ylab = "ozone (ppb)")
```

```
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
```

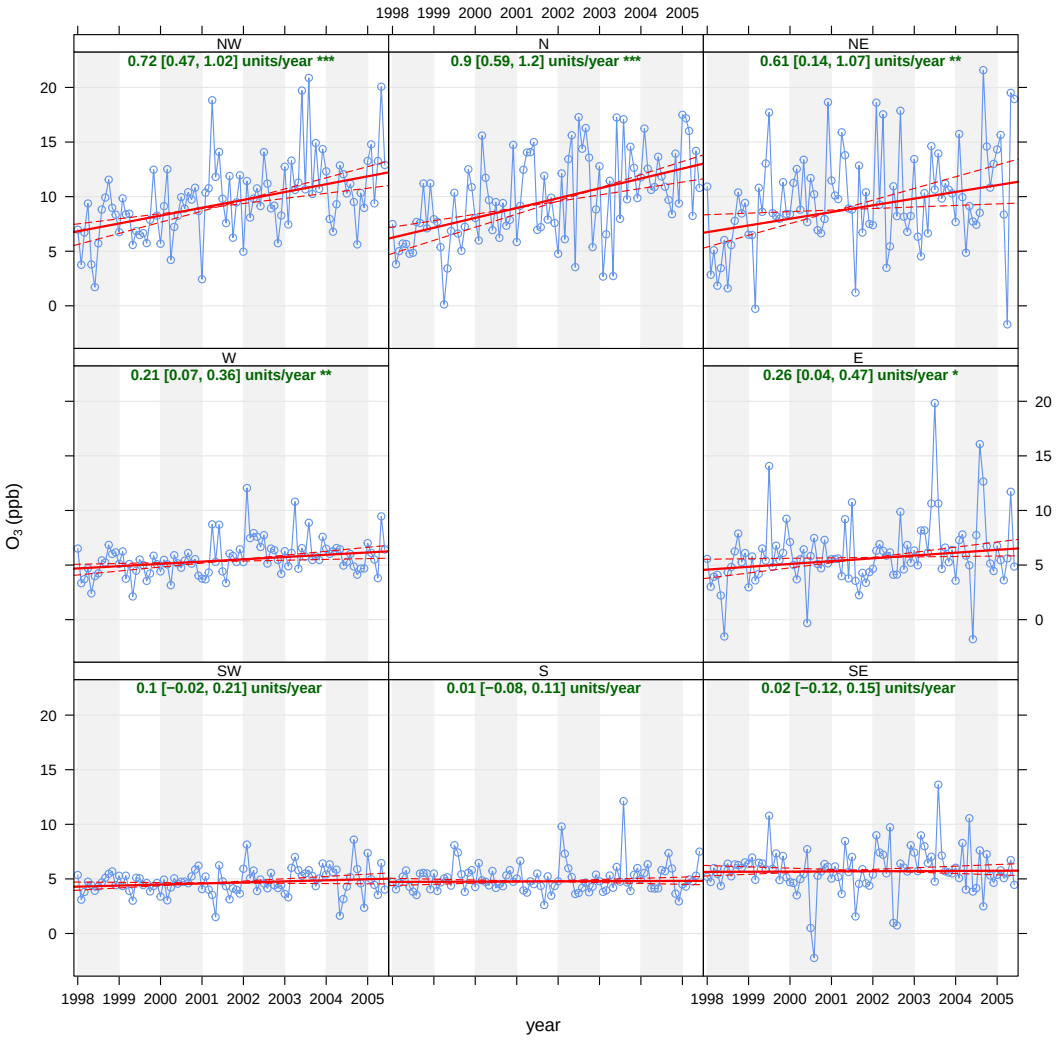


FIGURE 19.2 Trends in ozone at Marylebone Road split by eight wind sectors. The TheilSen function will automatically organise the separate panels by the different compass directions.

```
MKresults <-  TheilSen(mydata, pollutant = "o3", deseason = TRUE, type = "wd")
```

```
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
## [1] "Taking bootstrap samples. Please wait."
```

This returns a list of two data frames containing all the monthly mean values and trend statistics and an aggregated summary. The first 6 lines are shown next:

```
head(MKresults$data[[1]])
```

```
##   wd       date      conc         a            b   upper.a     upper.b  lower.a
## 1  E 1998-01-01  5.563380 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
## 2  E 1998-02-01  3.016144 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
## 3  E 1998-03-01  3.934497 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
## 4  E 1998-04-01  4.106889 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
## 5  E 1998-05-01  2.214696 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
## 6  E 1998-06-01 -1.540651 -2.655269 0.0007077532 -9.447407 0.001295295 4.363847
##        lower.b           p p.stars     slope intercept intercept.lower
## 1 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
## 2 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
## 3 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
## 4 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
## 5 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
## 6 0.0001144696 0.01669449       * 0.2583299 -2.655269        4.363847
##   intercept.upper      lower     upper slope.percent lower.percent
## 1       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
## 2       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
## 3       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
## 4       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
## 5       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
## 6       -9.447407 0.04178142 0.4727827      5.636794     0.7549229
##   upper.percent
## 1      12.44304
## 2      12.44304
## 3      12.44304
## 4      12.44304
## 5      12.44304
## 6      12.44304
```

Often only the trend statistics are required and not all the monthly values. These can be obtained by:

```
MKresults$data[[2]]
```

```
##   wd p.stars     conc          a            b    upper.a      upper.b   lower.a
## 1  E       * 5.993381  -2.655269 7.077532e-04  -9.447407 0.0012952951  4.363847
## 2  N     *** 9.794908 -19.024806 2.471079e-03 -28.871935 0.0032918234 -9.275425
## 3 NE      ** 9.695874 -10.454561 1.681889e-03 -24.644558 0.0029364214  4.391676
## 4 NW     *** 9.765453 -13.331253 1.970601e-03 -22.911529 0.0027893195 -5.581507
## 5  S         5.048152   4.335367 3.770842e-05   1.306559 0.0003018471  7.266568
## 6 SE         5.793174   5.214609 4.155925e-05   1.106889 0.0004074724  9.639716
## 7 SW         4.756367   1.576488 2.653631e-04  -1.925166 0.0005758312  5.343978
## 8  W      ** 5.621118  -1.154837 5.715179e-04  -6.029651 0.0009898862  2.988034
##         lower.b           p      slope  intercept intercept.lower
## 1  1.144696e-04 0.016694491 0.25832990  -2.655269        4.363847
## 2  1.611972e-03 0.000000000 0.90194378 -19.024806       -9.275425
## 3  3.865934e-04 0.006677796 0.61388943 -10.454561        4.391676
## 4  1.279310e-03 0.000000000 0.71926941 -13.331253       -5.581507
## 5 -2.160569e-04 0.811352254 0.01376357   4.335367        7.266568
## 6 -3.337383e-04 0.868113523 0.01516913   5.214609        9.639716
## 7 -6.210718e-05 0.113522538 0.09685754   1.576488        5.343978
## 8  2.037578e-04 0.003338898 0.20860402  -1.154837        2.988034
##   intercept.upper       lower     upper slope.percent lower.percent
## 1       -9.447407  0.04178142 0.4727827     5.6367943     0.7549229
## 2      -28.871935  0.58836992 1.2015155    14.4382219     8.1602260
## 3      -24.644558  0.14110659 1.0717938     9.0998946     1.6908374
## 4      -22.911529  0.46694823 1.0181016    10.5432502     6.2243175
## 5        1.306559 -0.07886078 0.1101742     0.2915387    -1.5594523
## 6        1.106889 -0.12181447 0.1487274     0.2689735    -1.9563639
## 7       -1.925166 -0.02266912 0.2101784     2.2575636    -0.4814195
## 8       -6.029651  0.07437159 0.3613084     4.4477743     1.4663558
##   upper.percent
## 1     12.443037
## 2     25.065293
## 3     19.898797
## 4     18.132332
## 5      2.507635
## 6      2.819954
## 7      5.302367
## 8      8.825500
```

In the results above the 'lower' and 'upper' fields provide the 95% (or chosen confidence interval using the `alpha` option) of the trend and 'slope' is the trend estimate expressed in units/year.

20 The smoothTrend function

20.1 Purpose

see also TheilSen timePlot

The `smoothTrend` function calculates smooth trends in the monthly mean concentrations of pollutants. In its basic use it will generate a plot of monthly concentrations and fit a smooth line to the data and show the 95 % confidence intervals of the fit. The smooth line is essentially determined using Generalized Additive Modelling using the `mgcv` package. This package provides a comprehensive and powerful set of methods for modelling data. In this case, however, the *model* is a relationship between time and pollutant concentration i.e. a trend. One of the principal advantages of this approach is that the amount of smoothness in the trend is optimised in the sense that it is neither too smooth (therefore missing important features) nor too variable (perhaps fitting 'noise' rather than real effects). Some background information on the use of this approach in an air quality setting can be found in Carslaw et al. (2007).

Appendix C considers smooth trends in more detail and considers how different

models can be developed that can be quite sophisticated. Readers should consider this section if they are considering trend analysis in more depth.

The user can select to deseasonalise the data first to provide a clearer indication of the overall trend on a monthly basis. The data are deseasonalised using the The `stl` function. The user may also select to use bootstrap simulations to provide an alternative method of estimating the uncertainties in the trend. In addition, the simulated estimates of uncertainty can account for autocorrelation in the residuals using a block bootstrap approach.

20.2 Options available

The `smoothTrend` function has the following options:

`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 linearly interpolated because `stl` cannot handle missing 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.

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.

20.3 Example of use

We apply the function to concentrations of O_3 and NO_2 using the code below. The first plot shows the smooth trend in raw O_3 concentrations, which shows a very clear seasonal cycle. By removing the seasonal cycle of O_3, a better indication of the trend is given, shown in the second plot. Removing the seasonal cycle is more effective for pollutants (or locations) where the seasonal cycle is stronger e.g. for ozone and background sites. Figure 20.1 shows the results of the simulations for NO_2 without the seasonal cycle removed. It is clear from this plot that there is little evidence of a seasonal cycle. The principal advantage of the smoothing approach compared with the Theil-Sen method is also clearly shown in this plot. Concentrations of NO_2 first decrease, then increase strongly. The trend is therefore not monotonic, violating the Theil-Sen assumptions. Finally, the last plot shows the effects of first deaseasonalising the data: in this case with little effect.

The `smoothTrend` function share many of the functionalities of the `TheilSen` function. Figure 20.2 shows the result of applying this function to O_3 concentrations. The code that produced Figure 20.2 was:

The `smoothTrend` function can easily be used to gain a large amount of information on trends easily. For example, how do trends in NO_2, O_3 and PM_{10} vary by season and wind sector. There are 8 wind sectors and four seasons i.e. 32 plots. In Figure 20.3 all three pollutants are chosen and two types (season and wind direction). We also reduce the number of axis labels and the line to improve clarity. There are numerous combinations of analyses that could be produced here and it is very easy to explore the data in a wide number of ways.

```
smoothTrend(mydata, pollutant = "o3", ylab = "concentration (ppb)",
            main = "monthly mean o3")

smoothTrend(mydata, pollutant = "o3", deseason = TRUE, ylab = "concentration (ppb)",
            main = "monthly mean deseasonalised o3")

smoothTrend(mydata, pollutant = "no2", simulate = TRUE, ylab = "concentration (ppb)",
            main = "monthly mean no2 (bootstrap uncertainties)")

## [1] "Taking bootstrap samples. Please wait..."

smoothTrend(mydata, pollutant = "no2", deseason = TRUE, simulate =TRUE,
            ylab = "concentration (ppb)",
            main = "monthly mean deseasonalised no2 (bootstrap uncertainties)")

## [1] "Taking bootstrap samples. Please wait..."
```

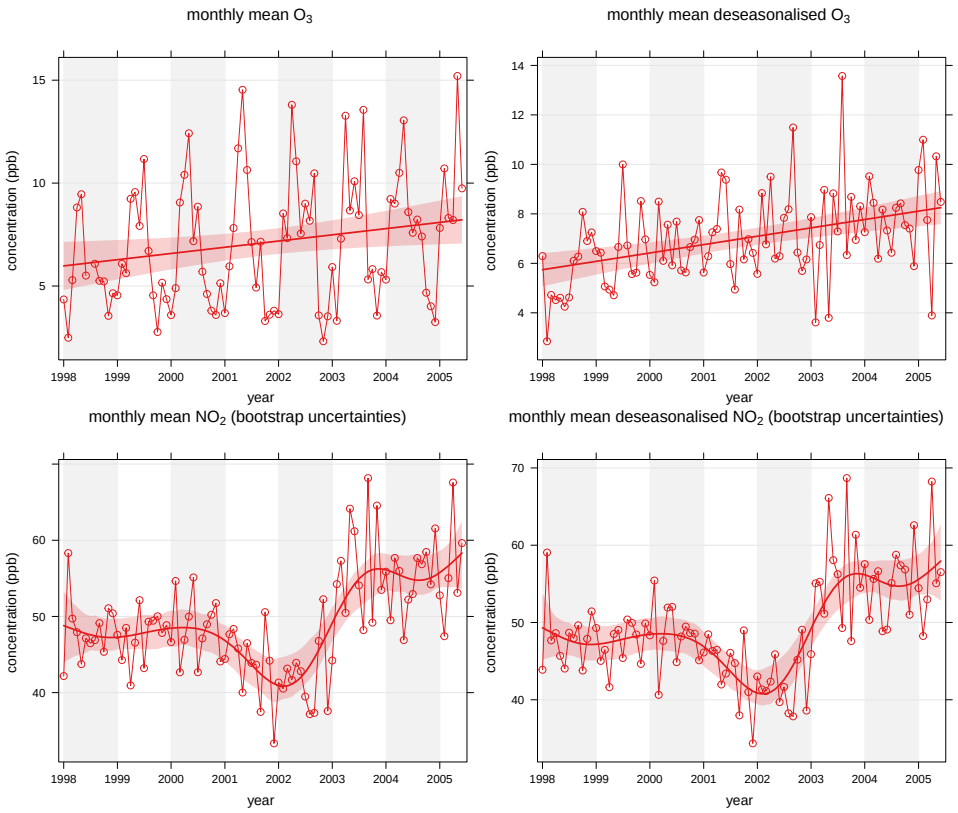


FIGURE 20.1 Examples of the smoothTrend function applied to Marylebone Road

```
smoothTrend(mydata, pollutant = "o3", deseason = TRUE,
            type = "wd")
```

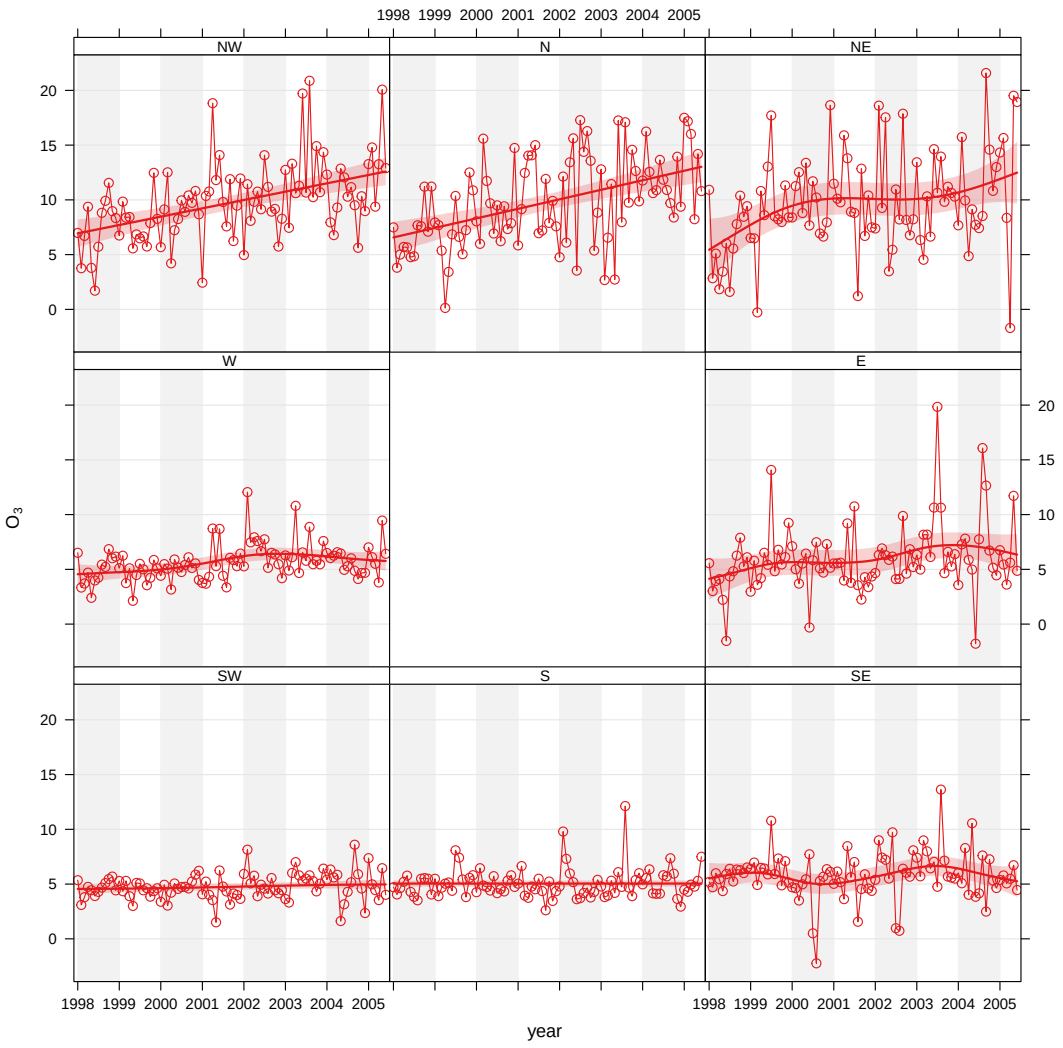


FIGURE 20.2 Trends in O_3 using the `smoothTrend` function applied to Marylebone Road. The shading shows the estimated 95 % confidence intervals. This plot can usefully be compared with Figure 19.2.

```
smoothTrend(mydata, pollutant = c("no2", "pm10", "o3"), type = c("wd", "season"),
            date.breaks = 3, lty = 0)
```

FIGURE 20.3 The smoothTrend function applied to three pollutants, split by wind sector and season.

21 The timeVariation function

21.1 Purpose

see also linearRelation

In air pollution, the variation of a pollutant by time of day and day of week can reveal useful information concerning the likely sources. For example, road vehicle emissions tend to follow very regular patterns both on a daily and weekly basis. By contrast some industrial emissions or pollutants from natural sources (e.g. sea salt aerosol) may well have very different patterns.

The timeVariation function produces four plots: day of the week variation, mean hour of day variation and a combined hour of day – day of week plot and a monthly plot. Also shown on the plots is the 95 % confidence interval in the mean. These uncertainty limits can be helpful when trying to determine whether one candidate source is different from another. The uncertainty intervals are calculated through bootstrap re-sampling, which will provide better estimates than the application of assumptions based on normality, particularly when there are few data available. The function can consider one or two input variables. In addition, there is the option of 'normalising' concentrations (or other quantities). Normalising is very useful for comparing the patterns of two different pollutants, which often cover very different ranges in concentration. Normalising is achieved by dividing the concentration of a pollutant by its mean value. Note also that any other variables besides pollutant concentrations can be considered e.g. meteorological or traffic data.

There is also an option difference which is very useful for considering the difference in two time series and how they vary over different temporal resolutions. Again, bootstrap re-sampling methods are used to estimate the uncertainty of the difference in two means.

Averaging wind direction

Care has been taken to ensure that wind direction (wd) is vector-averaged. Less obvious though is the uncertainty in wind direction. A pragmatic approach has been adopted here that considers how wind direction changes. For example, consider the following wind directions: 10, 10, 10, 180, 180, 180°. The standard deviation of these numbers is 93°. However, what actually occurs is the wind direction is constant at 10° then switches to 180°. In terms of changes there is a sequence of numbers: 0, 0, 170, 0, 0 with a standard deviation of 76°. We use the latter method as a basis of calculating the 95% confidence intervals in the mean.

There are also problems with simple averaging—for example, what is the average of 20 and 200°. It can't be known. In some situations where the wind direction is bi-modal with differences around 180°, the mean can be 'unstable'. For example, wind that is funnelled along a valley forcing it to be either easterly or westerly. Consider for example the mean of 0° and 179° (89.5°), but a small change in wind direction to 181° gives a mean of 270.5°. Some care should be exercised therefore when averaging wind direction. It is always a good idea to use the windRose function with type set to 'month' or 'hour'.

The timeVariation function is probably one of the most useful functions that can be used for the analysis of air pollution. Here are a few uses/advantages:

- Variations in time are one of the most useful ways of characterising air pollution for

a very wide range of pollutants including local urban pollutants and tropospheric background concentrations of ozone and the like.

- The function works well in conjunction with other functions such as `polarPlot`, where the latter may identify conditions of interest (say a wind speed/direction range). By sub-setting for those conditions in `timeVariation` the temporal characteristics of a particular source could be characterised and perhaps contrasted with another subset of conditions.
- The function can be used to compare a wide range of variables, if available. Suggestions include meteorological e.g. boundary layer height and traffic flows.
- The function can be used for comparing pollutants over different sites. See §(31.7) for examples of how to do this.
- The function can be used to compare one part of a time series with another. This is often a very powerful thing to do, particularly if concentrations are normalised. For example, there is often interest in knowing how diurnal/weekday/seasonal patterns vary with time. If a pollutant showed signs of an increase in recent years, then splitting the data set and comparing each part together can provide information on what is driving the change. Is there, for example, evidence that morning rush hour concentrations have become more important, or Sundays have become relatively more important? An example is given below using the `splitByDate` function.
- `timeVariation` can be used to consider the differences between two time series, which will have multiple benefits. For example, for model evaluation it can be very revealing to consider the difference between observations and modelled values over different time scales. Considering such differences can help reveal the character and some of the reasons for why a model departs from reality.

21.2 Options available

The `timeVariation` function has the following options:

`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"`.

21.3 Example of use

We apply the `timeVariation` function to PM_{10} concentrations and take the opportunity to filter the data to maximise the signal from the road. The `polarPlot` function described in (§15) is very useful in this respect in highlighting the conditions under which different sources have their greatest impact. A subset of data is used filtering for wind speeds > 3 m s^{-1} and wind directions from 100–270 degrees. The code used is:

The results are shown in Figure 21.1. The plot shown at the top-left shows the diurnal variation of concentrations for all days. It shows for example that PM_{10} concentrations tend to peak around 9 am. The shading shows the 95 % confidence intervals of the mean. The plot at the top-right shows how PM_{10} concentrations vary by day of

```
timeVariation(subset(mydata, ws > 3 & wd > 100 & wd < 270),
              pollutant = "pm10", ylab = "pm10 (ug/m3)")
```

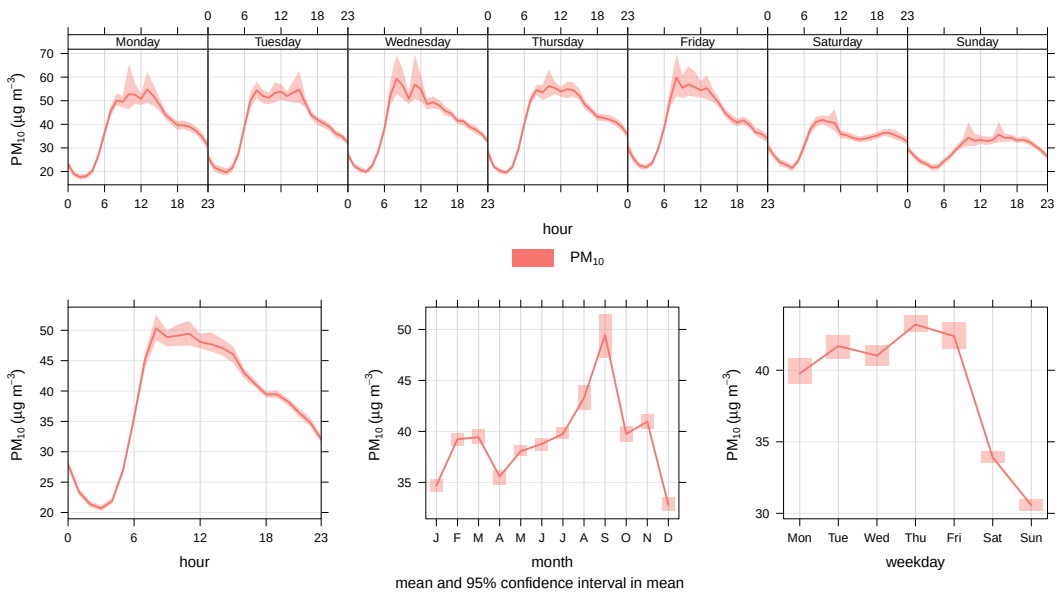


FIGURE 21.1 Example plot using the `timeVariation` function to plot PM_{10} concentrations at Marylebone Road.

the week. Here there is strong evidence that PM_{10} is much lower at the weekends and that there is a significant difference compared with weekdays. It also shows that concentrations tend to increase during the weekdays. Finally, the plot at the bottom shows both sets of information together to provide an overview of how concentrations vary.

timeVariation is also very useful for other variables such as traffic and meteorological data

Note that the plot need not just consider pollutant concentrations. Other useful variables (if available) are meteorological and traffic flow or speed data. Often, the combination of several sets of data can be very revealing.

The `subset` function is extremely useful in this respect. For example, if it were believed that a source had an effect under specific conditions; they can be isolated with the `subset` function. It is also useful if it is suspected that two or more sources are important that they can be isolated to some degree and compared. This is where the uncertainty intervals help — they provide an indication whether the behaviour of one source differs significantly from another.

Figure 21.2 shows the function applied to concentrations of NO_x, CO, NO_2 and O_3 concentrations. In this case the concentrations have been normalised. The plot clearly shows the markedly different temporal trends in concentration. For CO, there is a very pronounced increase in concentrations during the peak pm rush hour. The other important difference is on Sundays when CO concentrations are relatively much higher than NO_x. This is because flows of cars (mostly petrol) do not change that much by day of the week, but flows of vans and HGVs (diesel vehicles) are much less on Sundays. Note, however, that the monthly trend is very similar in each case — which indicates very similar source origins. Taken together, the plots highlight that traffic emissions dominate this site for CO and NO_x, but there are important difference in how these emissions vary by hour of day and day of week.

Also shown in the very different behaviour of O_3. Because O_3 reacts with NO, concentrations of NO_x and O_3 tend to be anti-correlated. Note also the clear peak in O_3

```
timeVariation(mydata, pollutant = c("nox", "co", "no2", "o3"), normalise = TRUE)
```

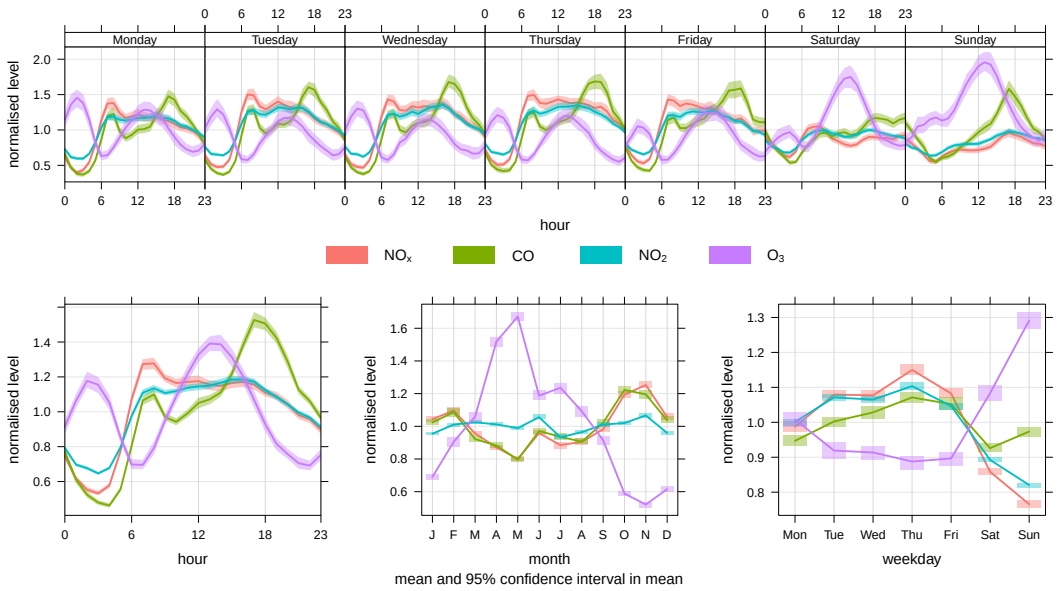


FIGURE 21.2 Example plot using the `timeVariation` function to plot NO_x, CO, NO_2 and O_3 concentrations at Marylebone Road. In this plot, the concentrations are normalised.

in April/May, which is due to higher northern hemispheric background concentrations in the spring. Even at a busy roadside site in central London this influence is clear to see.

Another example is splitting the data set by time. We use the `splitByDate` function to divide up the data into dates before January 2003 and after January 2003. This time the option `difference` is used to highlight how NO_2 concentrations have changed over these two periods. The results are shown in Figure 21.3. There is some indication in this plot that data after 2003 seem to show more of a double peak in the diurnal plots; particularly in the morning rush hour. Also, the difference line does more clearly highlight a more substantial change over weekdays and weekends. Given that cars are approximately constant at this site each day, the change may indicate a change in vehicle emissions from other vehicle types. Given that it is known that primary NO_2 emissions are known to have increased sharply from the beginning of 2003 onwards, this perhaps provides clues as to the principal cause.

In the next example it is shown how to compare one subset of data of interest with another. Again, there can be many reasons for wanting to do this and perhaps the data set at Marylebone Road is not the most interesting to consider. Nevertheless, the code below shows how to approach such a problem. The scenario would be that one is interested in a specific set of conditions and it would be useful to compare that set, with another set. A good example would be from an analysis using the `polarPlot` function where a 'feature' of interest has been identified—maybe an indication of a different source. But does this potentially different source behave in a different way in terms of temporal variation? If it does, then maybe that provides evidence to support that it is a different source. In a wider context, this approach could be used in many different ways depending on available data. A good example is the analysis of model output where many diagnostic meteorological data are available. This is an area that will be developed.

The approach here is to first make a new variable called 'feature' and fill it with the

```
## split data into two periods (see Utlities section for more details)
mydata <- splitByDate(mydata, dates= "1/1/2003",
                      labels = c("before Jan. 2003", "After Jan. 2003"))

timeVariation(mydata, pollutant = "no2", group = "split.by", difference = TRUE)
```

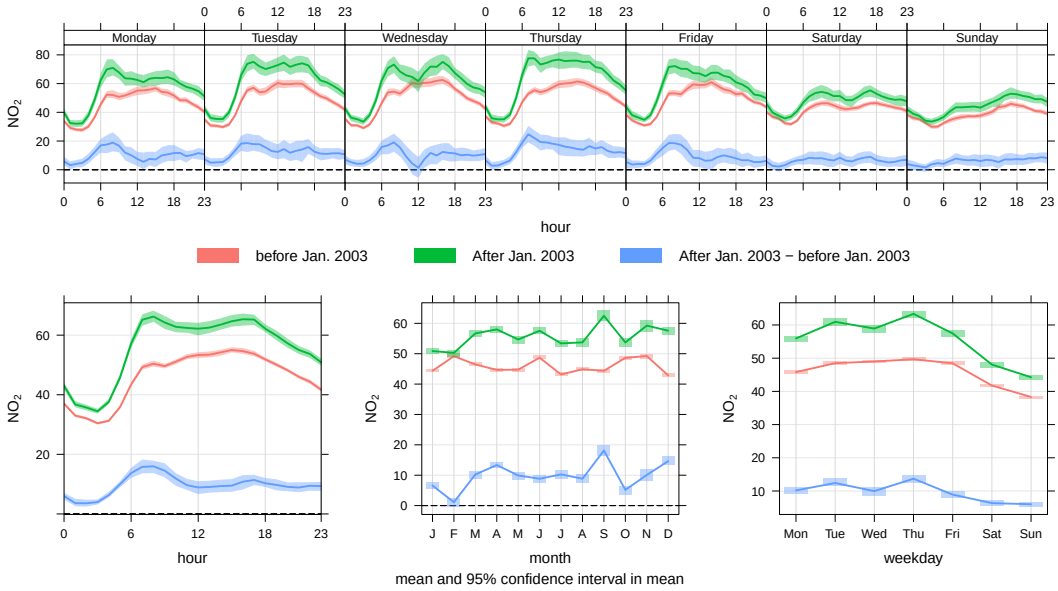


FIGURE 21.3 Example plot using the `timeVariation` function to plot NO_2 concentrations at Marylebone Road. In this plot, the concentrations are shown before and after January 2003.

value 'other'. A subset of data is defined and the associated locations in the data frame identified. The subset of data is then used to update the 'feature' field with a new description. This approach could be extended to some quite complex situations.

There are a couple of things to note in Figure 21.2. There seems to be evidence that for easterly winds > 4 m s^{-1} that concentrations of SO_2 are lower at night. Also, there is some evidence that concentrations for these conditions are also lower at weekends. This *might* reflect that SO_2 concentrations for these conditions tend to be dominated by tall stack emissions that have different activities to road transport sources. This technique will be returned to with different data sets in future.

By default `timeVariation` shows the mean variation in different temporal componnets and the 95% confidence interval in the mean. However, it is also possible to show how the data are distributed by using a different option for `statistic`. When `statistic = "median"` the median line is shown together with the 25/75th and 5/95th quantile values. Users can control the quantile values shown be setting the `conf.int`. For example, `conf.int = c(0.25, 0.99)` will show the median, 25/75th and 1/99th quantile values. The `statistic = "median"` option is therefore very useful for showing how the data are distributed — somewhat similar to a box and whisker plot. Note that it is expected that only one pollutant should be shown when `statistic = "median"` is used due to potential over-plotting; although the function will display several species of required. An example is shown in Figure 21.5 for PM_{10} concentrations.

```
## make a field called "site" and fill: make all values = "other"
mydata$feature <- "other"

## now find which indexes correspond to easterly conditions > 4m/s ws
id <- which(with(mydata, ws > 4 & wd > 0 & wd <= 180 ))

## use the ids to update the site column
## there are now two values in site: "other" and "easterly"
mydata$feature[id] <- "easterly"
timeVariation(mydata, pollutant ="so2", group = "feature",  ylab = "so2 (ppb)",
              difference = TRUE)
```

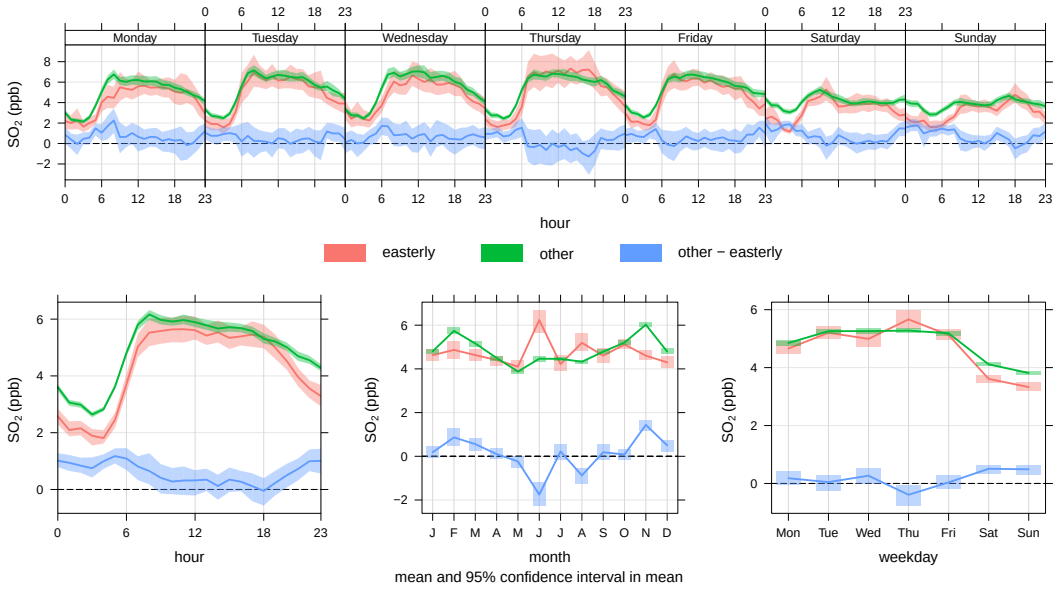


FIGURE 21.4 Example plot using the `timeVariation` function to plot SO_2 concentrations at Marylebone Road. In this plot, the concentrations are shown for a subset of easterly conditions and everything else. Note that the uncertainty in the mean values for easterly winds is greater than 'other'. This is mostly because the sample size is much lower for 'easterly' compared with 'other'.

```
timeVariation(mydata, pollutant = "pm10", statistic = "median",
              col = "firebrick")
```

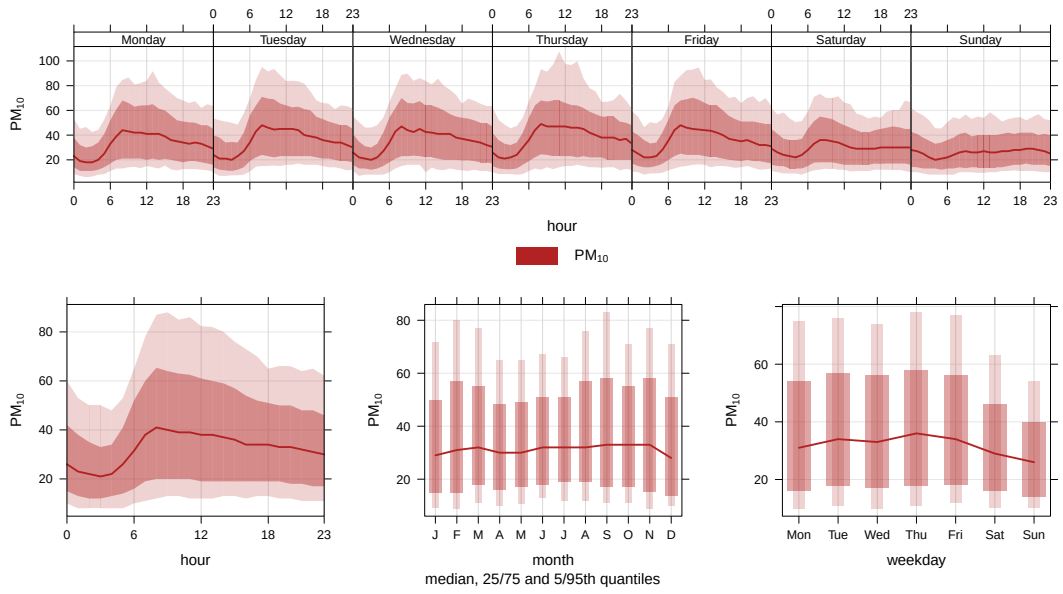


FIGURE 21.5 Example plot using the `timeVariation` function to show the variation in the median, 25/75th and 5/95th quantile values for PM_{10}. The shading shows the extent to the 25/75th and 5/95th quantiles.

21.4 Output

The timeVariation function produces several outputs that can be used for further analysis or plotting. It is necessary to read the output into a variable for further processing. The code below shows the different objects that are returned and the code shows how to access them.

```
myOutput <- timeVariation(mydata, pollutant = "so2")
## show the first part of the day/hour variation
## note that value = mean, and Upper/Lower the 95% confid. intervals
head(myOutput$data$day.hour)
```

```
##   variable     wkday hour                         default     Mean    Lower
## 1      so2    Monday    0 01 January 1998 to 23 June 2005 2.926235 2.619467
## 2      so2   Tuesday    0 01 January 1998 to 23 June 2005 3.213496 2.919813
## 3      so2 Wednesday    0 01 January 1998 to 23 June 2005 3.350445 3.128826
## 4      so2  Thursday    0 01 January 1998 to 23 June 2005 3.222399 2.993683
## 5      so2    Friday    0 01 January 1998 to 23 June 2005 3.641693 3.324750
## 6      so2  Saturday    0 01 January 1998 to 23 June 2005 4.252947 3.972240
##      Upper   ci
## 1 3.234655 0.95
## 2 3.523941 0.95
## 3 3.566802 0.95
## 4 3.484657 0.95
## 5 3.872712 0.95
## 6 4.568876 0.95
```

```
## can make a new data frame of this data e.g.
day.hour <- myOutput$data$day.hour
head(day.hour)
```

```
##   variable     wkday hour                         default     Mean    Lower
## 1      so2    Monday    0 01 January 1998 to 23 June 2005 2.926235 2.619467
## 2      so2   Tuesday    0 01 January 1998 to 23 June 2005 3.213496 2.919813
## 3      so2 Wednesday    0 01 January 1998 to 23 June 2005 3.350445 3.128826
## 4      so2  Thursday    0 01 January 1998 to 23 June 2005 3.222399 2.993683
## 5      so2    Friday    0 01 January 1998 to 23 June 2005 3.641693 3.324750
## 6      so2  Saturday    0 01 January 1998 to 23 June 2005 4.252947 3.972240
##      Upper   ci
## 1 3.234655 0.95
## 2 3.523941 0.95
## 3 3.566802 0.95
## 4 3.484657 0.95
## 5 3.872712 0.95
## 6 4.568876 0.95
```

All the numerical results are given by:

```
myOutput$data$day.hour ## are the weekday and hour results
myOutput$data$hour ## are the diurnal results
myOutput$data$day ## are the weekday results
myOutput$data$month ## are the monthly results
```

It is also possible to plot the individual plots that make up the (four) plots produced by timeVariation:

```
## just the diurnal variation
plot(myOutput, subset = "hour")
## day and hour
plot(myOutput, subset = "day.hour")
## weekday variation
plot(myOutput, subset = "day")
## monthly variation
plot(myOutput, subset = "month")
```

22 The scatterPlot function

22.1 Purpose

Scatter plots are extremely useful and a very commonly used analysis technique for considering how variables relate to one another. R does of course have many capabilities for plotting data in this way. However, it can be tricky to add linear relationships, or split scatter plots by levels of other variables etc. The purpose of the `scatterPlot` function is to make it straightforward to consider how variables are related to one another in a way consistent with other **openair** functions. We have added several capabilities that can be used just by setting different options, some of which are shown below.

- A smooth fit is automatically added to help reveal the underlying relationship between two variables together with the estimated 95% confidence intervals of the fit. This is in general an extremely useful thing to do because it helps to show the (possibly) non-linear relationship between variables in a very robust way — or indeed whether the relationship is linear.
- It is easy to add a linear regression line. The resulting equation is shown on the plot together with the R^2 value.
- For large data sets there is the possibility to 'bin' the data using hexagonal binning or kernel density estimates. This approach is very useful when there is considerable over-plotting.
- It is easy to show how two variables are related to one another dependent on levels of a third variable. This capability is very useful for exploring how different variables depend on one another and can help reveal the underlying important relationships.
- A plot of two variables can be colour-coded by a continuous colour scale of a third variable.
- It can handle date/time x-axis formats to provide an alternative way of showing time series, which again can be colour-coded by a third variable.

The `scatterPlot` function isn't really specific to atmospheric sciences, in the same way as other plots. It is more a function for convenience, written in a style that is consistent with other **openair** functions. Nevertheless, along with the `timePlot` function they do form an important part of **openair** because of the usefulness of understanding show variables relate to one another. Furthermore, there are many options to make it easy to explore data in an interactive way without worrying about processing data or formatting plots.

22.2 Options available

`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.

`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.

`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.

`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"`.

`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

```
data2003 <- selectByDate(mydata, year = 2003)
scatterPlot(data2003, x = "nox", y = "no2")
```

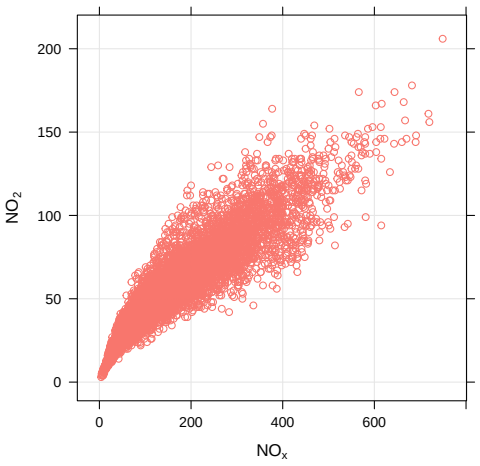


FIGURE 22.1 Scatter plot of hourly NO_x vs. NO_2 at Marylebone Road for 2003.

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.

22.3 Example of use

We provide a few examples of use and as usual, users are directed towards the help pages (type ?scatterPlot) for more extensive examples.

First we select a subset of data (2003) using the **openair** `selectByDate` function and plot NO_x vs. NO_2 (Figure 22.1).

Often with several years of data, points are over-plotted and it can be very difficult to see what the underlying relationship looks like. One very effective method to use in these situations is to 'bin' the data and to colour the intervals by the number of counts of occurrences in each bin. There are various ways of doing this, but 'hexagonal binning' is particularly effective because of the way hexagons can be placed next to one another.[14] To use hexagonal binning it will be necessary to install the `hexbin` package:

```
install.packages("hexbin")
```

Now it should be possible to make the plot by setting the method option to `method = "hexbin"`, as shown in Figure 22.2. The benefit of hexagonal binning is that it works equally well with enormous data sets e.g. several million records. In this case Figure 22.2 provides a clearer indication of the relationship between NO_x and NO_2 than Figure 22.1 because it reveals where most of the points lie, which is not apparent from Figure 22.1. Note that 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 for example (the default), `trans = function(x) log(x)` and `inv = function(x) exp(x)`. For a square root transform use `trans = sqrt` and `inv =`

[14] In fact it is not possible to have a shape with more than 6 sides that can be used to forma a lattice without gaps.

```
scatterPlot(data2003, x = "nox", y = "no2", method = "hexbin", col= "jet")
```

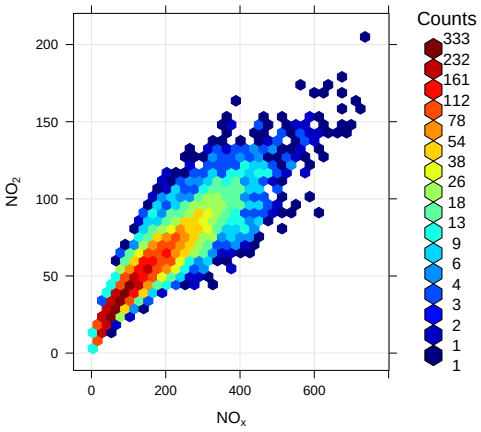


FIGURE 22.2 Scatter plot of hourly NO_x vs. NO_2 at Marylebone Road using hexagonal binning. The number of occurrences in each bin is colour-coded (not on a linear scale). It is now possible to see where most of the data lie and a better indication of the relationship between NO_x and NO_2 is revealed.

`function(x) x2`. To not apply any transformation `trans = NULL` and `inv = NULL` should be used.

Note that when `method = "hexbin"` there are various options that are useful e.g. a border around each bin and the number of bins. For example, to place a grey border around each bin and set the bin size try:

```
scatterPlot(mydata, x = "nox", y = "no2", method = "hexbin", col = "jet",
            border = "grey", xbin = 15)
```

The hexagonal binning and other binning methods are useful but often the choice of bin size is somewhat arbitrary. Another useful approach is to use a kernel density estimate to show where most points lie. This is possible in `scatterPlot` with the `method = "density"` option. Such a plot is shown in Figure 22.3.

Sometimes it is useful to consider how the relationship between two variables varies by levels of a third. In **openair** this approach is possible by setting the option `type`. When `type` is another numeric variables, four plots are produced for different quantiles of that variable. We illustrate this point by considering how the relationship between NO_x and NO_2 varies with different levels of O_3. We also take the opportunity to not plot the smooth line, but plot a linear fit instead and force the layout to be a 2 by 2 grid.

Finally, we show how to plot a continuous colour scale for a third *numeric* variable setting the value of `z` to the third variable. Figure 22.5 shows again the relationship between NO_x and NO_2 but this time colour-coded by the concentration of O_3. We also take the opportunity to split the data into seasons and weekday/weekend by setting `type = c("season", "weekend")`. There is an enormous amount of information that can be gained from plots such as this. Differences between weekdays and the weekend can highlight changes in emission sources, splitting by seasons can show seasonal influences in meteorology and background O_3 and colouring the data by the concentration of O_3 helps to show how O_3 concentrations affect NO_2 concentrations. For example, consider the summertime-weekday panel where it clearly shows that the higher NO_2 concentrations are associated with high O_3 concentrations. Indeed

```
scatterPlot(selectByDate(mydata, year = 2003), x = "nox", y = "no2",
            method = "density", col = "jet")

## (loaded the KernSmooth namespace)
```

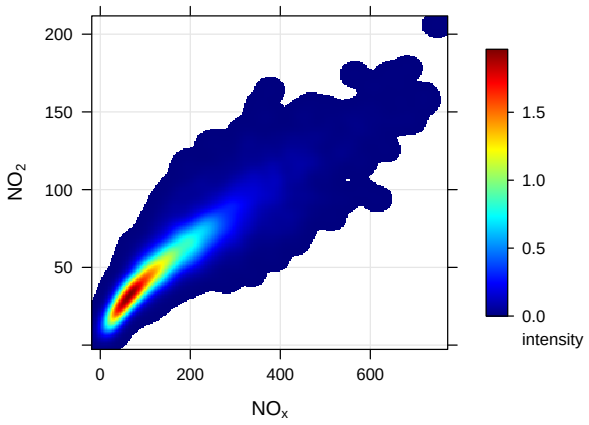


FIGURE 22.3 Scatter plot of hourly NO_x vs. NO_2 at Marylebone Road using a kernel density estimate to show where most of the points lie. The 'intensity' is a measure of how many points lie in a unit area of NO_x and NO_2 concentration.

```
scatterPlot(data2003, x = "nox", y = "no2", type = "o3", smooth = FALSE,
            linear = TRUE, layout = c(2, 2))
```

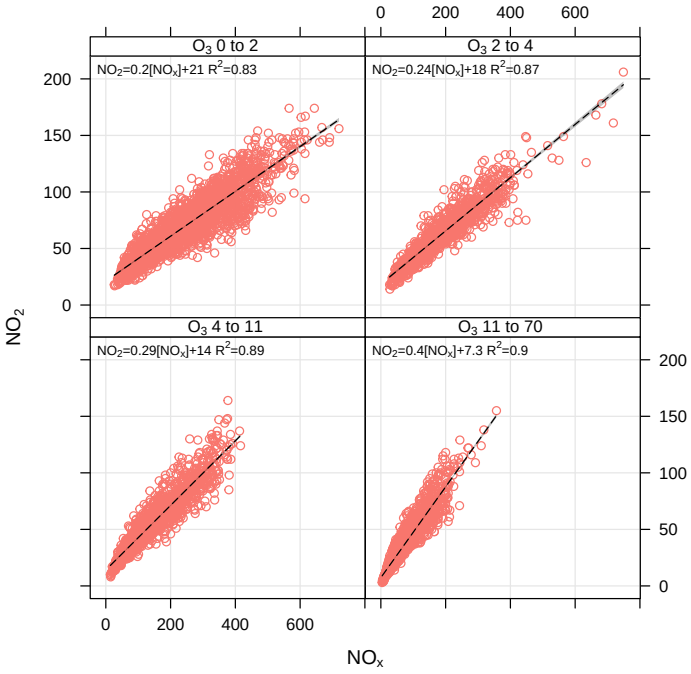


FIGURE 22.4 Scatter plot of hourly NO_x vs. NO_2 at Marylebone Road by different levels of O_3.

```
scatterPlot(data2003, x = "nox", y = "no2", z = "o3", type = c("season", "weekend"),
            limits = c(0, 30))
```

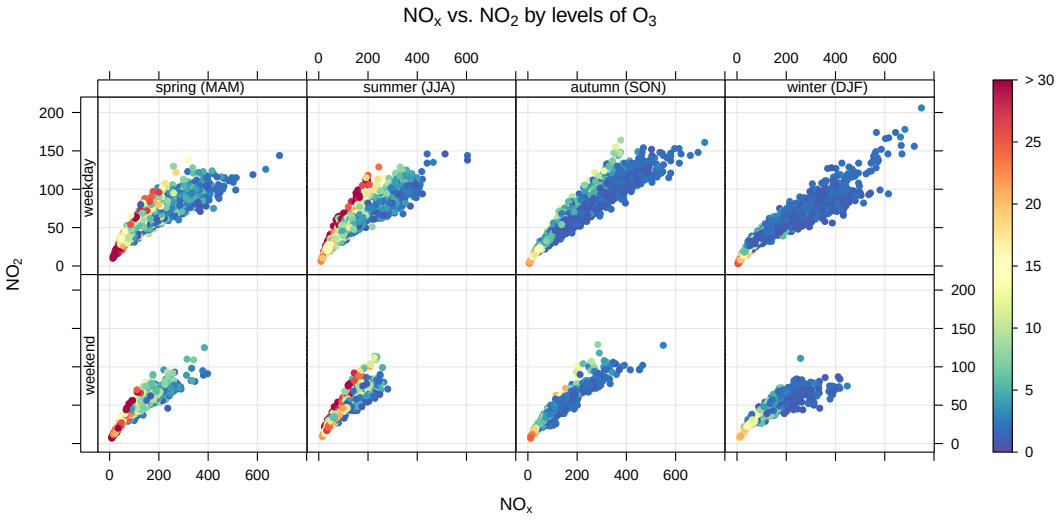


FIGURE 22.5 Scatter plot of hourly NO_x vs. NO_2 at Marylebone Road by different levels of O_3 split by season and weekday-weekend.

```
scatterPlot(selectByDate(data2003, month = 8), x = "date", y = "so2",
            z = "wd")
```

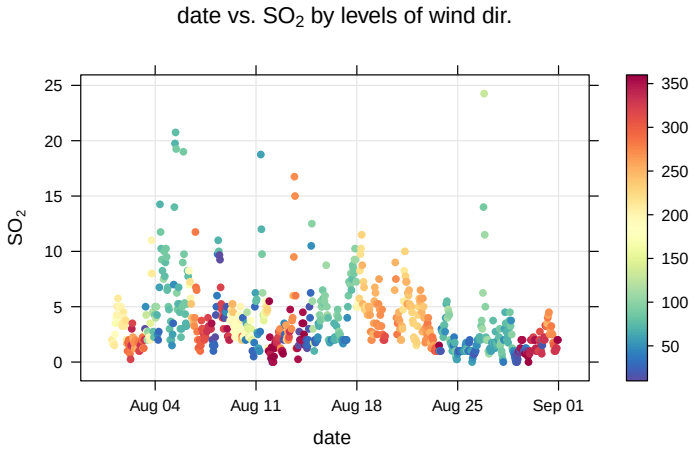


FIGURE 22.6 Scatter plot of date vs. SO_2 at Marylebone Road by different levels of wind direction for August 2003.

there are some hours where NO_2 is >100 ppb at quite low concentrations of NO_x (≈200 ppb). It would also be interesting instead of using O_3 concentrations from Marylebone Road to use O_3 from a background site.

Figure 22.5 was very easily produced but contains a huge amount of useful information showing the relationship between NO_x and NO_2 dependent upon the concentration of O_3, the season and the day of the week. There are of course numerous other plots that are equally easily produced.

Figure 22.6 shows that `scatterPlot` can also handles dates on the x-axis; in this case shown for SO_2 concentrations coloured by wind direction for August 2003.

23 The linearRelation function

see also
timeVariation
calcFno2

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

The relationships between pollutants can yield some very useful information about source emissions and how they change. A scatter plot 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`.

Consider the relationship between NO_x and NO_2. It is best to think of the relationship as:

$$y = m.x + c \tag{8}$$

i.e.

$$NO_2 = m.NO_x + c \tag{9}$$

In which case x corresponds to NO_x and y corresponds to NO_2. The plots show the gradient, m in what ever units the original data were in. For comparison with emission inventories it makes sense to have all the units expressed as mass. By contrast, oxidant slopes are best calculated in volume units e.g. ppb.

`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 NO_2 + O_3), 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.

23.1 Options available

`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. “monthly” will plot a monthly time series and “weekly” a weekly time series of the relationship between `x` and `y`. ”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. If set too high it may not be possible to fit a smooth line and warnings will be issues - but the plot still produced.

`ylim` y-axis limits, specified by the user.

`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` Predefined colour scheme, currently only enabled for "greyscale".

`span` span for `loess` fit. Controls the fit line: lower values produce a more "wiggly" fit.

`...` 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`.

23.2 Example of use

Some examples of the `linearRelation` function are given in this section. The first example considers the ratio of SO_2/NO_x, which is plotted in Figure 23.1.

Figure 23.1 shows the relationship between NO_x and SO_2. Early in the series (pre-1999) the ratio of SO_2/NO_x was relatively high (about 3.5 in volume units). However, from 1999 onwards the relationship has been relatively constant. One (probable) explanation for the higher ratio at the beginning of the series is due to a higher fuel sulphur content of petrol and diesel. There are many other examples shown in the package itself, type `?linearRelation` to see them.

One of the useful applications of this function is to consider the 'oxidant' (sum of NO_2 and O_3) slope where there are measurements of NO_x, NO_2 and O_3 at a site. At roadside sites the oxidant slope provides a good indication of the likely ratio of NO_2/NO_x in vehicle exhausts. Because there are few sites that measure O_3 at the roadside, the `calcFno2` function provides an alternative method of estimation. Figure 23.2 shows how the oxidant slope (an estimate of f-NO_2) varies by day of the week and hour of the day.

```
linearRelation(mydata, x = "nox", y = "so2")
```

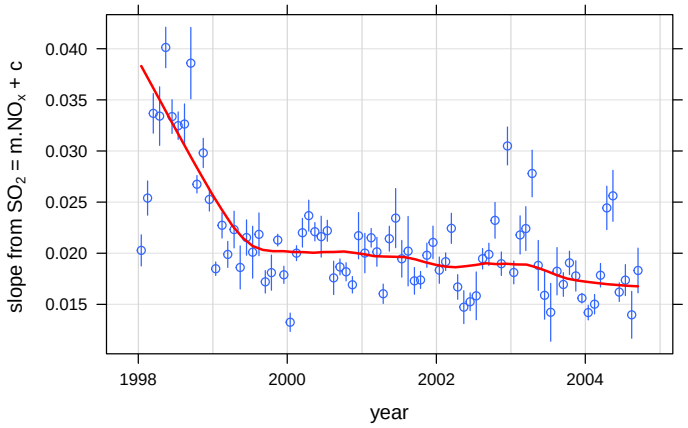


FIGURE 23.1 Relationship between NO_x and SO_2 using the `linearRelation` function. Note that the units of both pollutants are in ppb. The uncertainty in the slope of the hourly relationship between SO_2 and NO_x on a monthly basis is shown at 95 % confidence intervals. The smooth line and shaded area show the general trend using a loess smooth.

```
linearRelation(mydata, x = "nox", y = "ox", period = "day.hour")
```

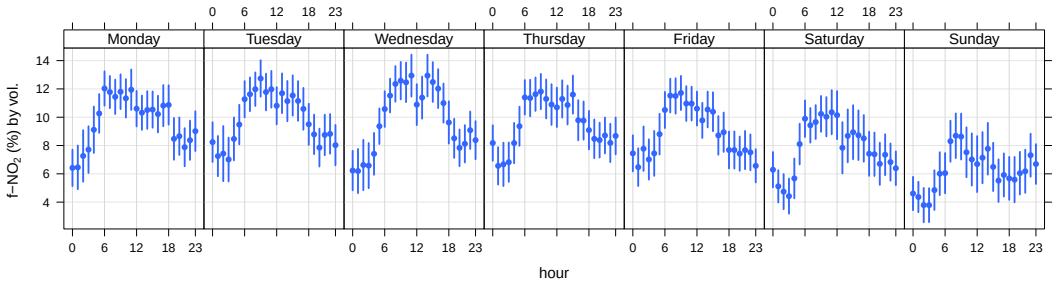


FIGURE 23.2 Oxidant slope by day of the week and hour of the day.

24 The `trendLevel` function

24.1 Purpose

The `trendLevel` function provides a way of rapidly showing a large amount of data in a condensed way. It is particularly useful for plotting the level of a value against two categorical variables. These categorical variables can pre-exist in a data set or be made on the fly using **openair**. By default it will show the mean value of a variable against two categorical variables but can also consider a wider range of statistics e.g. the maximum, frequency, or indeed a user-defined function. The function is much more flexible than this by showing temporal data and can plot 'heat maps' in many flexible ways. Both continuous colour scales and user-defined categorical scales can be used.

The `trendLevel` function shows how the value of a variable varies according to intervals of two other variables. The *x* and *y* variables can be categorical (factor or character) or numeric. The third variable (*z*) must be numeric and is coloured according to its value. Despite being called `trendLevel` the function is flexible enough to consider a wide range of plotting variables.

If the *x* and *y* variables are not categorical they are made so by splitting the data into quantiles (using `cutData`).

24.2 Options available

`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.

24.3 Example of use

The standard output from `trendLevel`is shown in Figure 24.1, which shows the variation in NO_x concentrations by year and hour of the day.

Figure 24.3 indicates that the highest NO_x concentrations most strongly associate with wind sectors about 200 degrees, appear to be decreasing over the years, but do not appear to associate with an SO_2 rich NO_x source. Using `type = "so2"` would have conditioned by absolute SO_2 concentration. As both a moderate contribution from an SO_2 rich source and a high contribution from an SO_2 poor source could generate similar SO_2 concentrations, such conditioning can sometimes blur interpretations. The

```
trendLevel(mydata, pollutant = "nox")
```

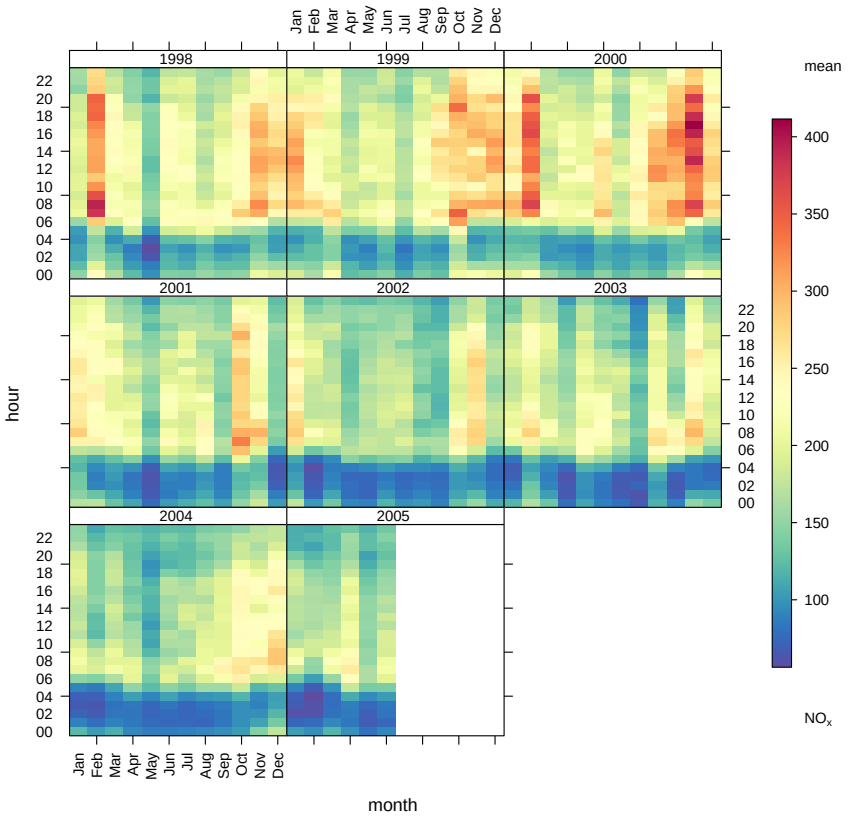


FIGURE 24.1 Standard `trendLevel` output.

use of this type of 'over pollutant' ratio reduces this blurring by focusing conditioning on cases when NO_x concentrations (be they high or low) associate with relatively high or low SO_2 concentrations.

The plot can be used in much more flexible ways. Here are some examples (not plotted):

A plot of mean O_3 concentration shown by season and by daylight/nighttime hours.

```
trendLevel(mydata, x = "season", y = "daylight", pollutant = "o3")
```

Or by season and hour of the day:

```
trendLevel(mydata, x = "season", y = "hour", pollutant = "o3",
           cols = "increment")
```

How about NO_x versus NO_2 coloured by the concentration of O_3? `scatterPlot` could also be used to produce such a plot. However, one interesting difference with using `trendLevel` is that the data are split into quantiles where equal numbers of data exist in each interval. This approach can make it a bit easier to see the underlying relationship between variables. A scatter plot may have too much data to be clear and also outliers (or regions with relatively few data) that make it harder to see what is going on. The plot generated by the command below makes it a bit easier to see that it is the higher quantiles of NO_2 that are associated with higher O_3 concentration (as well as low NO_x and NO_2 concentrations).

```
trendLevel(mydata, pollutant = "nox", y = "wd", border = "white")
```

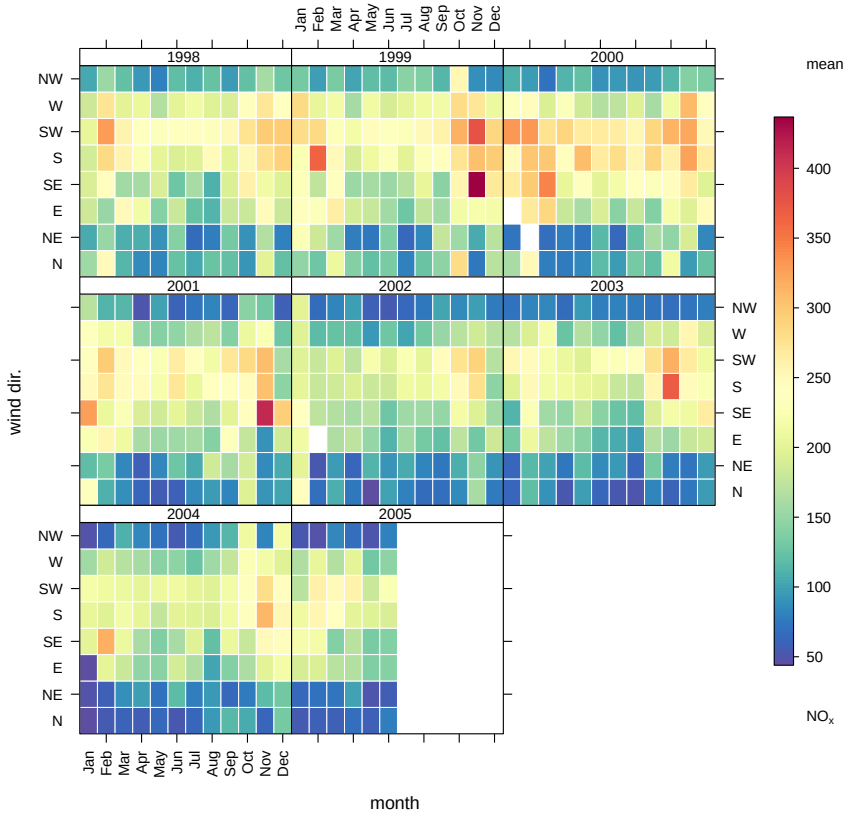


FIGURE 24.2 `trendLevel` output with wind direction as `y` axis. This plot also shows the effect of setting the border colour to white.

```
trendLevel(mydata, x = "nox", y = "no2", pollutant = "o3", border = "white",
           n.levels = 10, statistic = "max", limits = c(0, 50))
```

The plot can also be shown by wind direction sector, this time showing how O_3 varies by weekday, wind direction sector and NO_x quantile.

```
trendLevel(mydata, x = "nox", y = "weekday", pollutant = "o3",
           border = "white", n.levels = 10, statistic = "max",
           limits = c(0, 50), type = "wd")
```

By default `trendLevel` subsamples the plotted `pollutant` data by the supplied `x`, `y` and `type` parameters and in each case calculates the mean. The option `statistic` has always let you apply other statistics. For example, `trendLevel` also calculated the maximum via the option `statistic = "max"`. The user may also use their own statistic function.

As a simple example, consider the above plot which summarises by mean. This tells us about average concentrations. It might also be useful to consider a particular percentile of concentrations. This can be done by defining one's own function as shown in Figure 24.4.

This type of flexibility really opens up the potential of the function as a screening tool for the early stages of data analysis. Increased control of `x`, `y`, `type` and `statistic` allow you to very quick explore your data and develop an understanding of how different parameters interact. Patterns in `trendLevel` plots can also help to direct your

```
## new field: so2/nox ratio
mydata$ratio <- mydata$so2/mydata$nox

## condition by mydata$new
trendLevel(mydata, "nox", x = "year", y = "wd", type = "ratio")
```

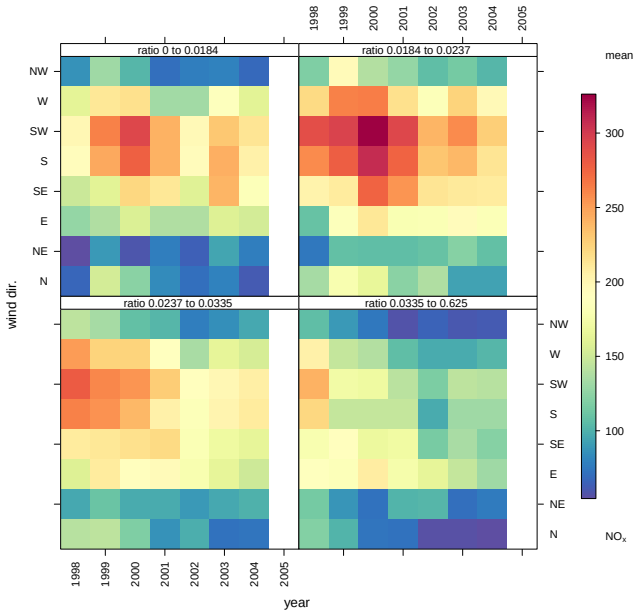


FIGURE 24.3 `trendLevel` output with SO_2: NO_x ratio `type` conditioning.

```
## function to estimate 95th percentile
percentile <- function(x) quantile(x, probs = 0.95, na.rm = TRUE)

## apply to present plot
trendLevel(mydata, "nox", x = "year", y = "wd", type = "ratio",
           statistic = percentile)
```

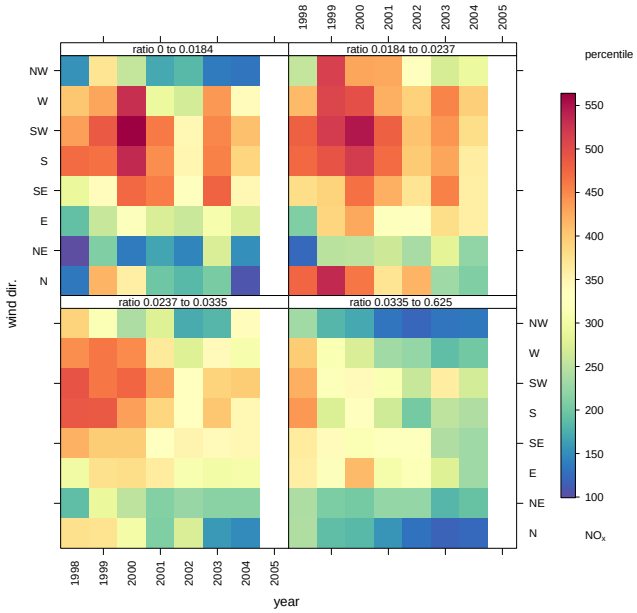


FIGURE 24.4 `trendLevel` using locally defined `statistic`.

```
trendLevel(mydata, pollutant = "no2",
           border = "white",  statistic = "max",
           breaks = c(0, 50, 100, 500),
           labels = c("low", "medium", "high"),
           cols = c("forestgreen", "yellow", "red"))
```

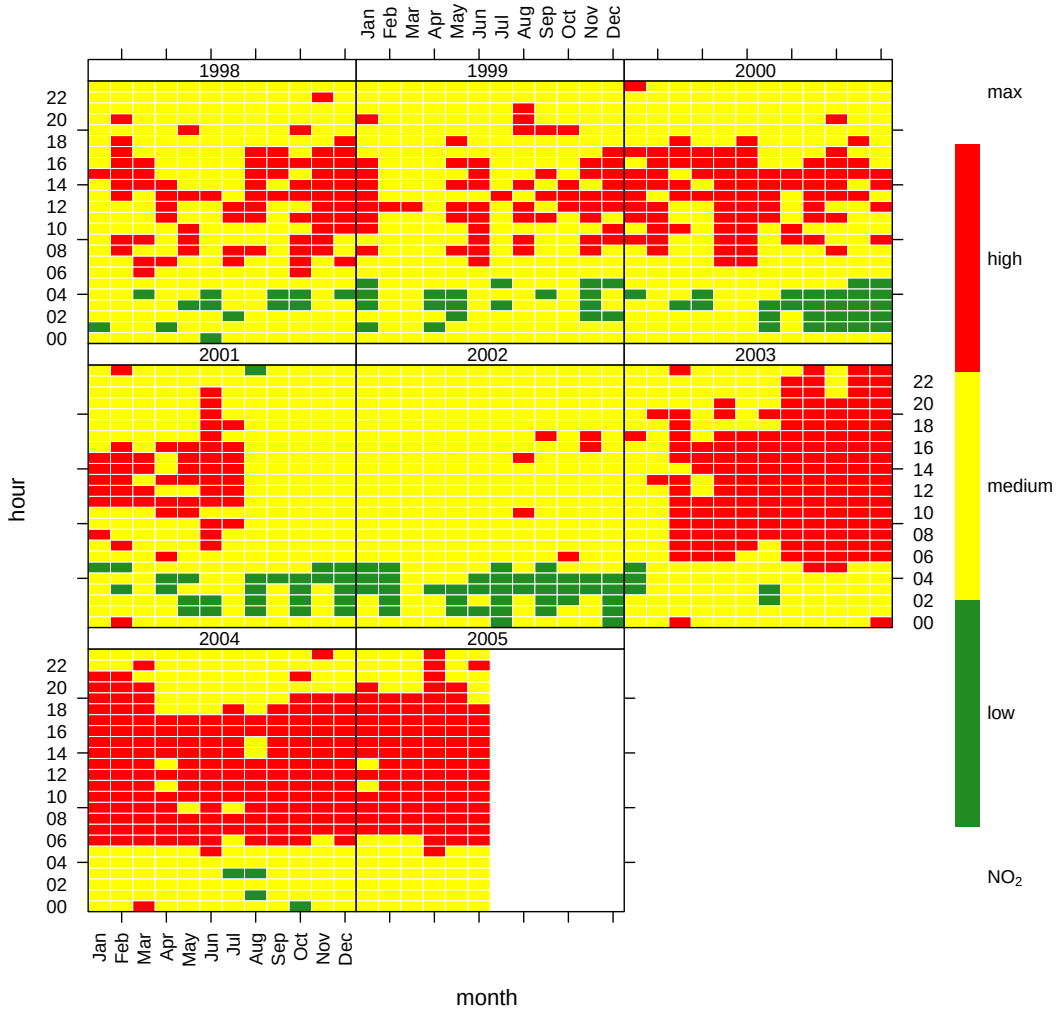


FIGURE 24.5 `trendLevel` plot for maximum NO_2 concentrations using a user-defined discrete colour scale.

openair analysis. For example, possible trends in data conditioned by year would suggest that functions like `smoothTrend` or `TheilSen` could provide further insight. Likewise, `windRose` or `polarPlot` could be useful next steps if wind speed and direct conditioning produces interesting features. However, perhaps most interestingly, novel conditioning or the incorporation of novel parameters in this type of highly flexible function provides a means of developing new data visualisation and analysis methods.

`trendLevel` can also be used with user defined discrete colour scales as shown in Figure 24.5. In this case the default *x* and *y* variables are chosen (month and hour) split by `type` (year).

25 GoogleMapsPlot function

25.1 Purpose

Air pollution is an inherently spatial problem. The spatial aspects can be considered in many different ways and the GoogleMapsPlot function considers simple maps based on the Google Static Maps API (http://code.google.com/apis/maps/documentation/staticmaps/). The initial aim of this function is to make it easy to plot spot location information e.g. a concentration, on a map. Compared with a full-blown GIS the GoogleMapsPlot function is limited, but nevertheless very useful. One of the most useful aspects is that the user only need supply latitude and longitude information for *anywhere in the world* and the map is automatically downloaded from the Google server.

Another benefit of the GoogleMapsPlot function is that it works in the same way as other **openair** functions e.g. allows fro conditioning. Therefore, it is trivial to plot panels of maps split by variables such as 'season'.

25.2 Options available

`mydata` The openair data frame to use to generate the GoogleMapsPlot plot.

`latitude,longitude` The names of the data series in `mydata` giving the latitudes and longitudes, respectively, of measurements. If only one latitude longitude pair are supplied, the function applies a default range to the plot. To override this either set the required range using `xlim` and `ylim` (see below) or the map `zoom` level. (Note: The default is equivalent to `zoom = 15`.)

`type` The type of data conditioning to apply before plotting. The default is will produce a single plot using the entire data. Other type options include "hour" (for hour of the day), "weekday" (for day of the week) and "month" (for month of the year), "year", "season" (string, summer, autumn or winter) and "daylight" (daylight or nighttime hour). But it is also possible to set `type` to the name of another variable in `mydata`, in which case the plotted data will be divided into quantiles based on that data series. See `cutData` for further details.(NOTE: type conditioning currently allows up to two levels of conditioning, e.g., `type = c("weekday", "daylight")`.)

`xlim,ylim` The x-axis and y-axis size ranges. By default these sized on the basis of `latitude` and `longitude`, but can be forced as part of the plot call. (NOTE: This are in-development and should be used with care. The RgoogleMaps argument `size = c(640, 640)` can be use to force map dimensions to square.)

`pollutant` If supplied, the name of a pollutant or variable in `mydata` that is to be evaluated at the each measurement point. Depending on settings, nominally `cols` and `cex`, the evaluation can be by colour, size or both.

`labels` If supplied, either the name of `mydata` column/field containing the labels to be used or a list, containing that field name (as `labels`), and any other label properties, e.g. `cex`, `col`, etc, required for fine-tuning label appearance.

`cols` The colour set to use to colour scaled data. Typically, `cols` is passed to `openColours` for evaluation, but can be forced to one colour using

e.g. col = "red". The special case cols = "greyscale" forces all plot components (the map, the data layer and the plot strip of type conditioning) to greyscale for black and white printing. See ?openColours for more details.

limits — By default, the data colour scale is fitted to the total data range. However, there are circumstances when the user may wish to set different ones. In such cases limits can be set in the form c(lower, upper) to modify the colour range.

cex — The size of data points plotted on maps. By default this NULL or pollutant if supplied. If NULL all points are plotted an equal size. If pollutant or the name of another variable in mydata this is used by scaled using cex.range. If necessary, cex can also be forced, e.g. cex = 1 to make all points the same size.

pch — The plot symbol to be used when plotting data. By default this is a solid circle (pch = 20), but can be any predefined symbol, e.g. pch = 1 is the open circle symbol used in most standard R plots. pch may also be the name of a variable in mydata for local control.

cex.range — The range to rescale cex values to if cex is supplied as a mydata variable name. This is intended to provide sensible data point points regardless of the variable value range but may be require fine-tuning.

xlab,ylab,main — The x-axis, y-axis and main title labels to be added to the plot. All labels are passed via quickText to handle formatting if enabled (auto.text = TRUE). By default GoogleMapsPlot uses latitude and longitude names as xlab and ylab, respectively.

axes — An alternative (short hand) option to add/remove (TRUE/FALSE) all x and y axis annotation and labelling.

map — If supplied, an RgoogleMaps output, to be used as a background map. If NULL (as in default), a map is produced using the RgoogleMaps-package function MapBackground, the supplied latitude and longitude ranges, and any additional RgoogleMaps-package arguments supplied as part of the plot call. (Note: the map object currently used in panel... functions is a modified form of this output, details to be confirmed.)

map.raster — Should the map be plotted as a raster object? The default TRUE uses panel.GoogleMapsRaster to produce the map layer, while the alternative (FALSE) uses panel.GoogleMaps. (NOTE: The raster version is typically much faster but may not be available for all computer systems.)

map.cols — Like cols a colour scale, but, if supplied, used to recolour the map layer before plotting. (NOTE: If set, this will override cols = "greyscale".)

aspect — The aspect ratio of the plot. If NULL (default), this is calculated by the function based on the data and xlim and ylim ranges.

as.table — as.table is a lattice option that controls the order in which multiple panels are displayed. The default (TRUE) produces layouts similar to other openair plot.

`plot.type` The method to use to produce the data layer for the plot. By default (`plot.type = "xy"`), this is an x-y style scatter plot, but can also be other pre-defined options (e.g. "level" for a levelplot) or a user-defined panel of a similar structire to `panel...` functions in `lattice`.

`plot.transparent` Data layer transparency control. When enabled, this forces colours used in the data layer to transparent, and can be a numeric setting the colour density, from invisible (0) to solid (1), or a logical (`TRUE` applying default 0.5). Note: User-defined colours (and some panel defaults when supplying specialist functions using e.g. `plot.type = panel...`) may sometimes supersede this option.

`key` Fine control for the color scale key. By default (`key = NULL`) the key is generated is a colour range exists, but can be forced (`key = TRUE/FALSE`) or controlled at a higher level (via `drawOpenKey`).

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

`key.header,key.footer` Header and footer labels to add to colour key, if drawn. If enabled (`auto.text = TRUE`), these arguments are passed to the scale key (`drawOpenKey`) via `quickText` to handle formatting.

`auto.text` Automatic routine text formatting. `auto.text = TRUE` allows labels (`xlab`, `ylab`, `main`, etc.) to be passed to the plot via `quickText`. `auto.text = FALSE` turns this option off and passes labels to the plot without modification.

`...` Addition options are passed on to `cutData` for `type` handling, `MapBackground` in `RgoogleMaps` for map layer production, and `xyplot` in `lattice` for data layer production.

25.3 Example of usage

To make things a bit more interesting we are going to consider O_3 concentrations across the UK. Hourly O_3 data from 16 sites for 2006 has been placed on a server together with a separate file consisting of the site names and locations. The first thing to do is import the data:

```
load(url("http://www.erg.kcl.ac.uk/downloads/Policy_Reports/AQdata/o3Measurements.RData"))
head(o3Measurements)
```

```
##                  date o3       site
## 1 2006-01-01 00:00:00 NA Aston.Hill
## 2 2006-01-01 01:00:00 74 Aston.Hill
## 3 2006-01-01 02:00:00 72 Aston.Hill
## 4 2006-01-01 03:00:00 72 Aston.Hill
## 5 2006-01-01 04:00:00 70 Aston.Hill
## 6 2006-01-01 05:00:00 66 Aston.Hill
```

```
load(url("http://www.erg.kcl.ac.uk/downloads/Policy_Reports/AQdata/siteDetails.RData"))
head(siteDetails)
```

```
##          site latitude longitude
## 1  Aston.Hill 52.50385 -3.034178
## 2  Bottesford 52.93028 -0.814722
## 3 Bush.Estate 55.86228 -3.205782
## 4 Eskdalemuir 55.31531 -3.206111
## 5   Glazebury 53.46008 -2.472056
## 6     Harwell 51.57108 -1.325283
```

In this example, we want to show what mean O_3 concentrations look like across the UK (and Ireland because Mace Head was included) and then consider the concentrations by season, and then take a look at peak hour concentrations. First it is necessary to calculate the means and maximums by season:

Figure 25.1 shows the annual mean concentration of O_3 at UK and Ireland sites. It is clear that the highest concentrations of O_3 are at Mace Head (Ireland) and Strath Vaich (Scotland) — sites that are well exposed to 'clean' North Atlantic air and where deposition processes are not so important; at least at Mace Head.

For mean concentrations Figure 25.2 shows that springtime concentrations are highest, which will in part be due to the northern hemispheric peak in O_3 concentrations (Monks 2000). Concentrations are particularly high at the remote sites of Mace Head (Ireland) and Strath Vaich (Scotland). By contrast, Figure 25.3 shows the peak hourly concentration of O_3. In this case there is a very different distribution of O_3 concentrations. The highest concentrations are now observed in the south-east of England in summer, which will be due to regional scale pollution episodes. By contrast, the wintertime O_3 concentrations are much lower everywhere.

```
## cut data into seasons
## load plyr package
library(plyr)
o3Measurements <- cutData(o3Measurements, "season")
## calculate means/maxes and merge...
annual <- ddply(o3Measurements, .(site), numcolwise(mean), na.rm = TRUE)
## by site AND season
means <- ddply(o3Measurements, .(site, season), numcolwise(mean), na.rm = TRUE)
peaks <- ddply(o3Measurements, .(site, season), numcolwise(max), na.rm = TRUE)
annual <- merge(annual, siteDetails, by = "site")
means <- merge(means, siteDetails, by = "site")
peaks <- merge(peaks, siteDetails, by = "site")

## now make first plot
GoogleMapsPlot(annual, lat = "latitude", long = "longitude", pollutant = "o3",
               maptype = "roadmap", col = "jet")
```

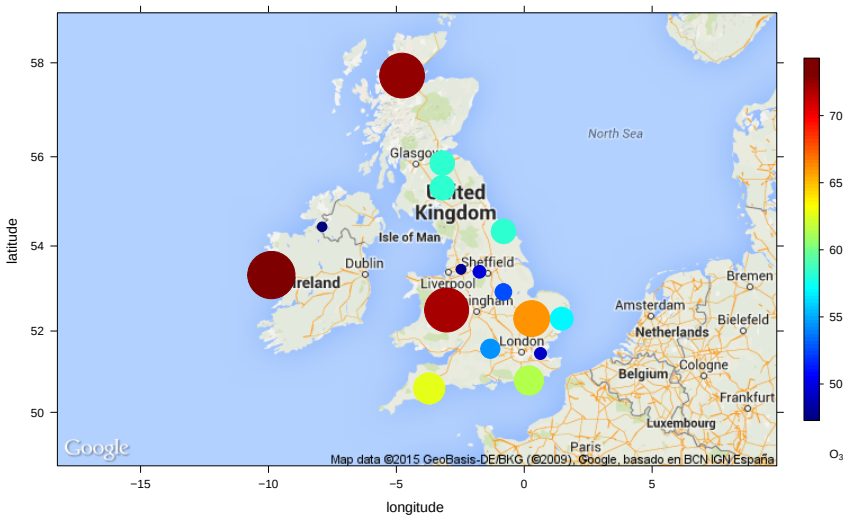


FIGURE 25.1 Mean concentrations of O_3 around the UK and Ireland (μg m^{-3}).

```
GoogleMapsPlot(means, lat = "latitude", long = "longitude", pollutant = "o3",
               type = "season", maptype = "roadmap", col = "jet")
```

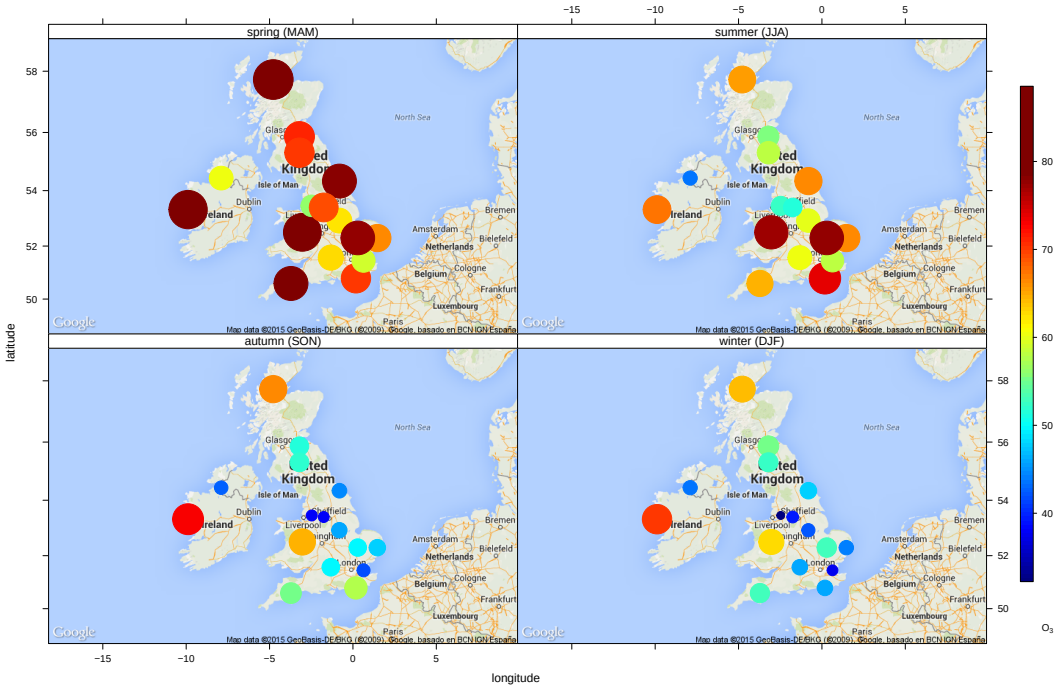


FIGURE 25.2 Mean hourly concentrations of O_3 around the UK and Ireland split by season (µg m^{-3}).

```
GoogleMapsPlot(peaks, lat = "latitude", long = "longitude", pollutant = "o3",
               type = "season", maptype = "roadmap", col = "jet")
```

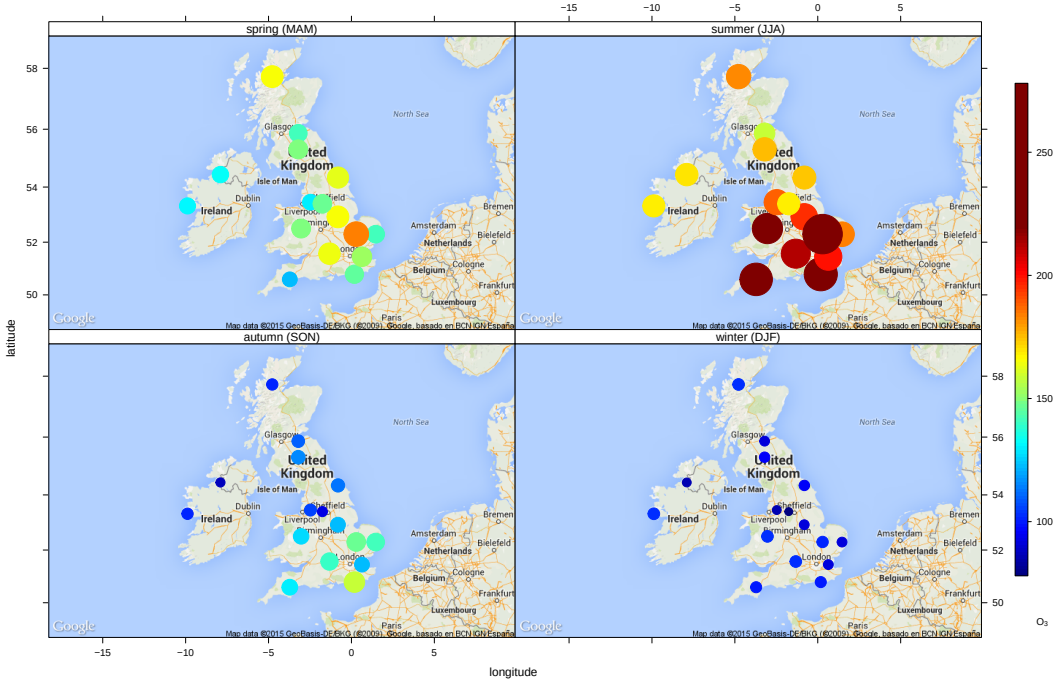


FIGURE 25.3 Maximum hourly concentrations of O_3 around the UK and Ireland split by season (µg m^{-3}).

26 openair back trajectory functions

Back trajectories are extremely useful in air pollution and can provide important information on air mass origins. Despite the clear usefulness of back trajectories, their use tends to be restricted to the research community. Back trajectories are used for many purposes from understanding the origins of air masses over a few days to undertaking longer term analyses. They are often used to filter air mass origins to allow for more refined analyses of air pollution — for example trends in concentration by air mass origin. They are often also combined with more sophisticated analyses such as cluster analysis to help group similar type of air mass by origin.

Perhaps one of the reasons why back trajectory analysis is not carried out more often is that it can be time consuming to do. This is particularly so if one wants to consider several years at several sites. It can also be difficult to access back trajectory data. In an attempt to overcome some of these issues and expand the possibilities for data analysis, **openair** makes several functions available to access and analyse pre-calculated back trajectories.

Currently these functions allow for the import of pre-calculated back trajectories are several pre-define locations and some trajectory plotting functions. In time all of these functions will be developed to allow more sophisticated analyses to be undertaken. Also it should be recognised that these functions are in their early stages of development and will may continue to change and be refined.

This `importTraj` 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`). 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. The data are stored on web-servers at King's College London in a similar way to `importKCL`, which makes it very easy to import pre-processed trajectory data for a range of locations and years. **Note — the back trajectories have been pre-calculated for specific locations and stored as .RData objects. Users should contact David Carslaw to request the addition of other locations.** So far only a few receptors are available to users but in time the number will increase. It should be feasible for example to run back trajectories for the past 20 years at all the EMEP sites in Europe.[15]

Users may for various reasons wish to run HYSPLIT themselves e.g. for different starting heights, longer periods or more locations. Code and instructions have been provided in Appendix D for users wishing to do this. Users can also use different means of calculating back trajectories e.g. ECMWF and plot them in **openair** provided a few basic fields are present: `date` (POSIXct), `lat` (decimal latitude), `lon` (decimal longitude) and `hour.inc` the hour offset from the arrival date (i.e. from zero decreasing to the length of the back trajectories). See `?importTraj` for more details.

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

[15] It takes about 15 hours to run 20 years of 96-hour back trajectories at 3-hour intervals.

trajectories separately.

openair uses the `mapproj` package to allow users to user different map projections. By default the projection used is Lambert conformal, which is a conic projection best used for mid-latitude areas. The HYSPLIT model itself will use any one of three different projections depending on the latitude of the origin. If the latitude greater than 55.0 (or less than −55.0) then a polar stereographic projection is used, if the latitude greater than −25.0 and less than 25.0 the mercator projection is used and elsewhere (the mid-latitudes) the Lambert projection. All these projections (and many others) are available in the `mapproj` package.

Users should see the help file for `importTraj` to get an up to date list of receptors where back trajectories have been calculated.

As an example, we will import trajectories for London in 2010. Importing them is easy:

```
traj <- importTraj(site = "london", year = 2010)
```

The file itself contains lots of information that is of use for plotting back trajectories:

```
head(traj)
```

```
##   receptor year month day hour hour.inc    lat    lon height pressure
## 1        1 2010     1   1    9        0 51.500 -0.100   10.0    994.7
## 2        1 2010     1   1    8       -1 51.766  0.057   10.3    994.9
## 3        1 2010     1   1    7       -2 52.030  0.250   10.5    995.0
## 4        1 2010     1   1    6       -3 52.295  0.488   10.8    995.0
## 5        1 2010     1   1    5       -4 52.554  0.767   11.0    995.4
## 6        1 2010     1   1    4       -5 52.797  1.065   11.3    995.6
##                 date2               date
## 1 2010-01-01 09:00:00 2010-01-01 09:00:00
## 2 2010-01-01 08:00:00 2010-01-01 09:00:00
## 3 2010-01-01 07:00:00 2010-01-01 09:00:00
## 4 2010-01-01 06:00:00 2010-01-01 09:00:00
## 5 2010-01-01 05:00:00 2010-01-01 09:00:00
## 6 2010-01-01 04:00:00 2010-01-01 09:00:00
```

The `traj` data frame contains among other things the latitude and longitude of the back trajectory, the height (m) and pressure (Pa) of the trajectory. The `date` field is the *arrival* time of the air-mass and is useful for linking with ambient measurement data.

The `trajPlot` function is used for plotting back trajectory lines and density plots and has the following options:

`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.

`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 `?mapproj` 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"`.

`...` 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.

```
trajPlot(selectByDate(traj, start = "15/4/2010", end ="21/4/2010"))

## Loading required package: mapproj
```

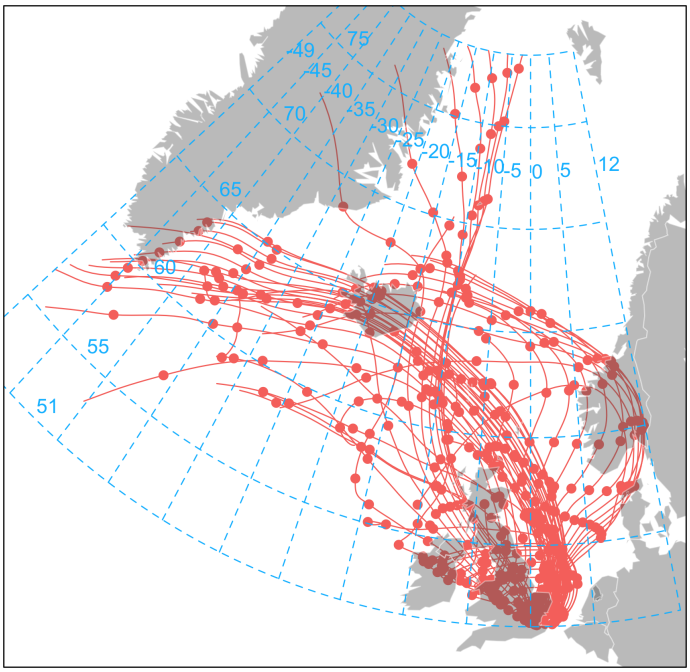


FIGURE 26.1 96-hour HYSPLIT back trajectories centred on London for 7 days in April 2010.

Next, we consider how to plot back trajectories with a few simple examples. The first example will consider a potentially interesting period when the Icelandic volcano, Eyjafjallajökull erupted in April 2010. The eruption of Eyjafjallajökull resulted in a flight-ban that lasted six days across many European airports. In Figure 26.1 `selectByDate` is used to consider the 7 days of interest and we choose to plot the back trajectories as lines rather than points (the default). Figure 26.1 does indeed show that many of the back trajectories originated from Iceland over this period. Note also the plot automatically includes a world base map. The base map itself is not at very high resolution by default but is useful for the sorts of spatial scales that back trajectories exist over. The base map is also global, so provided that there are pre-calculated back trajectories, these maps can be generated anywhere in the world. 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.[16]

Note that `trajPlot` will only plot *full length* trajectories. This can be important when plotting something like a single month e.g. by using `selectByDate` when on partial sections of some trajectories may be selected.

There are a few other ways of representing the data shown in Figure 26.1. For example, it might be useful to plot the trajectories for each day. To do this we need to make a new column 'day' which can be used in the plotting. The first example considers plotting the back trajectories in separate panels (Figure 26.2).

Another way of plotting the data is to group the trajectories by day and colour them. This time we also set a few other options to get the layout we want — shown in Figure 26.3.

[16] You will need to load the `mapdata` package i.e. `library(mapdata)`.

```
## make a day column
traj$day <- as.Date(traj$date)

## plot it choosing a specfic layout
trajPlot(selectByDate(traj, start = "15/4/2010", end = "21/4/2010"),
         type = "day", layout = c(7, 1))
```

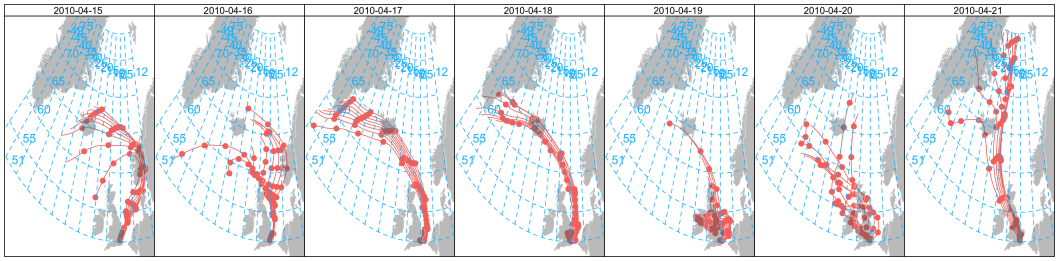


FIGURE 26.2 96-hour HYSPLIT back trajectories centred on London for 7 days in April 2010, shown separately for each day.

```
trajPlot(selectByDate(traj, start = "15/4/2010", end = "21/4/2010"),
         group = "day", col = "jet", lwd = 2, key.pos = "top",
         key.col = 4)
```

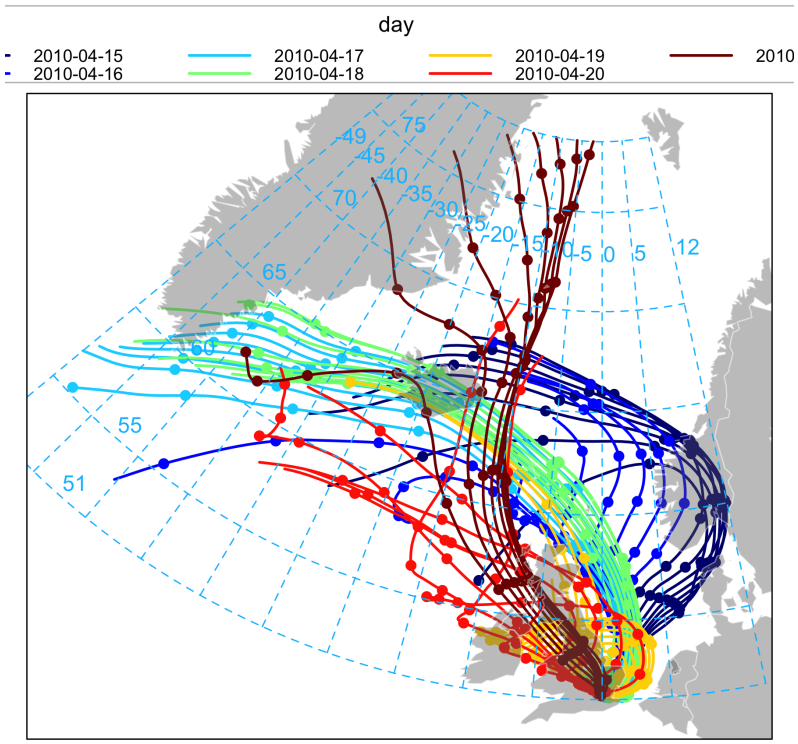


FIGURE 26.3 96-hour HYSPLIT back trajectories centred on London for 7 days in April 2010, shown grouped for each day and coloured accordingly.

So far the plots have provided information on where the back trajectories come from, grouped or split by day. It is also possible, in common with most other **openair** functions to split the trajectories by many other variables e.g. month, season and so on. However, perhaps one of the most useful approaches is to link the back trajectories with the concentrations of a pollutant. As mentioned previously, the back trajectory data has a column 'date' representing the arrival time of the air mass that can be used to link with concentration measurements. A couple of steps are required to do this using the `merge` function.

```
## import data for North Kensington
kc1 <- importAURN("kc1", year =2010)
# now merge with trajectory data by 'date'
traj <- merge(traj, kc1, by = "date")
## look at first few lines
head(traj)
```

```
##                  date receptor year month        day hour hour.inc    lat
## 1 2010-01-01 09:00:00        1 2010     1 2010-01-01    9        0 51.500
## 2 2010-01-01 09:00:00        1 2010     1 2010-01-01    8       -1 51.766
## 3 2010-01-01 09:00:00        1 2010     1 2010-01-01    7       -2 52.030
## 4 2010-01-01 09:00:00        1 2010     1 2010-01-01    6       -3 52.295
## 5 2010-01-01 09:00:00        1 2010     1 2010-01-01    5       -4 52.554
## 6 2010-01-01 09:00:00        1 2010     1 2010-01-01    4       -5 52.797
##      lon height pressure               date2 o3 no2  co so2 pm10 nox no pm2.5
## 1 -0.100   10.0    994.7 2010-01-01 09:00:00 46  29 0.3   0    8  38  6    NA
## 2  0.057   10.3    994.9 2010-01-01 08:00:00 46  29 0.3   0    8  38  6    NA
## 3  0.250   10.5    995.0 2010-01-01 07:00:00 46  29 0.3   0    8  38  6    NA
## 4  0.488   10.8    995.0 2010-01-01 06:00:00 46  29 0.3   0    8  38  6    NA
## 5  0.767   11.0    995.4 2010-01-01 05:00:00 46  29 0.3   0    8  38  6    NA
## 6  1.065   11.3    995.6 2010-01-01 04:00:00 46  29 0.3   0    8  38  6    NA
##   nv2.5 v2.5 nv10 v10 ws wd                 site code
## 1    NA   NA    8   0 NA NA London N. Kensington  KC1
## 2    NA   NA    8   0 NA NA London N. Kensington  KC1
## 3    NA   NA    8   0 NA NA London N. Kensington  KC1
## 4    NA   NA    8   0 NA NA London N. Kensington  KC1
## 5    NA   NA    8   0 NA NA London N. Kensington  KC1
## 6    NA   NA    8   0 NA NA London N. Kensington  KC1
```

This time we can use the option `pollutant` in the function `trajPlot`, which will plot the back trajectories coloured by the concentration of a pollutant. Figure 26.4 does seem to show elevated PM_{10} concentrations originating from Iceland over the period of interest. In fact, these elevated concentrations occur on two days as shown in Figure 26.2. However, care is needed when interpreting such data because other analysis would need to rule out other reasons why PM_{10} could be elevated; in particular due to local sources of PM_{10}. There are lots of **openair** functions that can help here e.g. `timeVariation` or `timePlot` to see if NO_x concentrations were also elevated (which they seem to be). It would also be worth considering other sites for back trajectories that could be less influenced by local emissions.

However, it is possible to account for the PM that is local to some extent by considering the relationship between NO_x and PM_{10} (or $PM_{2.5}$). For example, using `scatterPlot` (not shown):

```
scatterPlot(kc1, x = "nox", y = "pm2.5", avg = "day", linear = TRUE)
```

which suggests a gradient of 0.084. Therefore we can remove the PM_{10} that is associated NO_x in `kc1` data, making a new column `pm.new`:

```
trajPlot(selectByDate(traj, start = "15/4/2010", end = "21/4/2010"),
         pollutant = "pm10", col = "jet", lwd =2)
```

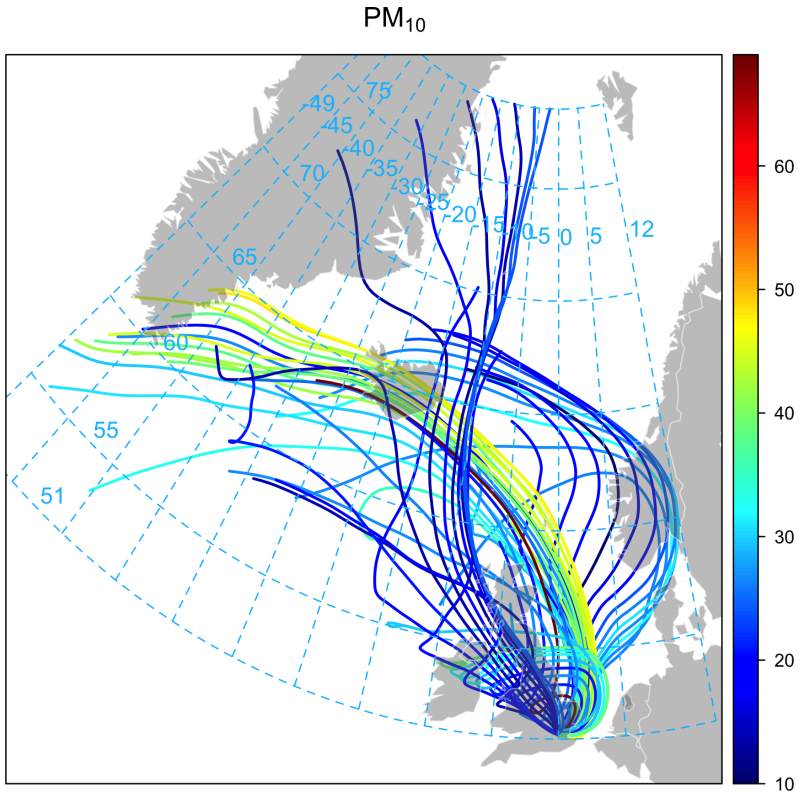


FIGURE 26.4 96-hour HYSPLIT back trajectories centred on London for 7 days in April 2010, coloured by the concentration of PM_{10} (µg m^{-3}).

```
kc1 <- transform(kc1, pm.new = pm10 - 0.084 * nox)
```

We have already merged `kc1` with `traj`, so to keep things simple we import `traj` again and merge it with `kc1`. Note that if we had thought of this initially, `pm.new` would have been calculated first before merging with `traj`.

```
traj <- importTraj(site = "london", year = 2010)
traj <- merge(traj, kc1, by = "date")
```

Now it is possible to plot the trajectories:

```
trajPlot(selectByDate(traj, start = "15/4/2010", end = "21/4/2010"),
          pollutant = "pm.new", col = "jet", lwd = 2)
```

Which, interestingly still clearly shows elevated PM_{10} concentrations for those two days that cross Iceland. The same is also true for $PM_{2.5}$. However, as mentioned previously, checking other sites in more rural areas would be a good idea.

26.1 Trajectory gridded frequencies

The HYSPLIT model itself contains various analysis options for gridding trajectory data. Similar capabilities are also available in **openair** where the analyses can be extended using other **openair** capabilities. It is useful to gain an idea of where trajectories come

from. Over the course of a year representing trajectories as lines or points results in a lot of over-plotting. Therefore it is useful to grid the trajectory data and calculate various statistics by considering latitude-longitude intervals.

The first analysis considers the number of unique trajectories in a particular grid square. This is achieved by using the `trajLevel` function and setting the `statistic` option to "frequency". Figure 26.5 shows the frequency of back trajectory crossings for the North Kensington data. In this case it highlights that most trajectory origins are from the west and north for 2010 at this site. Note that in this case, `pollutant` can just be the trajectory height (or another numeric field) rather than an actual pollutant because only the frequencies are considered.

The `trajLevel` function has the following options:

`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.

`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 `?mapproj` 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"`.

```
trajLevel(traj, statistic = "frequency")
```

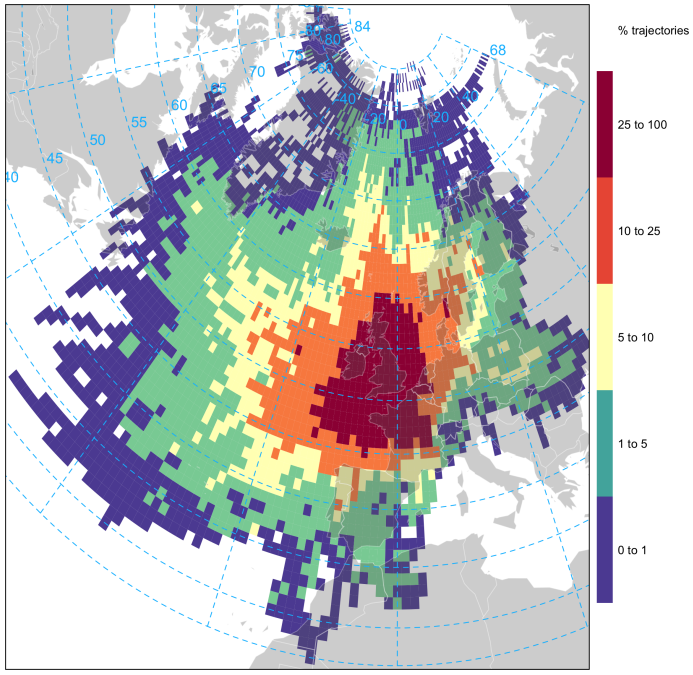


FIGURE 26.5 Gridded back trajectory frequencies. The `border = NA` option removes the border around each grid cell.

`...` 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.

It is also possible to use hexagonal binning to gain an idea about trajectory frequencies. In this case each 3-hour point along each trajectory is used in the counting. The code below focuses more on Europe and uses the hexagonal binning method. Note that the effect of the very few high number of points at the origin has been diminished by plotting the data on a log scale — see page 191 for details.

26.2 Trajectory source contribution functions

Back trajectories offer the possibility to undertake receptor modelling to identify the location of major emission sources. When many back trajectories (over months to years) are analysed in specific ways they begin to show the geographic origin most associated with elevated concentrations. With enough (dissimilar) trajectories those locations leading to the highest concentrations begin to be revealed. When a whole year of back trajectory data is plotted the individual back trajectories can extend 1000s of km. There are many approaches using back trajectories in this way and Fleming et al. (2012) provide a good overview of the methods available. **openair** has implemented a few of these techniques and over time these will be refined and extended.

```
trajLevel(subset(traj, lat > 30 & lat <70 & lon > -30 & lon <20),
          method = "hexbin", col = "jet",
          xbin = 40)
```

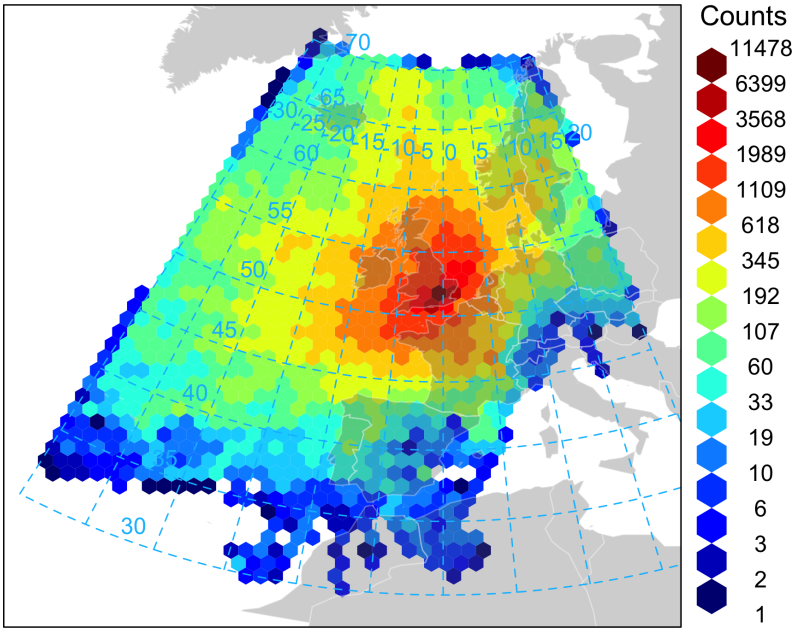


FIGURE 26.6 Gridded back trajectory frequencies with hexagonal binning.

26.2.1 Identifying the contribution of high concentration back trajectories

A useful analysis to undertake is to consider the pattern of frequencies for two different conditions. In particular, there is often interest in the origin of high concentrations for different pollutants. For example, compared with data over a whole year, how do the frequencies of occurrence differ? Figure 26.7 shows an example of such an analysis for PM_{10} concentrations. By default the function will compare concentrations >90th percentile with the full year. The percentile level is controlled by the option `percentile`. Note also there is an option `min.bin` that will exclude grid cells where there are fewer than `min.bin` data points.

Figure 26.7 shows that compared with the whole year, high PM_{10} concentrations (>90th percentile) are more prevalent when the trajectories originate from the east, which is seen by the positive values in the plot. Similarly there are relatively fewer occurrences of these high concentration back trajectories when they originate from the west. This analysis is in keeping with the highest PM_{10} concentrations being largely controlled by secondary aerosol formation from air-masses originating during anticyclonic conditions from mainland Europe.

Note that it is also possible to use conditioning with these plots. For example to split the frequency results by season:

```
trajLevel(traj, pollutant = "pm10", statistic = "frequency", col = "heat",
          type = "season", border = "white")
```

```
trajLevel(traj, pollutant = "pm10", statistic = "difference",
          col = c("skyblue", "white", "tomato"), min.bin = 3, border = NA)
```

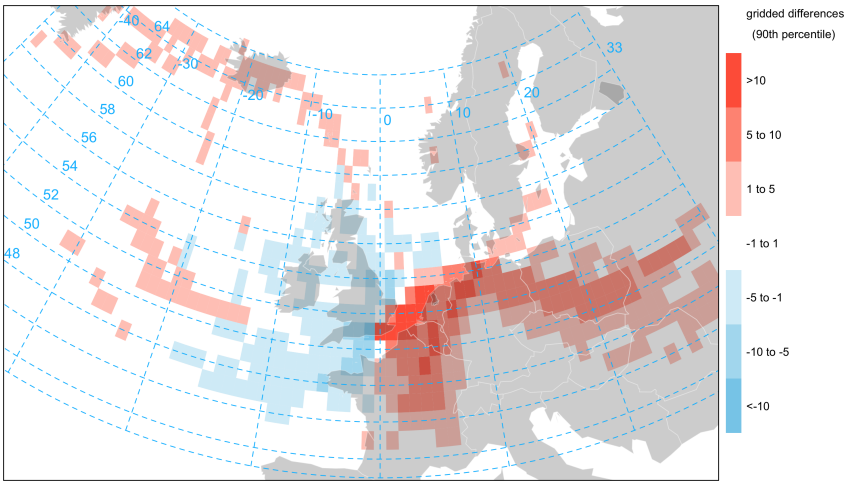


FIGURE 26.7 Gridded back trajectory frequencies showing the percentage difference in occurrence for high PM_{10} concentrations (90th percentile) compared with conditions over the full year.

26.2.2 Allocating trajectories to different wind sectors

One of the key aspects of trajectory analysis is knowing something about where air masses have come from. Cluster analysis can be used to group trajectories based on their origins and this is discussed in Section 26.3. A simple approach is to consider different wind sectors e.g. N, NE, E and calculate the proportion of time a particular back trajectory resides in a specific sector. It is then possible to allocate a particular trajectory to a sector based on some assumption about the proportion of time it is in that sector — for example, assume a trajectory is from the west sector if it spends at least 50% of its time in that sector or otherwise record the allocation as 'unallocated'. The code below can be used as the basis of such an approach.

First we import the trajectories, which in this case are for London in 2010:

```
traj <- importTraj(site = "london", year = 2010)
```

```
## need start/end lat and lon to work out angles
id <- which(traj$hour.inc == 0)
y0 <- traj$lat[id[1]]
x0 <- traj$lon[id[1]]

## calculate angle and then assign sector
traj <- transform(traj, angle = atan2(lon - x0, lat - y0) * 360 / 2 / pi)
ids <- which(traj$angle < 0)
traj$angle[ids] <- traj$angle[ids] + 360

traj$sector <- cut(traj$angle, breaks = seq(22.5, 382.5, 45),
                               labels = c("NE", "E", "SE", "S", "SW", "W",
                               "NW", "N"))
traj[ , "sector"][is.na(traj[ , "sector"])] <- "N" # for wd < 22.5

## count frequencies of sectors for each trajectory and the maximum
alloc <- tapply(traj$sector, traj$date, table)
alloc <- as.data.frame(do.call(rbind, alloc))
alloc$max <- apply(alloc, 1, max)

## identify the most frequent sector
alloc$sec <- sapply(1:nrow(alloc),
                    function(x) colnames(alloc)[which.max(alloc[x, ])])

## assign to most frequent sector, or label unallocated
## below assumes at least 50 out of 96 hours for assignment
alloc$sec <- ifelse(alloc$max > 50, alloc$sec, "unallocated")
alloc$date <- as.POSIXct(rownames(alloc), "GMT")

alloc <- alloc[, c("date", "sec")]

## merge with orginal data
traj <- merge(traj, alloc, by = "date", all = TRUE)
```

Now it is possible to post-process the data. `traj` now has the angle, sector and allocation (`sec`).

```
head(traj)
```

```
##                  date receptor year month day hour hour.inc    lat    lon
## 1 2010-01-01 09:00:00        1 2010     1   1    9        0 51.500 -0.100
## 2 2010-01-01 09:00:00        1 2010     1   1    8       -1 51.766  0.057
## 3 2010-01-01 09:00:00        1 2010     1   1    7       -2 52.030  0.250
## 4 2010-01-01 09:00:00        1 2010     1   1    6       -3 52.295  0.488
## 5 2010-01-01 09:00:00        1 2010     1   1    5       -4 52.554  0.767
## 6 2010-01-01 09:00:00        1 2010     1   1    4       -5 52.797  1.065
##   height pressure               date2    angle sector         sec
## 1   10.0    994.7 2010-01-01 09:00:00  0.00000      N unallocated
## 2   10.3    994.9 2010-01-01 08:00:00 30.55019     NE unallocated
## 3   10.5    995.0 2010-01-01 07:00:00 33.43987     NE unallocated
## 4   10.8    995.0 2010-01-01 06:00:00 36.48747     NE unallocated
## 5   11.0    995.4 2010-01-01 05:00:00 39.44005     NE unallocated
## 6   11.3    995.6 2010-01-01 04:00:00 41.93103     NE unallocated
```

First, merge the air quality data from North Kensington:

```
traj <- merge(traj, kc1, by = "date")
```

We can work out the mean concentration by allocation, which shows the clear importance for the east and south-east sectors.

```
tapply(traj$pm2.5, traj$sec, mean, na.rm = TRUE)
```

```
##           E           N          NE          NW           S          SE
##    21.64671    11.79452    12.61986    13.60474    14.50000    28.72500
##          SW unallocated           W
##    10.80992    15.47481    11.75882
```

Finally, the percentage of the year in each sector can be calculated as follows:

```
100 * prop.table(table(traj$sec))
```

```
##
##           E           N          NE          NW           S          SE
##    6.712201    5.512401   10.252995    9.164367    2.170508    2.067150
##          SW unallocated           W
##    8.750937   29.530061   25.839380
```

26.2.3 Potential Source Contribution Function (PSCF)

If `statistic = "pscf"` then the Potential Source Contribution Function (PSCF) is plotted. The PSCF calculates the probability that a source is located at latitude *i* and longitude *j* (Fleming et al. 2012; Pekney et al. 2006). The PSCF is somewhat analogous to the CPF function described on page 116 that considers local wind direction probabilities. In fact, the two approaches have been shown to work well together (Pekney et al. 2006). The PSCF approach has been widely used in the analysis of air mass back trajectories. Ara Begum et al. (2005) for example assessed the method against the known locations of wildfires and found it performed well for $PM_{2.5}$, EC (elemental carbon) and OC (organic carbon) and that other (non-fire related) species such as sulphate had different source origins. 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

$$PSCF = \frac{m_{ij}}{n_{ij}} \tag{10}$$

where n_{ij} is the number of times that the trajectories passed through the cell (i, j) and m_{ij} is the number of times that a source concentration was high when the trajectories passed through the cell (i, j). The criterion for determining m_{ij} is controlled by `percentile`, which by default is 90. Note also that cells with few data have a weighting factor applied to reduce their effect.

An example of a PSCF plot is shown in Figure 26.8 for $PM_{2.5}$ for concentrations >90th percentile. This Figure gives a very clear indication that the principal (high) sources are dominated by source origins in mainland Europe — particularly around the Benelux countries.

26.2.4 Concentration Weighted Trajectory (CWT)

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 (Hsu et al. 2003). 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. This approach is sometimes referred to as the CWT or CF (concentration field). A grid domain was used as in the PSCF method. For each grid cell, the mean

```
trajLevel(subset(traj, lon > -20 & lon < 20 & lat > 45 & lat < 60),
          pollutant = "pm2.5", statistic = "pscf", col = "increment",
          border = NA)
```

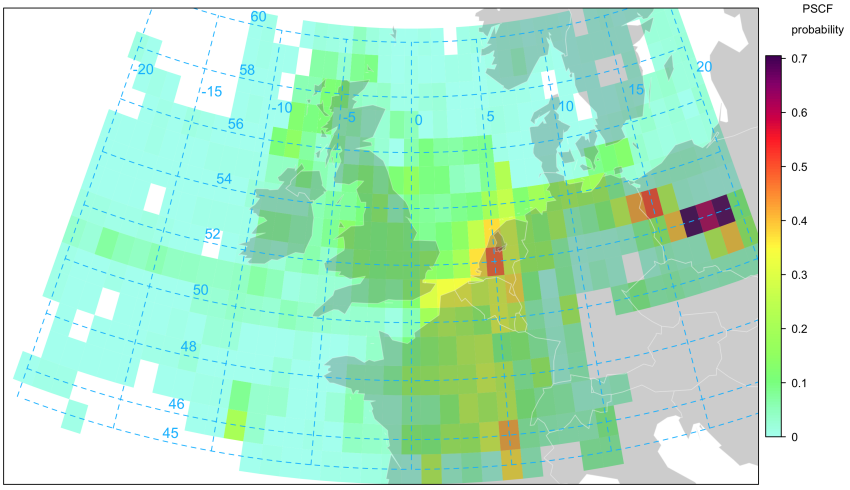


FIGURE 26.8 PSCF probabilities for $PM_{2.5}$ concentrations (90th percentile).

(CWT) or logarithmic mean (used in the Residence Time Weighted Concentration (RTWC) method) concentration of a pollutant species was calculated as follows:

$$ln(\overline{C}_{ij}) = \frac{1}{\sum_{k=1}^{N} \tau_{ijk}} \sum_{k=1}^{N} ln(c_k)\tau_{ijk} \tag{11}$$

where i and j are the indices of grid, k the index of trajectory, N the total number of trajectories used in analysis, c_k the pollutant concentration measured upon arrival of trajectory k, and τ_{ijk} the residence time of trajectory k in grid cell (i, j). A high value of $\overline{C}_{ij}$ means that, air parcels passing over cell (i, j) would, on average, cause high concentrations at the receptor site.

Figure 26.9 shows the situation for $PM_{2.5}$ concentrations. It was calculated by recording the associated $PM_{2.5}$ concentration for each point on the back trajectory based on the arrival time concentration using 2010 data. The plot shows the geographic areas most strongly associated with high $PM_{2.5}$ concentrations i.e. to the east in continental Europe. Both the CWT and PSCF methods have been shown to give similar results and each have their advantages and disadvantages (Lupu and Maenhaut 2002; Hsu et al. 2003). Figure 26.9 can be compared with Figure 26.8 to compare the overall identification of source regions using the CWT and PSCF techniques. Overall the agreement is good in that similar geographic locations are identified as being important for $PM_{2.5}$.

Figure 26.9 is useful, but it can be clearer if the trajectory surface is smoothed, which has been done for $PM_{2.5}$ concentrations shown in Figure 26.10.

In common with most other **openair** functions, the flexible 'type' option can be used to split the data in different ways. For example, to plot the smoothed back trajectories for $PM_{2.5}$ concentrations by season.

```
trajLevel(subset(traj, lat > 40 & lat < 70 & lon >-20 & lon <20),
          pollutant = "pm2.5", type = "season", statistic = "pscf",
          layout = c(4, 1))
```

It should be noted that it makes sense to analyse back trajectories for pollutants that

```
trajLevel(subset(traj,lon > -20 & lon < 20 & lat > 45 & lat < 60),
          pollutant = "pm2.5", statistic="cwt", col = "increment",
          border = "white")
```

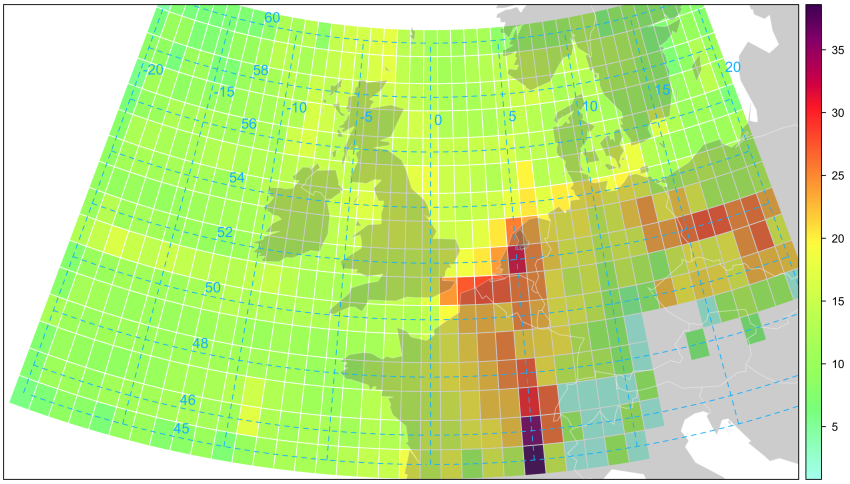


FIGURE 26.9 Gridded back trajectory concentrations showing mean $PM_{2.5}$ concentrations using the CWT approach.

```
trajLevel(subset(traj, lat > 45 & lat < 60 & lon >-20 & lon <20),
          pollutant ="pm2.5", statistic = "cwt", smooth = TRUE,
          col = "increment")
```

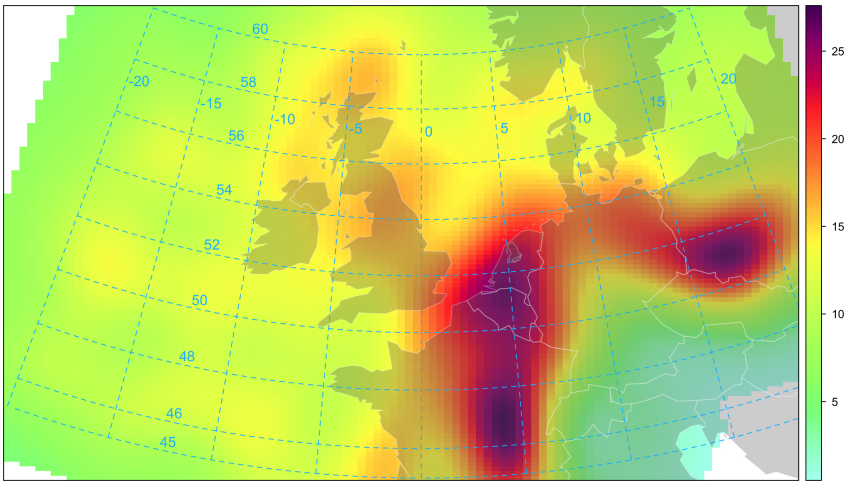


FIGURE 26.10 Gridded and smoothed back trajectory concentrations showing mean $PM_{2.5}$ concentrations using the CWT approach.

have a large regional component — such as particles or O_3. It makes little sense to analyse pollutants that are known to have local impacts e.g. NO_x. However, a species such as NO_x can be helpful to exclude 'fresh' emissions from the analysis.

26.3 Back trajectory cluster analysis with the `trajCluster` function

Often it is useful to use cluster analysis on back trajectories to group similar air mass origins together. The principal purpose of clustering back trajectories is to post-process data according to cluster origin. By grouping data with similar geographic origins it is possible to gain information on pollutant species with similar chemical histories. There are several ways in which clustering can be carried out and several measures of the similarity of different clusters. A key issue is how the *distance matrix* is calculated, which determines the similarity (or dissimilarity) of different back trajectories. The simplest measure is the Euclidean distance. However, an angle-based measure is also often used. The two distance measures are defined below. In **openair** the distance matrices are calculated using C++ code because their calculation is computationally intensive. Note that these calculations can also be performed directly in the HYSPLIT model itself.

The Euclidean distance between two trajectories is given by Equation 12. Where X_1, Y_1 and X_2, Y_2 are the latitude and longitude coordinates of back trajectories 1 and 2, respectively. n is the number of back trajectory points (96 hours in this case).

$$d_{1,2} = \left(\sum_{i=1}^{n} ((X_{1i} - X_{2i})^2 + (Y_{1i} - Y_{2i}))^2 \right)^{1/2} \tag{12}$$

The *angle* distance matrix is a measure of how similar two back trajectory points are in terms of their angle from the origin i.e. the starting location of the back trajectories. The angle-based measure will often capture some of the important circulatory features in the atmosphere e.g. situations where there is a high pressure located to the east of the UK. However, the most appropriate distance measure will be application dependent and is probably best tested by the extent to which they are able to differentiate different air-mass characteristics, which can be tested through post-processing. The angle-based distance measure is defined as:

$$d_{1,2} = \frac{1}{n} \sum_{i=1}^{n} cos^{-1} \left(0.5 \frac{A_i + B_i + C_i}{\sqrt{A_i B_i}} \right) \tag{13}$$

where

$$A_i = (X_1(i) - X_0)^2 + (Y_1(i) - Y_0)^2 \tag{14}$$

$$B_i = (X_2(i) - X_0)^2 + (Y_2(i) - Y_0)^2 \tag{15}$$

$$C_i = (X_2(i) - X_1(i))^2 + (Y_2(i) - Y_1(i))^2 \tag{16}$$

where X_0 and Y_0 are the coordinates of the location being studied i.e. the starting location of the trajectories.

The `trajCluster` function has the following options:

`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`?mapproj` 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.

`...` 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.

```
clust <- trajCluster(traj, method = "Angle", n.cluster= 6, col = "Set2")
```

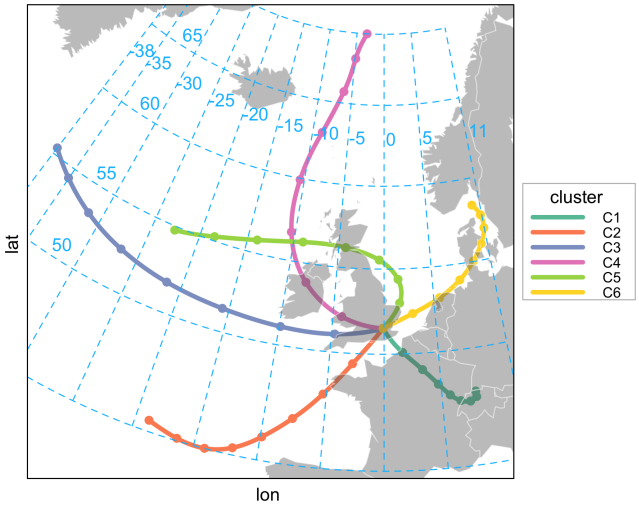


FIGURE 26.11 The 6-cluster solution to back trajectories calculated for the London North Kensington site for 2011 showing the mean trajectory for each cluster.

As an example we will consider back trajectories for London in 2011.

First, the back trajectory data for London is imported together with the air pollution data for the North Kensington site (KC1).

```
traj <- importTraj(site = "london", year = 2011)
kc1 <- importKCL(site = "kc1", year = 2011)

## NOTE - mass units are used
## ug/m3 for NOx, NO2, SO2, O3; mg/m3 for CO
## PM10_raw is raw data multiplied by 1.3
```

The clusters are straightforward to calculate. In this case the back trajectory data (`traj`) is supplied and the angle-based distance matrix is used. Furthermore, we choose to calculate 6 clusters and choose a specific colour scheme. In this case we read the output from `trajCluster` into a variable `clust` so that the results can be post-processed.

`clust` returns all the back trajectory information together with the cluster (as a character). This data can now be used together with other data to analyse results further. However, first it is possible to show all trajectories coloured by cluster, although for a year of data there is significant overlap and it is difficult to tell them apart.

```
trajPlot(clust, group = "cluster")
```

A useful way in which to see where these air masses come from by trajectory is to produce a frequency plot by cluster. Such a plot (not shown, but code below) provides a good indication of the spread of the different trajectory clusters as well as providing an indication of where air masses spend most of their time. For the London 2011 data it can be seen cluster 1 is dominated by air from the European mainland to the south.

```
trajLevel(clust, type = "cluster", col = "increment", border = NA)
```

Perhaps more useful is to merge the cluster data with measurement data. In this

case the data at North Kensington site are used. Note that in merging these two data frames it is not necessary to retain all 96 back trajectory hours and for this reason we extract only the first hour.

```
kc1 <- merge(kc1, subset(clust, hour.inc == 0), by = "date")
```

Now `kc1` contains air pollution data identified by cluster. The size of this data frame is about a third of the original size because back trajectories are only run every 3 hours.

The numbers of each cluster are given by:

```
table(kc1[, "cluster"])
```

```
##
##  C1  C2  C3  C4  C5  C6
## 347 661 989 277 280 333
```

i.e. is dominated by clusters 3 and 2 from west and south-west (Atlantic).

Now it is possible to analyse the concentration data according to the cluster. There are numerous types of analysis that can be carried out with these results, which will depend on what the aims of the analysis are in the first place. However, perhaps one of the first things to consider is how the concentrations vary by cluster. As the summary results below show, there are distinctly different mean concentrations of most pollutants by cluster. For example, clusters 1 and 6 are associated with much higher concentrations of PM_{10}— approximately double that of other clusters. Both of these clusters originate from continental Europe. Cluster 5 is also relatively high, which tends to come from the rest of the UK. Other clues concerning the types of air-mass can be gained from the mean pressure. For example, cluster 5 is associated with the highest pressure (1014 kPa), and as is seen in Figure 26.11 the shape of the line for cluster 5 is consistent with air-masses associated with a high pressure system (a clockwise-type sweep).

```
ddply(kc1, .(cluster), numcolwise(mean), na.rm = TRUE)
```

```
##   cluster      nox      no2       o3      so2        co pm10_raw     pm10
## 1      C1 89.48174 51.14386 31.94724 2.973813 0.3576075 32.33807 36.89308
## 2      C2 40.54300 30.74948 39.20698 1.316902 0.2260685 18.12048 17.55242
## 3      C3 46.92437 32.19128 39.51372 1.615229 0.2177106 19.39711 17.77589
## 4      C4 43.78059 30.52097 39.54522 1.323552 0.2019769 19.80298 17.67844
## 5      C5 56.64469 39.00516 41.25035 2.058801 0.2261857 24.37554 25.63715
## 6      C6 64.50604 42.25319 46.14615 2.670879 0.2847380 31.71167 37.03610
##       pm25     v2.5 nv10     nv2.5 receptor year    month      day     hour
## 1 31.34557 8.015291  NaN 23.330275        1 2011 7.659942 15.28242 10.11527
## 2 11.55000 3.131250  NaN  8.418750        1 2011 6.267776 15.45083 10.58850
## 3 11.23128 2.794413  NaN  8.436872        1 2011 7.470172 15.75329 10.41052
## 4 10.98770 2.172131  NaN  8.815574        1 2011 6.862816 17.75812 10.97112
## 5 16.52282 4.497925  NaN 12.024896        1 2011 5.225000 16.13214 10.12500
## 6 29.73000 7.568562  NaN 22.147157        1 2011 4.381381 15.79880 10.88288
##   hour.inc  lat  lon height pressure len
## 1        0 51.5 -0.1     10 1005.400  97
## 2        0 51.5 -0.1     10 1005.210  97
## 3        0 51.5 -0.1     10 1006.086  97
## 4        0 51.5 -0.1     10 1006.798  97
## 5        0 51.5 -0.1     10 1014.349  97
## 6        0 51.5 -0.1     10 1010.639  97
```

Simple plots can be generated from these results too. For example, it is easy to consider the temporal nature of the volatile component of $PM_{2.5}$ concentrations (`v2.5` in the `kc1` data frame). Figure 26.12 for example shows how the concentration of the

```
trendLevel(kc1, pollutant = "v2.5", type = "cluster", layout = c(6, 1),
           cols = "increment")
```

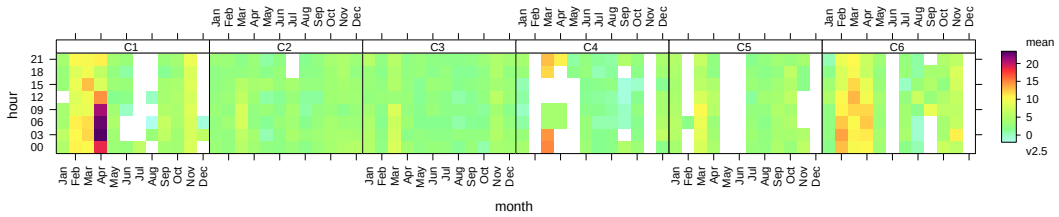


FIGURE 26.12 Some of the temporal characteristics of the volatile $PM_{2.5}$ component plotted by month and hour of the day and by cluster for the London North Kensington site for 2011.

```
timeProp(kc1, pollutant="pm25", avg.time = "day", proportion = "cluster",
         col="Set2", key.position = "top", key.columns = 6)
```

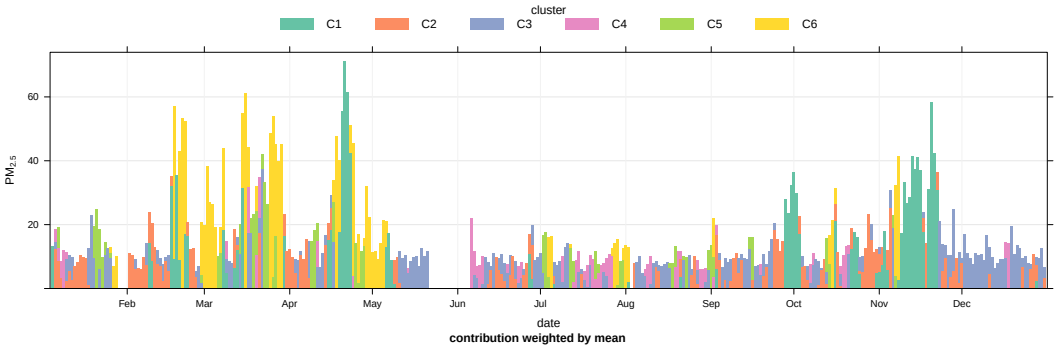


FIGURE 26.13 Temporal variation in daily $PM_{2.5}$ concentrations at the North Kensington site show by contribution of each cluster.

volatile component of $PM_{2.5}$ concentrations varies by cluster by plotting the hour of day-month variation. It is clear from Figure 26.12 that the clusters associated with the highest volatile $PM_{2.5}$ concentrations are clusters 1 and 6 (European origin) and that these concentrations peak during spring. There is less data to see clearly what is going on with cluster 5. Nevertheless, the cluster analysis has clearly separated different air mass characteristics which allows for more refined analysis of different air-mass types.

Similarly, as considered in Section 15, the `timeVariation` function can also be used to consider the temporal components.

Another useful plot to consider is `timeProp`, which can show how the concentration of a pollutant is comprised. In this case it is useful to plot the time series of $PM_{2.5}$ *and* show how much of the concentration is contributed to by each cluster. Such a plot is shown in Figure 26.13. It is now easy to see for example that during the spring months many of the high concentration events were due to clusters 1 and 6, which correspond to European origin air-masses as shown in Figure 26.11.

27 Model evaluation — the modStats function

27.1 Purpose

The modStats function provides key model evaluation statistics for comparing models against measurements and models against other models.

There are a very wide range of evaluation statistics that can be used to assess model performance. There is, however, no single statistic that encapsulates all aspects of interest. For this reason it is useful to consider several performance statistics and also to understand the sort of information or insight they might provide.

In the following definitions, O_i represents the ith observed value and M_i represents the ith modelled value for a total of n observations.

Fraction of predictions within a factor or two, *FAC2*

The fraction of modelled values within a factor of two of the observed values are the fraction of model predictions that satisfy:

$$0.5 \leq \frac{M_i}{O_i} \leq 2.0 \tag{17}$$

Mean bias, *MB*

The mean bias provides a good indication of the mean over or under estimate of predictions. Mean bias in the same units as the quantities being considered.

$$MB = \frac{1}{n}\sum_{i=1}^{N} M_i - O_i \tag{18}$$

Mean Gross Error, *MGE*

The mean gross error provides a good indication of the mean error regardless of whether it is an over or under estimate. Mean gross error is in the same units as the quantities being considered.

$$MGE = \frac{1}{n}\sum_{i=1}^{N} |M_i - O_i| \tag{19}$$

Normalised mean bias, *NMB*

The normalised mean bias is useful for comparing pollutants that cover different concentration scales and the mean bias is normalised by dividing by the observed concentration.

$$NMB = \frac{\sum_{i=1}^{n} M_i - O_i}{\sum_{i=1}^{n} O_i} \tag{20}$$

Normalised mean gross error, *NMGE*

The normalised mean gross error further ignores whether a prediction is an over or under estimate.

$$NMGE = \frac{\sum_{i=1}^{n} |M_i - O_i|}{\sum_{i=1}^{n} O_i} \tag{21}$$

Root mean squared error, *RMSE*

The RMSE is a commonly used statistic that provides a good overall measure of how close modelled values are to predicted values.

$$RMSE = \left(\frac{\sum_{i=1}^{n} (M_i - O_i)^2}{n} \right)^{1/2} \tag{22}$$

Correlation coefficient, *r*

The (Pearson) correlation coefficient is a measure of the strength of the linear relationship between two variables. If there is perfect linear relationship with positive slope between the two variables, $r = 1$. If there is a perfect linear relationship with negative slope between the two variables $r = -1$. A correlation coefficient of 0 means that there is no linear relationship between the variables. Note that `modStats` accepts an option `method`, which can be set to "kendall" and "spearman" for alternative calculations of *r*.

$$r = \frac{1}{(n-1)} \sum_{i=1}^{n} \left(\frac{M_i - \overline{M}}{\sigma_M} \right) \left(\frac{O_i - \overline{O}}{\sigma_O} \right) \tag{23}$$

Coefficient of Efficiency, *COE*

The *Coefficient of Efficiency* based on Legates and McCabe (2012) and Legates and McCabe Jr (1999). 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.

$$COE = 1.0 - \frac{\sum_{i=1}^{n} |M_i - O_i|}{\sum_{i=1}^{n} |O_i - \overline{O}|} \tag{24}$$

Index of Agreement, *IOA*

The *Index of Agreement*, *IOA* is commonly used in model evaluation (Willmott et al. 2011). It 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. It is defined as (with $c = 2$):

$$IOA = \begin{cases} 1.0 - \dfrac{\sum_{i=1}^{n} |M_i - O_i|}{c \sum_{i=1}^{n} |O_i - \overline{O}|}, when \\ \sum_{i=1}^{n} |M_i - O_i| \leq c \sum_{i=1}^{n} |O_i - \overline{O}| \\ \dfrac{c \sum_{i=1}^{n} |O_i - \overline{O}|}{\sum_{i=1}^{n} |M_i - O_i|} - 1.0, when \\ \sum_{i=1}^{n} |M_i - O_i| > c \sum_{i=1}^{n} |O_i - \overline{O}| \end{cases}$$

(25)

27.2 Options available

`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"`

27.3 Example of use

The function can be called very simply and only requires two numeric fields to compare. To show how the function works, some synthetic data will be generated for 5 models.

```
## observations; 100 random numbers
set.seed(10)
obs <- 100 * runif(100)
mod1 <- data.frame(obs, mod = obs + 10, model = "model 1")
mod2 <- data.frame(obs, mod = obs + 20 * rnorm(100), model = "model 2")
mod3 <- data.frame(obs, mod = obs - 10 * rnorm(100), model = "model 3")
mod4 <- data.frame(obs, mod = obs / 2 + 10 * rnorm(100), model = "model 4")
mod5 <- data.frame(obs, mod = obs * 1.5 + 3 * rnorm(100), model = "model 5")
modData <- rbind(mod1, mod2, mod3, mod4, mod5)
head(modData)
```

```
##         obs      mod   model
## 1 50.747820 60.74782 model 1
## 2 30.676851 40.67685 model 1
## 3 42.690767 52.69077 model 1
## 4 69.310208 79.31021 model 1
## 5  8.513597 18.51360 model 1
## 6 22.543662 32.54366 model 1
```

We now have a data frame with observations and predictions for 5 models. The evaluation of the statistics is given by:

```
modStats(modData, obs = "obs", mod = "mod", type = "model")
```

```
##     model   n FAC2         MB       MGE         NMB      NMGE      RMSE
## 1 model 1 100 0.89  10.000000 10.000000  0.22455508 0.2245551 10.000000
## 2 model 2 100 0.79   0.922391 16.592192  0.02071276 0.3725861 19.318041
## 3 model 3 100 0.88   1.013647  7.887325  0.02276196 0.1771139  9.451354
## 4 model 4 100 0.56 -20.603684 21.860655 -0.46266619 0.4908921 25.758520
## 5 model 5 100 0.96  22.521685 22.569229  0.50573588 0.5068035 26.132866
##           r          COE       IOA
## 1 1.0000000  0.538289594 0.7691448
## 2 0.8258202  0.233921216 0.6169606
## 3 0.9371237  0.635833987 0.8179170
## 4 0.8142735 -0.009329173 0.4953354
## 5 0.9963753 -0.042044773 0.4789776
```

It is possible to rank the statistics based on the *Coefficient of Efficiency*, which is a good general indicator of model performance.

```
modStats(modData, obs = "obs", mod = "mod", type = "model", rank.name = "model")
```

```
##     model   n FAC2         MB       MGE         NMB      NMGE      RMSE
## 3 model 3 100 0.88   1.013647  7.887325  0.02276196 0.1771139  9.451354
## 1 model 1 100 0.89  10.000000 10.000000  0.22455508 0.2245551 10.000000
## 2 model 2 100 0.79   0.922391 16.592192  0.02071276 0.3725861 19.318041
## 4 model 4 100 0.56 -20.603684 21.860655 -0.46266619 0.4908921 25.758520
## 5 model 5 100 0.96  22.521685 22.569229  0.50573588 0.5068035 26.132866
##           r          COE       IOA
## 3 0.9371237  0.635833987 0.8179170
## 1 1.0000000  0.538289594 0.7691448
## 2 0.8258202  0.233921216 0.6169606
## 4 0.8142735 -0.009329173 0.4953354
## 5 0.9963753 -0.042044773 0.4789776
```

The `modStats` function is however much more flexible than indicated above. While it is useful to calculate model evaluation statistics in a straightforward way it can be much more informative to consider the statistics split by different periods.

Data have been assembled from a Defra model evaluation exercise which consists of hourly O_3 predictions at 15 receptor points around the UK for 2006. The aim here is not to identify a particular model that is 'best' and for this reason the models are simply referred to as 'model 1', 'model 2' and so on. We will aim to make the data more widely available. However, data set has this form:

```
load("~/openair/Data/modelData.RData")
head(modTest)
```

```
##         site                date o3   mod   group
## 1 Aston.Hill 2006-01-01 00:00:00 NA    NA model 1
## 2 Aston.Hill 2006-01-01 01:00:00 74 65.28 model 1
## 3 Aston.Hill 2006-01-01 02:00:00 72 64.64 model 1
## 4 Aston.Hill 2006-01-01 03:00:00 72 64.46 model 1
## 5 Aston.Hill 2006-01-01 04:00:00 70 64.88 model 1
## 6 Aston.Hill 2006-01-01 05:00:00 66 65.80 model 1
```

There are columns representing the receptor location (`site`), the date, measured values (`o3`), model predictions (`mod`) and the model itself (`group`). There are numerous ways in which the statistics can be calculated. However, of interest here is how the models perform at a single receptor by season. The seasonal nature of O_3 is a very important characteristic and it is worth considering in more detail. The statistics are easy enough to calculate as shown below. In this example a subset of the data is selected to consider only the Harwell site. Second, the `type` option is used to split the calculations by season and model. Finally the statistics are grouped by the *IOA* for each season. It is now very easy how model performance changes by season and which models perform best in each season.

```
options(digits = 2) ## don't display too many decimal places
modStats(subset(modTest, site == "Harwell"), obs = "o3", mod = "mod",
         type = c("season", "group"), rank = "group")
```

```
##          season   group    n FAC2     MB  MGE    NMB NMGE RMSE    r     COE
## 1  spring (MAM) model 1 1905 0.88   8.58 16.5  0.137 0.26   22 0.58  0.0747
## 2  spring (MAM) model 4 1905 0.88  -2.04 16.8 -0.032 0.27   22 0.45  0.0584
## 3  spring (MAM) model 3 1905 0.87  11.90 17.7  0.190 0.28   24 0.52  0.0082
## 4  spring (MAM) model 2 1905 0.88   4.98 18.1  0.079 0.29   24 0.36 -0.0151
## 5  spring (MAM) model 8 1905 0.87  10.77 19.5  0.172 0.31   24 0.59 -0.0940
## 6  spring (MAM) model 6 1825 0.82   7.79 20.0  0.123 0.32   26 0.48 -0.1181
## 7  spring (MAM) model 7 1905 0.79 -13.70 21.4 -0.218 0.34   26 0.53 -0.1950
## 8  spring (MAM) model 5 1905 0.73 -13.39 24.3 -0.213 0.39   30 0.31 -0.3619
## 9  spring (MAM) model 9 1905 0.74   1.01 24.9  0.016 0.40   33 0.26 -0.3922
## 10 summer (JJA) model 1 2002 0.92   3.83 15.5  0.063 0.26   21 0.75  0.3171
## 11 summer (JJA) model 7 2002 0.92   7.01 15.7  0.116 0.26   21 0.80  0.3082
## 12 summer (JJA) model 3 2002 0.89  14.02 18.3  0.232 0.30   24 0.80  0.1967
## 13 summer (JJA) model 2 2002 0.90   7.16 18.7  0.118 0.31   25 0.64  0.1787
## 14 summer (JJA) model 4 2002 0.90  10.01 19.0  0.165 0.31   24 0.72  0.1648
## 15 summer (JJA) model 6 1917 0.85  11.35 22.3  0.186 0.37   27 0.67  0.0274
## 16 summer (JJA) model 9 2002 0.79   1.39 24.8  0.023 0.41   34 0.30 -0.0909
## 17 summer (JJA) model 5 2002 0.80   1.78 24.8  0.029 0.41   32 0.27 -0.0920
## 18 summer (JJA) model 8 2002 0.84  25.60 27.6  0.423 0.46   32 0.81 -0.2149
## 19 autumn (SON) model 1 2172 0.87   4.67 12.7  0.095 0.26   17 0.68  0.2224
## 20 autumn (SON) model 7 2172 0.85  -5.95 13.0 -0.121 0.26   16 0.69  0.2045
## 21 autumn (SON) model 2 2172 0.88   2.61 14.2  0.053 0.29   18 0.49  0.1299
## 22 autumn (SON) model 4 2172 0.82   7.63 16.8  0.155 0.34   22 0.41 -0.0290
## 23 autumn (SON) model 6 2081 0.80  10.02 17.5  0.202 0.35   22 0.54 -0.0752
## 24 autumn (SON) model 3 2172 0.83  14.17 17.6  0.287 0.36   23 0.56 -0.0834
## 25 autumn (SON) model 5 2172 0.80   7.05 18.1  0.143 0.37   25 0.29 -0.1115
## 26 autumn (SON) model 8 2170 0.84  13.79 18.4  0.280 0.37   23 0.66 -0.1318
## 27 autumn (SON) model 9 2172 0.76   5.65 19.1  0.115 0.39   25 0.44 -0.1712
## 28 winter (DJF) model 1 1847 0.80  -1.85  9.6 -0.040 0.21   12 0.89  0.5738
## 29 winter (DJF) model 3 2117 0.77  -1.97 10.8 -0.043 0.24   14 0.85  0.5279
## 30 winter (DJF) model 8 2117 0.71  -3.01 11.1 -0.066 0.24   15 0.89  0.5143
## 31 winter (DJF) model 6 1915 0.69   0.66 14.6  0.014 0.32   20 0.75  0.3672
## 32 winter (DJF) model 2 2117 0.76  -4.88 14.6 -0.107 0.32   18 0.76  0.3621
## 33 winter (DJF) model 4 2117 0.72  -1.04 16.3 -0.023 0.36   21 0.66  0.2871
## 34 winter (DJF) model 9 2110 0.55  -7.65 18.3 -0.168 0.40   25 0.73  0.1990
## 35 winter (DJF) model 5 2117 0.66  -8.67 19.1 -0.190 0.42   24 0.57  0.1649
## 36 winter (DJF) model 7 2117 0.40 -23.14 23.5 -0.507 0.51   28 0.81 -0.0239
##      IOA
## 1   0.54
## 2   0.53
## 3   0.50
## 4   0.49
## 5   0.45
## 6   0.44
## 7   0.40
## 8   0.32
## 9   0.30
## 10  0.66
## 11  0.65
## 12  0.60
## 13  0.59
## 14  0.58
## 15  0.51
## 16  0.45
## 17  0.45
## 18  0.39
## 19  0.61
## 20  0.60
## 21  0.56
## 22  0.49
## 23  0.46
## 24  0.46
## 25  0.44
## 26  0.43
## 27  0.41
## 28  0.79
```

Note that it is possible to read the results of the `modStats` function into a data frame, which then allows the results to be plotted. This is generally a good idea when there is a lot of numeric data to consider and plots will convey the information better.

The `modStats` function is much more flexible than indicated above and can be used in lots of interesting ways. The `type` option in particular makes it possible to split the statistics in numerous ways. For example, to summarise the performance of models by site, model and day of the week:

```
modStats(modStats, obs = "o3", mod = "mod",
         type = c("site", "weekday", "group"), rank = "group")
```

Similarly, if other data are available e.g. meteorological data or other pollutant species then these variables can also be used to test models against ranges in their values. This capability is potentially very useful because it allows for a much more probing analysis into model evaluation. For example, with wind speed and direction it is easy to consider how model performance varies by wind speed intervals or wind sectors, both of which could reveal important performance characteristics.

28 Model evaluation — the TaylorDiagram function

28.1 Purpose

The *Taylor Diagram* is one of the more useful methods for evaluating model performance. Details of the diagram can be found at `http://www-pcmdi.llnl.gov/about/staff/Taylor/CV/Taylor_diagram_primer.pdf` and in Taylor (2001). 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.

We first show a diagram and then pick apart the different components to understand how to interpret it. The diagram can look overly complex but once it is understood how to interpret the three main characteristics it becomes much easier to understand. A typical diagram is shown in Figure 28.1 for nine anonymised models used for predicting hourly O_3 concentrations at 15 sites around the UK.

The plots shown in Figure 28.2 break the Taylor Diagrams into three components to aid interpretation. The first plot (top left) highlights the comparison of variability in for each model compared with the measurements. The variability is represented by the standard deviation of the observed and modelled values. The plot shows that the observed variability (given by the standard deviation) is about 27 (µg m^{-3}) and is marked as 'observed' on the x-axis. The magnitude of the variability is measured as the **radial** distance from the origin of the plot (the red line with the arrow shows the standard deviation for model *g*, which is about 25 µg m^{-3}). To aid interpretation the radial dashed line is shown from the 'observed' point. Each model is shown in this

```
TaylorDiagram(modTest, obs = "o3", mod = "mod", group = "group")
```

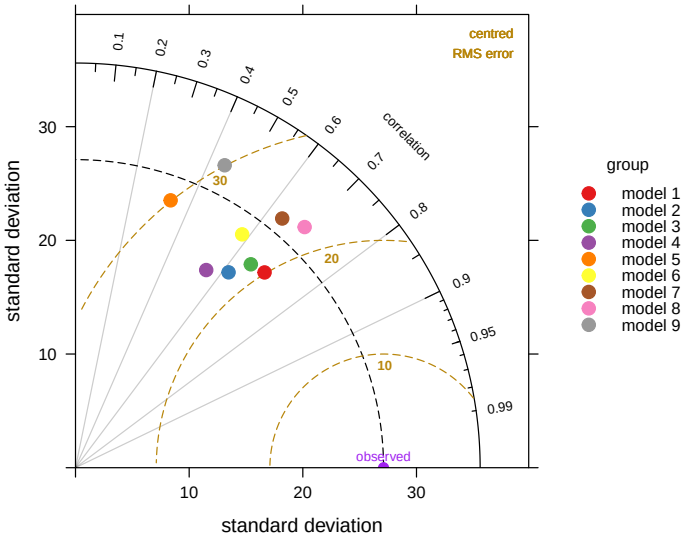


FIGURE 28.1 An example of the use of the `TaylorDiagram` function.

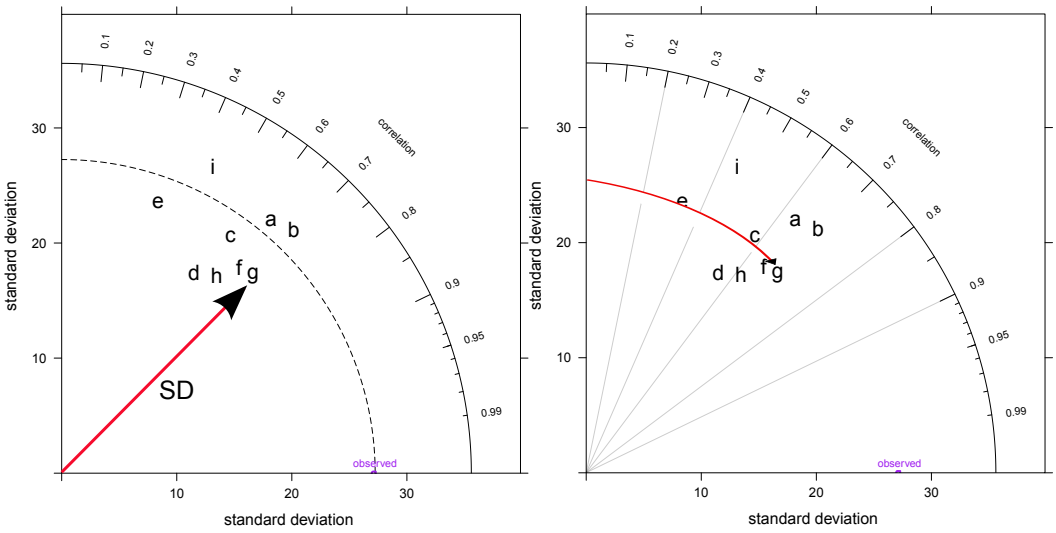
case by the position of the letters a to i. On this basis it can be seen that models 1, *a*, *b* have more variability than the measurements (because they extend beyond the dashed line), whereas the others have less variability than the measurements. Models *a* and *b* are also closed to the dashed line and therefore have the closest variability compared with the observations.

The next statistic to consider is the correlation coefficient, *R* shown by the top-right Figure in Figure 28.2. This is shown on the arc and points that lie closest to the x-axis have the highest correlation. The grey lines help to show this specific correlation coefficients. The red arc shows *R*=0.7 for model *g*. The best performing models with the highest *R* are models *b* and *g* with correlation coefficients around 0.7. Two models stand out as having much worse correlations with the observations: models *e* and *i* (values of around 0.4).

Finally, the lower plot in Figure 28.2 highlights the *centred* root-mean square error (RMS). It is centred because the mean values of the data (observations and predictions) are subtracted first. The concentric dashed lines emanating from the 'observed' point show the value of the RMS error — so points furthest from the 'observed' value are the worst performing models because they have the highest RMS errors. On this basis, model *g* has the lowest error of about 20 µg m^{-3}, shown again by the red line. Models *e* and *i* are considerably worse because they have RMS errors of around 30 µg m^{-3}.

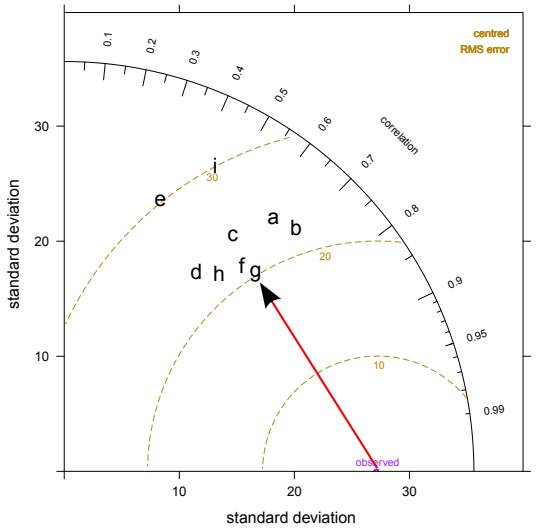
So which model is best? Taken as a whole it is probably model *g* because it has reasonably similar variability compared with the observations, the highest correlation and the least RMS error. However, models *f* and *b* also look to be good. Perhaps it is easier to conclude that models *e* and *i* are not good ....

Note that in cases where there is a column 'site' it makes sense to use `type = "site"` to ensure that the statistics are calculated on a per site basis and each panel represents a single site.



(A) Taylor Diagram highlighting the variation in standard deviation.

(B) Taylor Diagram highlighting the variation in correlation coefficient.



(C) Taylor Diagram highlighting the variation in the centred RMS error.

FIGURE 28.2 Taylor Diagrams broken down to highlight how to interpret the three main statistics. The red line/arrow indicate how to read interpret each of the three statistics.

28.2 Options available

`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`.

`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.

28.3 Example of use

The example used here carries on from the previous section using data from a Defra model evaluation exercise. As mentioned previously, the use of the `type` option offers enormous flexibility for comparing models. However, we will only focus on the seasonal evaluation of the models. In the call below, `group` is the column that identified the model and `type` is the conditioning variable that produces in this case four panels — one for each season. Note that in this case we focus on a single site.

Figure 28.3 contains a lot of useful information. Consider the summertime comparison first. All models tend to underestimate the variability of O_3 concentrations because they all lies withing the black dashed line. However, models 7 and 9 are close to the observed variability. The general underestimate of the variability for summertime conditions might reflect that the models do not adequately capture regional O_3 episodes when concentrations are high. Models 7 and 8 do best in terms of high correlation with the measurements (around 0.8) and lowest RMS error (around 20–22 $\mu g\ m^{-3}$). Models 3, 5 and 6 tend to do worse on all three statistics during the summer.

By contrast, during wintertime conditions models 1 and 3 are clearly best. From an evaluation perspective it would be useful to understand why some models are better for wintertime conditions and others better in summer and this is clearly something that could be investigated further.

There are many other useful comparisons that can be undertaken easily. A few of these are shown below, but not plotted.

```
## select a single site
LH <- subset(modTest, site == "Lullington.Heath")
TaylorDiagram(LH, obs = "o3", mod = "mod", group = "group", type = "season")
```

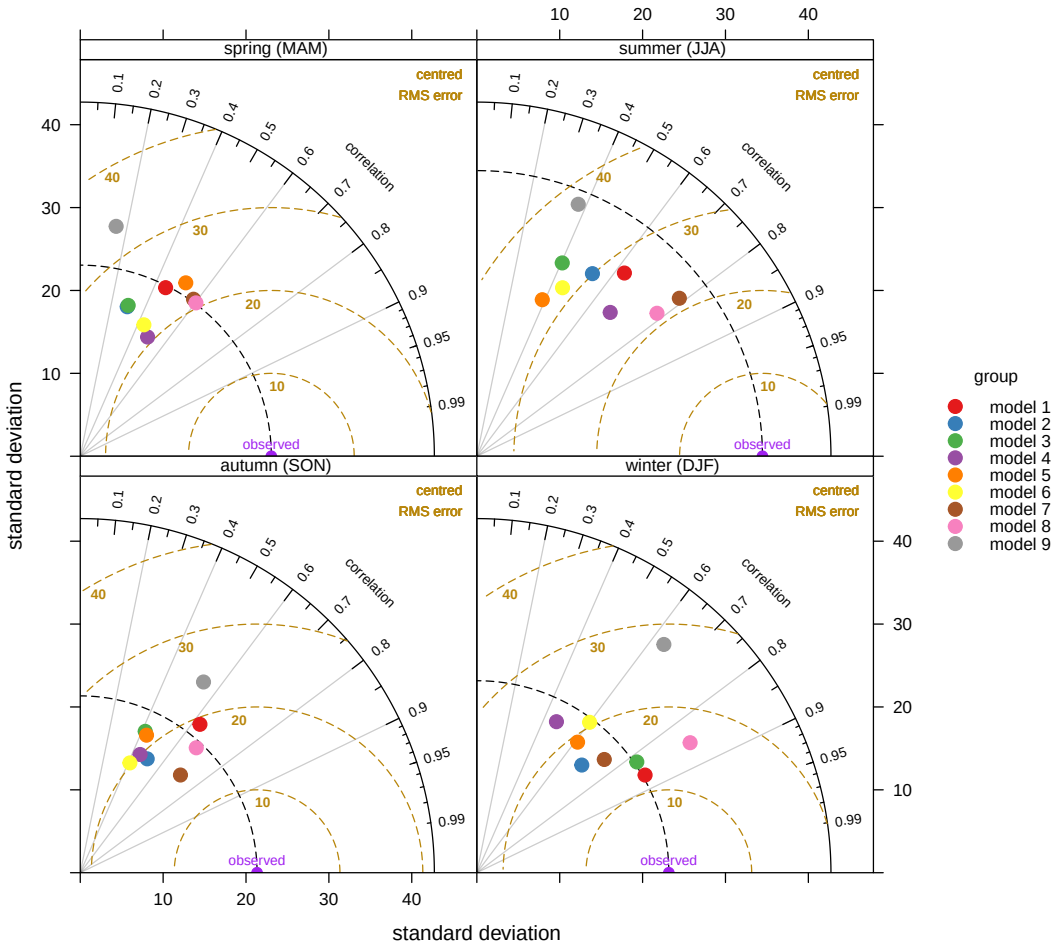


FIGURE 28.3 Use of the TaylorDiagam function to show model performance for 9 models used to predict O_3 concentrations at the Lullington Heath site.

```
## by receptor comparison
TaylorDiagram(modTest, obs = "o3", mod = "mod", group = "group", type = "site")

## by month comparison for a SINGLE site
TaylorDiagram(subset(modTest, site == "Harwell"), obs = "o3", mod = "mod",
              group = "group", type = "month")

## By season AND daylight/nighttime
TaylorDiagram(subset(modTest, site == "Harwell"), obs = "o3", mod = "mod",
              group = "group", type = c("season", "daylight"))
```

29 Model evaluation — the conditionalQuantile and conditionalEval functions

29.1 Purpose

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 µg m^{-3} 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 basic function is considerably enhanced by allowing flexible conditioning easily e.g. to evaluate model performance by season, day of the week and so on, as in other **openair** functions.

29.2 Options available

`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.

29.3 Example of use

To make things more interesting we will use data from a model evaluation exercise organised by Defra in 2010/2011. A large number of models were evaluated as part of the evaluation but we only consider hourly ozone predictions from the CMAQ model being used at King's College London.

First the data are loaded:

```
load("~/openair/Data/CMAQozone.RData")
head(CMAQ.KCL)
```

```
##         site                date o3 rollingO3Meas mod rollingO3Mod    group
## 1 Aston.Hill 2006-01-01 00:00:00 NA            NA  93           NA CMAQ.KCL
## 2 Aston.Hill 2006-01-01 01:00:00 74            NA  92           NA CMAQ.KCL
## 3 Aston.Hill 2006-01-01 02:00:00 72            NA  92           NA CMAQ.KCL
## 4 Aston.Hill 2006-01-01 03:00:00 72            NA  92           NA CMAQ.KCL
## 5 Aston.Hill 2006-01-01 04:00:00 70            NA  92           NA CMAQ.KCL
## 6 Aston.Hill 2006-01-01 05:00:00 66            NA  92           NA CMAQ.KCL
```

The data consists of hourly observations of O_3 in µg m^{-3} at 15 rural O_3 sites in the UK together with predicted values.[17] First of all we consider O_3 predictions across all sites to help illustrate the purpose of the function. The results are shown in Figure 29.1. An explanation of the Figure is given in its caption.

[17] We thank Dr Sean Beevers and Dr Nutthida Kitwiroon for access to these data.

```
conditionalQuantile(CMAQ.KCL, obs = "o3", mod = "mod")
```

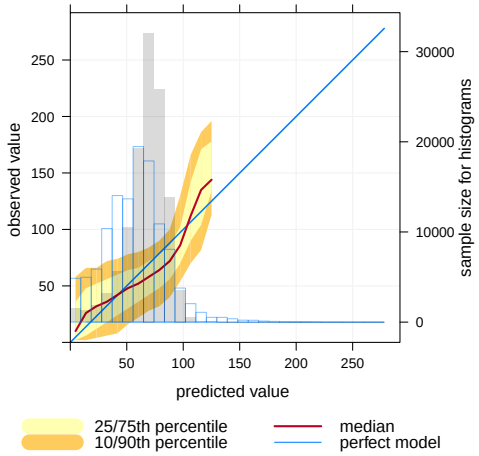


FIGURE 29.1 Example of the use of conditional quantiles applied to the KCL CMAQ model for 15 rural O_3 monitoring sites in 2006, for hourly data. The blue line shows the results for a perfect model. In this case the observations cover a range from 0 to 270 µg m^{-3}. The red line shows the median value of the predictions. The maximum predicted value is 125 µg m^{-3}, somewhat less than the maximum observed value. The shading shows the predicted quantile intervals i.e. the 25/75th and the 10/90th. A perfect model would lie on the blue line and have a very narrow spread. There is still some spread because even for a perfect model a specific quantile interval will contain a range of values. However, for the number of bins used in this plot the spread will be very narrow. Finally, the histogram shows the counts of predicted values.

A more informative analysis can be undertaken by considering conditional quantiles separately by site, which is easily done using the `type` option. The results are shown in Figure 29.2. It is now easier to see where the model performs best and how it varies by site type. For example, at a remote site in Scotland like Strath Vaich it is clear that the model does not capture either the lowest or highest O_3 concentrations very well.

As with other **openair** functions, the ability to consider conditioning can really help with interpretation. For example, what do the conditional quantiles at Lullington Heath (in south-east England) look like by season? This is easily done by subsetting the data to select that site and setting `type = "season"`, as shown in Figure 29.3. These results show that winter predictions have good coverage i.e. with width of the blue 'perfect model' line is the same as the observations. However, the predictions tend to be somewhat lower than observations for most concentrations (the median line is below the blue line) — and the width of the 10/75th and 10/90th percentiles is quite broad. However, the area where the model is less good is in summer and autumn because the predictions have low coverage (the red line only covers less than half of the observation line and the width of the percentiles is wide).

Of course it is also easy to plot by hour of the day, day of the week, by daylight/night-time and so on — easily. All these approaches can help better understand *why* a model does not perform very well rather than just quantifying its performance. Also, these types of analysis are particularly useful when more than one model is involved in a comparison as in the recent Defra model evaluation exercise, which we will come back to later when some of the results are published.

There are numerous ways in which model performance can be assessed, including the use of common statistical measures described in Section 27. These approaches are very useful for comparing models against observations and other models. How-

```
conditionalQuantile(CMAQ.KCL, obs = "o3", mod = "mod", type = "site")
```

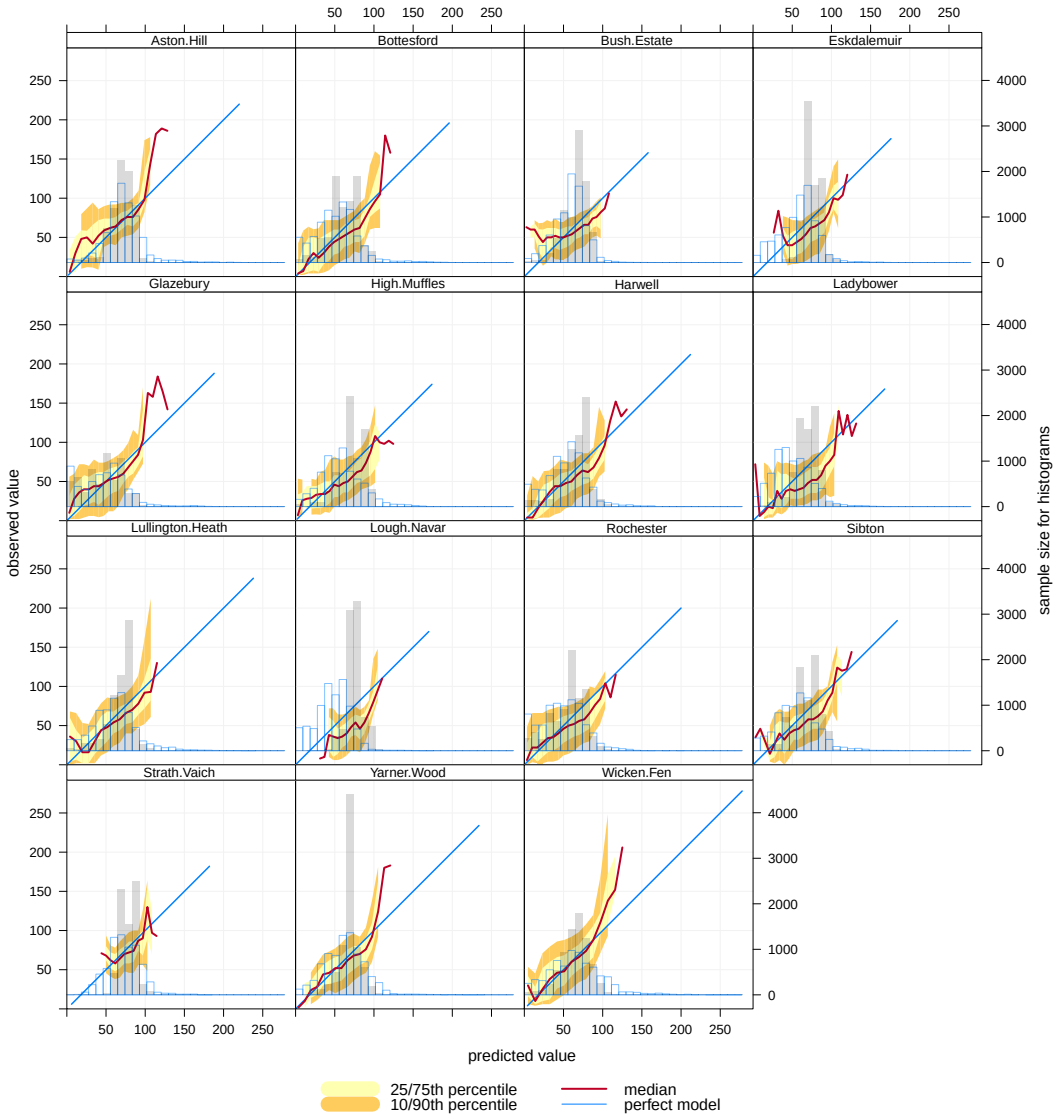


FIGURE 29.2 Conditional quantiles by site for 15 O_3 monitoring sites in the UK.

ever, model developers would generally like to know *why* a model may have poor performance under some situations. This is a much more challenging issue to address. However, useful information can be gained by considering how other variables vary simultaneously.

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 Section 11 for more details on the types of

```
conditionalQuantile(subset(CMAQ.KCL, site == "Lullington.Heath"), obs = "o3",
                    mod = "mod", type = "season")
```

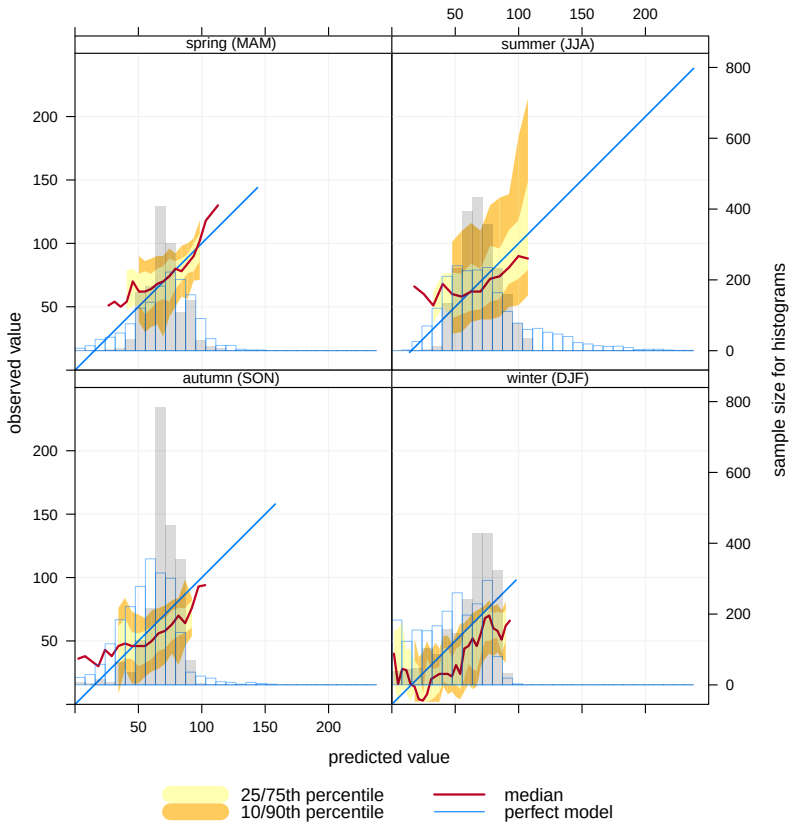


FIGURE 29.3 Conditional quantiles at Lullington Heath conditioned by season.

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. Again, this would help show whether model performance in predicting concentrations of O_3 for example is related to low to high wind speed conditions.

`conditionalEval` can also simultaneously plot the model performance of other observed/predicted variable *pairs* according to different model evaluation statistics. These statistics derive from the Section 27 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 NO_x 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 prediction intervals* of the main variable assessed in the conditional quantile plot 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 NO_x 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 NO_x 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 NO_x concentrations and lower ozone concentrations. Similarly if wind speed predictions were good and NO_x was over predicted it might suggest an over-estimate of NO_x 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.

The conditionalEval function has the following options:

mydata — A data frame containing the field obs and mod representing observed and modelled values.

obs — The name of the observations in mydata.

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

var.obs — Other variable observations for which statistics should be calculated. Can be more than length one e.g. var.obs = c("nox.obs", "ws.obs").

var.mod — Other variable predictions for which statistics should be calculated. Can be more than length one e.g. var.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.

As an example, similar data to that described above from CMAQ have been used as an example.

A subset of the data for the North Kensington site can be imported as shown below.

```
load(url("http://www.erg.kcl.ac.uk/downloads/Policy_Reports/AQdata/condDat.RData"))
```

The file contains observed and modelled hourly values for O_3, NO_x, wind speed, wind direction, temperature and relative humidity.

```
head(condDat)
```

```
##                   date O3.obs NOx.obs ws.obs wd.obs temp.obs rh.obs O3.mod
## 5  2006-01-01 00:00:00     10      29    4.6    190      4.9     89     15
## 10 2006-01-01 01:00:00     15      18     NA    210      5.1     90     17
## 15 2006-01-01 02:00:00     11      20    2.6    220      4.9     94     18
## 20 2006-01-01 03:00:00     11      19    3.6    270      5.7     91     18
## 25 2006-01-01 04:00:00     11      17    3.1    270      5.0     94     18
## 30 2006-01-01 05:00:00     12      16    3.6    260      5.8     94     15
##    NOx.mod ws.mod wd.mod temp.mod rh.mod
## 5       24    2.8    224      3.9     93
## 10      20    2.6    226      3.9     93
## 15      18    2.5    236      2.9     99
## 20      18    2.5    253      2.9     99
## 25      18    2.2    275      3.9     97
## 30      21    2.4    285      4.8     94
```

```
conditionalEval(condDat, obs = "O3.obs", mod = "O3.mod",
                statistic = "ws.obs",
                col.var = "Set3")
```

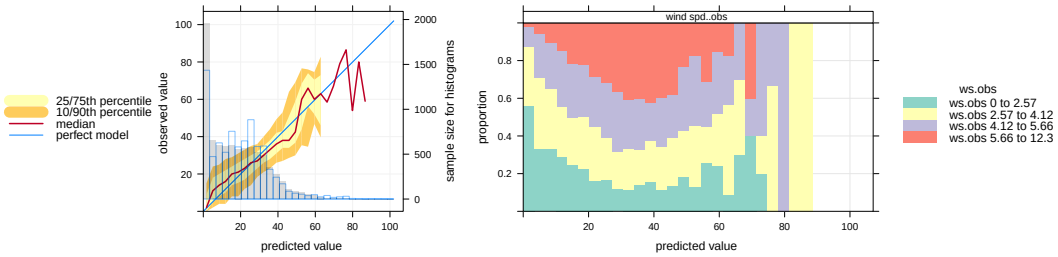


FIGURE 29.4 Conditional quantiles at for O_3 concentrations (left plot). On the right is a plot showing how the wind speed varies across the O_3 prediction intervals.

The `conditionalEval` function can be used in a straightforward way to provide information on how predictions depend on another variable in general. In this case the option `statistic` can refer to another variable in the data frame to see how the quality of predictions depend on values of that variable. For example, in Figure 29.4 it can be seen how wind speed varies across the O_3 prediction intervals. At low predicted concentrations of O_3 there is a high proportion of low wind speed conditions (0 to 2.57 m s^{-1}). When O_3 is predicted to be around 40 ppb the wind speed tends to be higher — and finally at higher predicted concentrations of O_3 the wind speed tends to decrease again. The important aspect of plotting data in this way is that it can directly relate the prediction performance to values of other variables, which should help develop a much better idea of the conditions that matter most. The user can therefore develop a good feel for the types of conditions where a model performs well or poorly and this might provide clues as to the underlying reasons for specific model behaviour.

In an extension to Figure 29.4 it is possible to derive information on the simultaneous model performance of other variables. Figure 29.5 shows the conditional quantile plot for hourly O_3 predictions. This shows among other things that concentrations of O_3 tend to be under-predicted for concentrations less than about 20 ppb. The Figure on the right shows the simultaneous *model performance* for wind speed and NO_x for the *same* prediction intervals as shown in the conditional quantile plot. The plot on the right shows that for low concentrations of predicted O_3 there is a tendency for NO_x concentrations to be overestimated (NMB≈0.2 to 0.4) and wind speeds to be underestimated (NMB≈ −0.2 to −0.3). One possible explanation for this behaviour is that the meteorological model tends to produce wind speeds that are too low, which would result in higher concentrations of NO_x, which in turn would result in lower concentrations of O_3. Note that it is possible to include more than one statistic, which would be plotted in a new panel e.g. `statistic = c("NMB", "r")`.

In essence the `conditionalEval` function provides more information on model performance that can help better diagnose potential problems. Clearly, there are many other ways in which the results can be analysed, which will depend on the data available.

A plot using temperature predictions shows that for most of the range in O_3 predictions there is very little bias in temperature (although there is some negative bias in temperature for very low concentration O_3 predictions):

```
conditionalEval(condDat, obs = "O3.obs", mod = "O3.mod",
                var.obs = c("NOx.obs", "ws.obs"),
                var.mod = c("NOx.mod", "ws.mod"),
                statistic = "NMB",
                var.names = c("nox", "wind speed"))
```

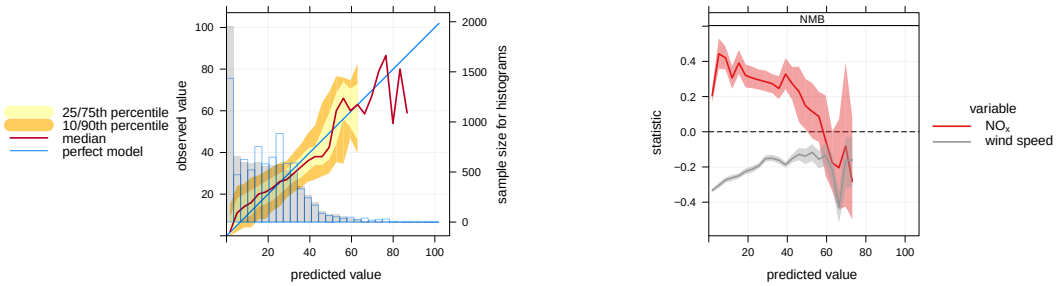


FIGURE 29.5 Conditional quantiles at for O_3 concentrations (left plot). On the right is the model performance for wind speed and NO_x predictions, which in this case is for the Normalised Mean Bias.

```
conditionalEval(condDat, obs = "O3.obs", mod = "O3.mod",
                var.obs = c("temp.obs", "ws.obs"),
                var.mod = c("temp.mod", "ws.mod"),
                statistic = "NMB", var.names = c("temperature", "wind speed"))
```

Finally (but not shown) it can be very useful to consider model performance in terms of air mass origin. In the example below, trajectories are imported, a cluster analysis undertaken and then evaluated using `conditionalEval`.

```
## import trajectories for 2006
traj <- importTraj("london", 2006)

## carry out a cluster analysis
cl <- trajCluster(traj, method = "Angle", n.cluster = 5)

## merge with orginal model eval data
condDat <- merge(condDat, cl, by = "date")

## plot it
conditionalEval(condDat, obs = "O3.obs", mod = "O3.mod",
                statistic = "cluster",
                col.var = "Set3")
```

30 The calcFno2 function—estimating primary NO_2 fractions

30.1 Purpose

see also linearRelation for oxidant slopes if NO_x, NO_2 and O_3 are available

Recent research has shown that emissions of directly emitted (primary) NO_2 from road vehicles have increased and these increases have had important effects on concentrations of NO_2 (Carslaw 2005; Carslaw and Beevers 2004; Carslaw and Carslaw 2007). Many organisations would like to quantify the level of primary NO_2 from the analysis of ambient monitoring data to help with their air quality management responsibilities. The difficulty is that this is not a straightforward thing to do — it requires some form of modelling. In some situations where NO, NO_2 and O_3 are measured, it is possible to derive an estimate of the primary NO_2 fraction by considering the gradient

in 'total oxidant' defined as NO_2 + O_3 (Clapp and Jenkin 2001).[18] However, where we most want to estimate primary NO_2 (roadside sites), O_3 is rarely measured and an alternative approach must be used. One approach is using a simple constrained chemistry model described in (Carslaw and Beevers 2005). The `calcFno2` method is based on this work but makes a few simplifying assumptions to make it easier to code.

There are several assumptions that users should be aware of when using this function. First, it is most reliable at estimating f-NO_2 when the roadside concentration is much greater than the background concentration. Second, it is best if the chosen background site is reasonably close to the roadside site and not greatly affected by local sources (which it should not be as a background site). The way the calculations work is to try and distinguish between NO_2 that is directly emitted from vehicles and that derived through the reaction between NO and O_3. During summertime periods when concentrations of NO_x are lower these two influences tend to vary linearly with NO_x, making it difficult for the method to separate them. It can often be useful to select only winter months under these situations (or at least October–March). Therefore, as a simplifying assumption, the time available for chemical reactions to take place, τ, is set to 60 seconds. We have tested the method at roadside sites where O_3 is also measured and τ = 60 s seems to provide a reasonably consistent calculation of f-NO_2.

Note that in some situations it may be worth filtering the data e.g. by wind direction to focus on the road itself. In this respect, the `polarPlot` function described on page 125 can be useful.

30.2 Options available

`input` A data frame with the following fields. `nox` and`no2` (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`.

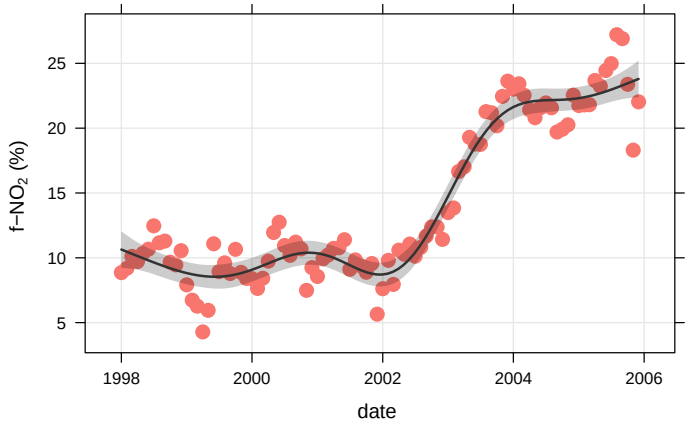
30.3 Example of use

We apply the technique to roadside data at Marylebone Road, with additional data from a nearby background site at North Kensington and appropriate meteorological variables. The data can be downloaded from the **openair** website (`http://www.openair-project.org`), and remember to change the file path below. The code run is:[19]

[18] Note that the volume fraction of NO_2 of total NO_x is termed *f-NO_2*. See the AQEG report for more information (AQEG 2008).

[19] Note that the choice to give the plot a heading is optional.

```
results <- calcFno2(fno2Data, main = "Trends in f-NO2 at Marylebone Road",
                    pch = 16, smooth = TRUE, cex = 1.5)
```




```
## the results are a list of two data frame, the first is the f-NO2 results:

head(results$data[[1]])
```

```
##              date fno2
## 1.1998 1998-01-01  8.9
## 2.1998 1998-02-01  9.2
## 3.1998 1998-03-01 10.1
## 4.1998 1998-04-01  9.7
## 5.1998 1998-05-01 10.3
## 6.1998 1998-06-01 10.7
```

```
## the second are the hourly nox, no2 and estimated o3:
head(results$data[[2]])
```

```
##                  date nox no2   o3
## 1 1998-01-01 00:00:00 285  56 0.19
## 2 1998-01-01 03:00:00 493  73 0.11
## 3 1998-01-01 04:00:00 468  70 0.35
## 4 1998-01-01 05:00:00 264  54 1.22
## 5 1998-01-01 06:00:00 171  47 0.92
## 6 1998-01-01 07:00:00 195  51 2.18
```

FIGURE 30.1 Plot from the application of the calcFno2 function applied to Marylebone Road. The plot shows a smooth fit with 95 % confidence intervals.

```
load("~/openair/Data/f-no2Data.RData")
## check first few lines of the file
head(fno2Data)
```

```
##               date nox no2 back_nox back_no2 back_o3 temp cl  wd
## 1 01/01/1998 00:00 285  39       49       34       1  3.3  2 280
## 2 01/01/1998 01:00  NA  NA       56       35       0  3.3  5 230
## 3 01/01/1998 02:00  NA  NA       41       31       4  4.4  5 190
## 4 01/01/1998 03:00 493  52       52       33       1  3.9  5 170
## 5 01/01/1998 04:00 468  78       41       30       3  3.9  5 180
## 6 01/01/1998 05:00 264  42       36       29       6  3.3  2 190
```

Now apply the function, and collect the results as shown in Figure 30.1, with additional options that are sent to scatterPlot.

Note that this is a different way to run a function compared with what has been done

```
results <- calcFno2(fno2Data, user.fno2 = 0.095, smooth = TRUE, pch = 16, cex = 1.5)
```

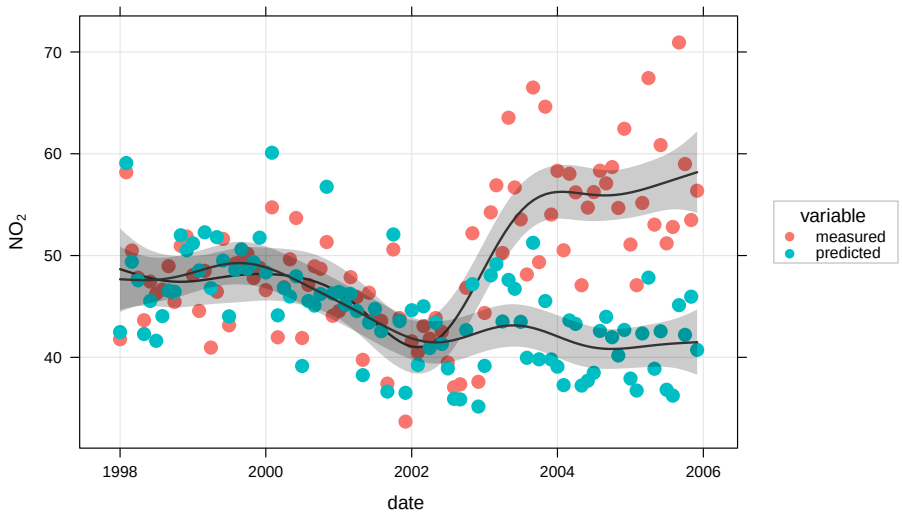


FIGURE 30.2 Plot from the application of the `calcFno2` function applied to Marylebone Road with a forced f-NO_2 values of 0.095 over the whole series. The red line and shading shows the trend in actual measurements and the blue line and shading the predicted trend in NO_2 if the f-NO_2 ratio had remained at 0.095.

previously. This time the results are read into the variable `results`, which stores the monthly mean estimated f-NO_2 values.

The results (expressed as a percentage) for f-NO_2 are then available for any other processing or plotting. The function also automatically generates a plot of monthly mean f-NO_2 values as shown in Figure 30.1. It is clear from this Figure that f-NO_2 was relatively stable at around 10 % until the end of 2002, before increasing sharply during 2003 — and remaining at about 20 % until the end of 2007. At this particular site the increases in f-NO_2 are very apparent.

An interesting question is what would NO_2 concentrations have been if f-NO_2 remained at the 10 % level, or indeed any other level. The `calcFno2` function also allows the user to input their own f-NO_2 level and make new predictions. In this case the code run is slightly different, shown in Figure 30.2.

By providing a value to the option `user.fno2` (expressed as a fraction), the function will automatically calculate NO_2 concentrations with the chosen f-NO_2 value applied to the whole time series. In this case `results` will return a data frame with dates and NO_2 concentrations. In addition a plot is produced as shown in Figure 30.2. The blue line and shading show the measured data and highlight a clear increase in NO_2 concentrations from 2003 onwards. The red line and shading shows the predicted values assuming (in this case) that f-NO_2 was constant at 0.095. Based on these results it is clear that NO_2 concentrations would have been substantially less if it were not for the recent increases in f-NO_2.

31 Utility functions

31.1 Selecting data by date

Selecting by date/time in R can be intimidating for new users—and time consuming for all users. The `selectByDate` function aims to make this easier by allowing users

to select data based on the British way of expressing date i.e. d/m/y. This function should be very useful in circumstances where it is necessary to select only part of a data frame.

The function has the following options.

`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.

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

##                    date  ws  wd nox no2 o3 pm10 so2   co pm25         split.by
## 8761 1999-01-01 00:00:00 5.0 140  88  35  4   21 3.8 1.02   18 before Jan. 2003
## 8762 1999-01-01 01:00:00 4.1 160 132  41  3   17 5.2 2.70   11 before Jan. 2003
## 8763 1999-01-01 02:00:00 4.8 160 168  40  4   17 6.5 2.87    8 before Jan. 2003
## 8764 1999-01-01 03:00:00 4.9 150  85  36  3   15 4.2 1.62   10 before Jan. 2003
## 8765 1999-01-01 04:00:00 4.7 150  93  37  3   16 4.2 1.02   11 before Jan. 2003
## 8766 1999-01-01 05:00:00 4.0 160  74  29  5   14 3.9 0.72   NA before Jan. 2003
##       feature ratio
## 8761 easterly 0.044
## 8762 easterly 0.040
## 8763 easterly 0.039
## 8764 easterly 0.049
## 8765 easterly 0.046
## 8766    other 0.052

tail(data.1999)

##                     date  ws  wd nox no2 o3 pm10 so2  co pm25         split.by
## 17515 1999-12-31 18:00:00 4.7 190 226  39 NA   29 5.5 2.4   23 before Jan. 2003
## 17516 1999-12-31 19:00:00 4.0 180 202  37 NA   27 4.8 2.1   23 before Jan. 2003
## 17517 1999-12-31 20:00:00 3.4 190 246  44 NA   30 5.9 2.5   23 before Jan. 2003
## 17518 1999-12-31 21:00:00 3.7 220 231  35 NA   28 5.3 2.2   23 before Jan. 2003
## 17519 1999-12-31 22:00:00 4.1 200 217  41 NA   31 4.8 2.2   26 before Jan. 2003
## 17520 1999-12-31 23:00:00 3.2 200 181  37 NA   28 3.5 1.8   22 before Jan. 2003
##       feature ratio
## 17515   other 0.024
## 17516   other 0.024
## 17517   other 0.024
## 17518   other 0.023
## 17519   other 0.022
## 17520   other 0.019

## 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"))
```

The function can be used directly in other functions. For example, to make a polar plot using year 2000 data:

```
polarPlot(selectByDate(mydata, year = 2000), pollutant = "so2")
```

31.2 Selecting run lengths of values above a threshold — pollution episodes

A seemingly easy thing to do that has relevance to air pollution episodes is to select run lengths of contiguous values of a pollutant above a certain threshold. For example, one might be interested in selecting O_3 concentrations where there are at least 8 consecutive hours above 90 ppb. In other words, a selection that combines both a threshold and *persistence*. These periods can be very important from a health perspective and it can be useful to study the conditions under which they occur. But

how do you select such periods easily? The `selectRunning` utility function has been written to do this. It could be useful for all sorts of situations e.g.

- Selecting hours where primary pollutant concentrations are persistently high — and then applying other **openair** functions to analyse the data in more depth.
- In the study of particle suspension or deposition etc. it might be useful to select hours where wind speeds remain high or rainfall persists for several hours to see how these conditions affect particle concentrations.
- It could be useful in health impact studies to select blocks of data where pollutant concentrations remain above a certain threshold.

The `selectRunning` has the following options:

`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.

As an example we are going to consider O_3 concentrations at a semi-rural site in south-west London (Teddington). The data can be downloaded as follows:

```
ted <- importKCL(site = "td0", year = 2005:2009, met = TRUE)
## see how many rows there are
nrow(ted)
```

We are going to contrast two polar plots of O_3 concentration. The first uses all hours in the data set, and the second uses a subset of hours. The subset of hours is defined by O_3 concentrations above 90 ppb for periods of at least 8-hours i.e. what might be considered as ozone episode conditions.

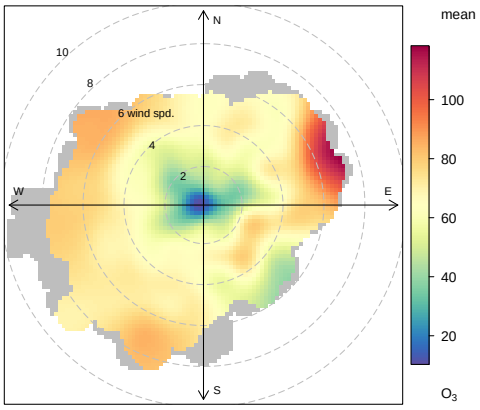
```
episode <- selectRunning(ted, pollutant = "o3", threshold = 90, run.len = 8)
## see how many rows there are
nrow(episode)

## [1] 1399
```

Now we are going to produce two bivariate polar plots shown in Figure 31.1.

The results are shown in Figure 31.1. The polar plot for all data (left plot of Figure 31.1) shows that the highest O_3 concentrations tend to occur for high wind speed conditions from almost every direction. Lower concentrations are observed for low wind speeds because concentrations of NO_x are higher, resulting in O_3 destruction. By contrast, a polar plot of the episode conditions (right plot of Figure 31.1) is very different. In this case there is a clear set of conditions where these criteria are met i.e. lengths of at least 8-hours where the O_3 concentration is at least 90 ppb. It is clear the highest concentrations are dominated by south-easterly conditions i.e. corresponding to easterly flow from continental Europe where there has been time to the O_3 chemistry to take place.

```
polarPlot(ted, pollutant = "o3", min.bin = 2)
```




```
polarPlot(episode, pollutant = "o3", min.bin = 2)
```

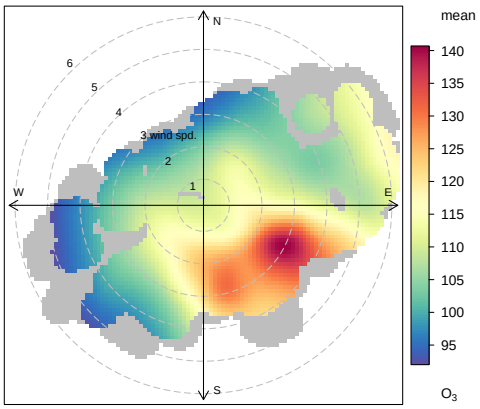


FIGURE 31.1 Example of using the `selectRunning` function to select episode hours to produce bivariate polar plots of O_3 concentration.

Another interesting test plot is to consider NO_x concentrations at Marylebone Road — see Figure 15.1, which shows that high concentrations are dominated by a swathe of south-westerly wind conditions (even for high wind speeds). However, if a selection is made of episode conditions (defined here as NO_x concentrations >500 ppb for at least 5-hours), then it can be seen that it is actually the low wind speed conditions that dominate. These conditions correspond to low in-canyon wind speeds and low wind speeds across London, which tend to elevate local and background NO_x concentrations. Even though high concentrations of NO_x are observed at high wind speeds, it does not seem that these conditions are as important for episode conditions. Users can run the code below to verify these observations.

```
episode <- selectRunning(mydata, pollutant = "nox", threshold = 800, run.len = 5)
polarPlot(episode, pollutant = "nox", min.bin = 2)
```

31.3 Calculating rolling means

Some air pollution statistics such as for O_3 and particulate matter are expressed as rolling means and it is useful to be able to calculate these. It can also be useful to help smooth-out data for clearer plotting. The `rollingMean` function makes these

calculations. One detail that can be important is that for some statistics a mean is only considered valid if there are a sufficient number of valid readings over the averaging period. Often there is a requirement for at least 75 % data capture. For example, with an averaging period of 8 hours and a data capture threshold of 75%, at least 6 hours are required to calculate the mean.

The function is called as follows; in this case to calculate 8-hour rolling mean concentrations of O_3.

```
data(mydata)
mydata <- rollingMean(mydata, pollutant = "o3", hours = 8,
                      new.name = "rollingo3", data.thresh = 75)
tail(mydata)
```

```
##                     date  ws  wd nox no2 o3 pm10 so2  co pm25 rollingo3
## 65528 2005-06-23 07:00:00 1.5 250 404 156  4   49  NA 1.8   28       6.9
## 65529 2005-06-23 08:00:00 1.5 260 388 145  6   48  NA 1.6   26       8.5
## 65530 2005-06-23 09:00:00 1.5 210 404 168  7   58  NA 1.3   34      12.2
## 65531 2005-06-23 10:00:00 2.6 240 387 175 10   55  NA 1.3   34      14.0
## 65532 2005-06-23 11:00:00 3.1 220 312 125 15   52  NA 1.3   33        NA
## 65533 2005-06-23 12:00:00 3.1 220 287 119 17   55  NA 1.3   35        NA
```

Note that calculating rolling means shortens the length of the data set. In the case of O_3, no calculations are made for the last 7 hours.

Type `help(rollingMean)` into R for more details. Note that the function currently only works with a single site.

31.4 Aggregating data by different time intervals

Aggregating data by different averaging periods is a common and important task. There are many reasons for aggregating data in this way:

1. Data sets may have different averaging periods and there is a need to combine them. For example, the task of combining an hourly air quality data set with a 15-minute average meteorological data set. The need here would be to aggregate the 15-minute data to 1-hour before merging.

2. It is extremely useful to consider data with different averaging times in a straightforward way. Plotting a very long time series of hourly or higher resolution data can hide the main features and it would be useful to apply a specific (but flexible) averaging period to the data for plotting.

3. Those who make measurements during field campaigns (particularly for academic research) may have many instruments with a range of different time resolutions. It can be useful to re-calculate time series with a common averaging period; or maybe help reduce noise.

4. It is useful to calculate statistics other than means when aggregating e.g. percentile values, maximums etc.

5. For statistical analysis there can be short-term autocorrelation present. Being able to choose a longer averaging period is sometimes a useful strategy for minimising autocorrelation.

In aggregating data in this way, there are a couple of other issues that can be useful to deal with at the same time. First, the calculation of proper vector-averaged wind direction is essential. Second, sometimes it is useful to set a minimum number of data

points that must be present before the averaging is done. For example, in calculating monthly averages, it may be unwise to not account for data capture if some months only have a few valid points.

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. 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.

All these issues are (hopefully) dealt with by the `timeAverage` function. The options are shown below, but as ever it is best to check the help that comes with the **openair** package.

see also timePlot for plotting with different averaging times and statistics

The `timeAverage` function has the following options:

`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 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"`. 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`.

```
## load in fresh version of mydata
data(mydata)
```

To calculate daily means from hourly (or higher resolution) data:

```
daily <- timeAverage(mydata, avg.time = "day")
head(daily)
```

```
##         date   ws  wd nox no2  o3 pm10 so2  co pm25
## 1 1998-01-01  6.8 188 154  39 6.9   18 3.2 2.7  NaN
## 2 1998-01-02  7.1 223 132  39 6.5   28 3.9 1.8  NaN
## 3 1998-01-03 11.0 226 120  38 8.4   20 3.2 1.7  NaN
## 4 1998-01-04 11.5 223 105  35 9.6   21 3.0 1.6  NaN
## 5 1998-01-05  6.6 237 175  46 5.0   24 4.5 2.1  NaN
## 6 1998-01-06  4.4 197 214  45 1.3   35 5.7 2.5  NaN
```

Monthly 95th percentile values:

```
monthly <- timeAverage(mydata, avg.time = "month", statistic = "percentile",
                       percentile = 95)
head(monthly)
```

```
##         date   ws wd nox no2 o3 pm10 so2  co pm25
## 1 1998-01-01 11.2 45 371  69 14   53  11 4.0   NA
## 2 1998-02-01  8.2 17 524  92  7   69  17 5.6   NA
## 3 1998-03-01 10.6 38 417  85 15   61  18 4.9   NA
## 4 1998-04-01  8.2 44 384  82 20   52  15 4.2   NA
## 5 1998-05-01  7.6 41 300  80 25   61  13 3.6   40
## 6 1998-06-01  8.5 51 377  74 15   53  12 4.3   34
```

2-week averages but only calculate if at least 75% of the data are available:

```
twoweek <- timeAverage(mydata, avg.time = "2 week", data.thresh = 75)
```

```
## [1] "Input data time interval assumed is 3600 sec"
```

```
head(twoweek)
```

```
##         date  ws  wd nox no2  o3 pm10 so2  co pm25
## 1 1997-12-29 7.0 212 167  41 4.6   29 4.5 2.2   NA
## 2 1998-01-12 4.9 221 173  42 4.7   29 5.1 1.9   NA
## 3 1998-01-26 2.8 242 233  51 2.3   35 8.1 2.4   NA
## 4 1998-02-09 4.4 215 276  57 2.6   44 9.0 2.9   NA
## 5 1998-02-23 6.9 237 248  57 5.0   29 9.8 2.6   NA
## 6 1998-03-09 3.0 288 160  45 5.6   33 8.6 1.6   NA
```

`timeAverage` also works the other way in that it can be used to derive higher temporal resolution data e.g. hourly from daily data or 15-minute from hourly data. An example of usage would be the combining of daily mean particle data with hourly meteorological data. There are two ways these two data sets can be combined: either average the meteorological data to daily means or calculate hourly means from the particle data. The `timeAverage` function when used to 'expand' data in this way will repeat the original values the number of times required to fill the new time scale. In the example below we calculate 15-minute data from hourly data. As it can be seen, the first line is repeated four times and so on.

```
data15 <- timeAverage(mydata, avg.time = "15 min", fill = TRUE)
head(data15, 20)
```

```
##                  date  ws  wd nox no2 o3 pm10 so2   co pm25
## 1  1998-01-01 00:00:00 0.6 280 285  39  1   29 4.7  3.4   NA
## 2  1998-01-01 00:15:00 0.6 280 285  39  1   29 4.7  3.4   NA
## 3  1998-01-01 00:30:00 0.6 280 285  39  1   29 4.7  3.4   NA
## 4  1998-01-01 00:45:00 0.6 280 285  39  1   29 4.7  3.4   NA
## 5  1998-01-01 01:00:00 2.2 230  NA  NA NA   37  NA   NA   NA
## 6  1998-01-01 01:15:00 2.2 230  NA  NA NA   37  NA   NA   NA
## 7  1998-01-01 01:30:00 2.2 230  NA  NA NA   37  NA   NA   NA
## 8  1998-01-01 01:45:00 2.2 230  NA  NA NA   37  NA   NA   NA
## 9  1998-01-01 02:00:00 2.8 190  NA  NA  3   34 6.8  9.6   NA
## 10 1998-01-01 02:15:00 2.8 190  NA  NA  3   34 6.8  9.6   NA
## 11 1998-01-01 02:30:00 2.8 190  NA  NA  3   34 6.8  9.6   NA
## 12 1998-01-01 02:45:00 2.8 190  NA  NA  3   34 6.8  9.6   NA
## 13 1998-01-01 03:00:00 2.2 170 493  52  3   35 7.7 10.2   NA
## 14 1998-01-01 03:15:00 2.2 170 493  52  3   35 7.7 10.2   NA
## 15 1998-01-01 03:30:00 2.2 170 493  52  3   35 7.7 10.2   NA
## 16 1998-01-01 03:45:00 2.2 170 493  52  3   35 7.7 10.2   NA
## 17 1998-01-01 04:00:00 2.4 180 468  78  2   34 8.1  8.9   NA
## 18 1998-01-01 04:15:00 2.4 180 468  78  2   34 8.1  8.9   NA
## 19 1998-01-01 04:30:00 2.4 180 468  78  2   34 8.1  8.9   NA
## 20 1998-01-01 04:45:00 2.4 180 468  78  2   34 8.1  8.9   NA
```

The `timePlot` can apply this function directly to make it very easy to plot data with different averaging times and statistics.

31.5 Calculating percentiles

`calcPercentile` makes it straightforward to calculate percentiles for a single pollutant. It can take account of different averaging periods, data capture thresholds — see Section 31.4 for more details. The function has the following options:

`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.

For example, to calculate the 25, 50, 75 and 95th percentiles of O_3 concentration by year:

```
calcPercentile(mydata, pollutant  = "o3", percentile = c(25, 50, 75, 95),
               avg.time = "year")
```

```
##         date percentile.25 percentile.50 percentile.75 percentile.95
## 1 1998-01-01             2             4             7            16
## 2 1999-01-01             2             4             9            21
## 3 2000-01-01             2             4             9            22
## 4 2001-01-01             2             4            10            24
## 5 2002-01-01             2             4            10            24
## 6 2003-01-01             2             4            11            24
## 7 2004-01-01             2             5            11            23
## 8 2005-01-01             3             7            16            28
```

31.6 The `corPlot` function — correlation matrices

Understanding how different variables are related to one another is always important. However, it can be difficult to easily develop an understanding of the relationships when many different variables are present. One of the useful techniques used is to plot a *correlation matrix*, which provides the correlation between all pairs of data. The basic idea of a correlation matrix has been extended to help visualise relationships between variables by Friendly (2002) and Sarkar (2007).

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 — imagine a 'fuzz' of points with no relationship between them.

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`.)

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.

The `corPlot` function has the following options:

`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.

`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"`.

`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.

An example of the `corPlot` function is shown in Figure 31.2. In this Figure it can be seen the highest correlation coefficient is between PM_{10} and $PM_{2.5}$ (r = 0. 84) and that the correlations between SO_2, NO_2 and NO_x are also high. O_3 has a negative correlation with most pollutants, which is expected due to the reaction between NO and O_3. It is not that apparent in Figure 31.2 that the order the variables appear is due to their similarity with one another, through hierarchical cluster analysis. In this case we have chosen to also plot a *dendrogram* that appears on the right of the plot. Dendrograms provide additional information to help with visualising how groups of variables are related to one another. Note that dendrograms can only be plotted for `type = "default"` i.e. for a single panel plot.

Note also that the `corPlot` accepts a `type` option, so it possible to condition the data in many flexible ways, although this may become difficult to visualise with too many panels. For example:

```
corPlot(mydata, type = "season")
```

When there are a very large number of variables present, the `corPlot` is a very effective way of quickly gaining an idea of how variables are related. As an example (not plotted) it is useful to consider the hydrocarbons measured at Marylebone Road. There is a lot of information in the hydrocarbon plot (about 40 species), but due to the hierarchical clustering it is possible to see that isoprene, ethane and propane behave differently to most of the other hydrocarbons. This is because they have different (non-vehicle exhaust) origins. Ethane and propane results from natural gas leakage whereas isoprene is biogenic in origin (although some is from vehicle exhaust too). It is also worth considering how the relationships change between the species over the years as hydrocarbon emissions are increasingly controlled, or maybe the difference between summer and winter blends of fuels and so on.

```
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)
```

```
corPlot(mydata, dendrogram = TRUE)
```

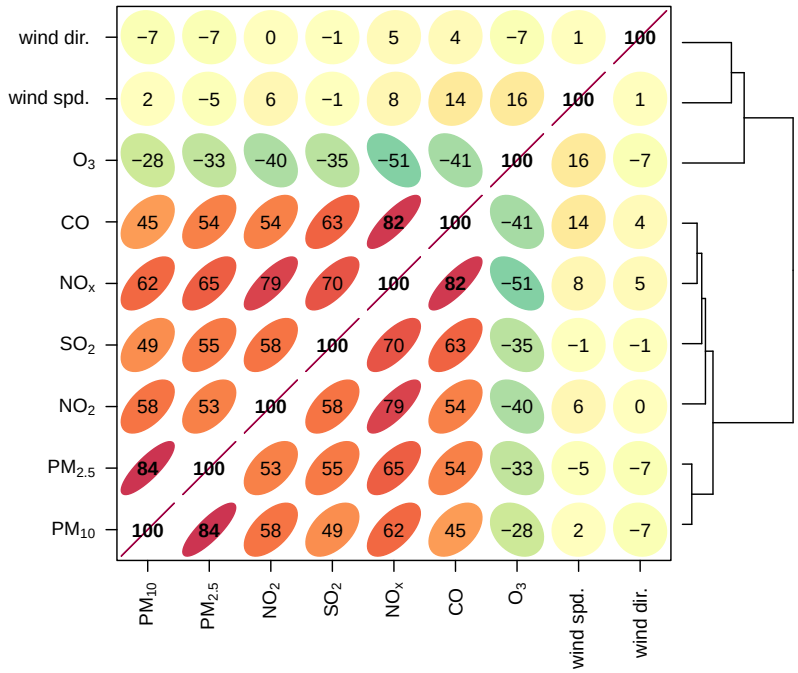


FIGURE 31.2 Example of a correlation matrix showing the relationships between variables.

31.7 Preparing data to compare sites, for model evaluation and intervention analysis

Many of the functions described have the potential to be extremely flexible. Mention has already been made of how to compare different sites in some of the functions. It was stated that the data had to be in a certain format for the functions to work. This section describes a few simple functions to do this — and more.

31.7.1 Intervention analysis

Another common scenario is that there is interest in showing plots by different time intervals *on the same scale*. There could be all sorts of reasons for wanting to do this. A classic example would be to show a before/after plot due to some intervention such as a low emission zone. Again, the function below exploits the flexible 'site' option available in many functions.

A small helper function `splitByDate` has been written to simplify chopping up a data set into different defined periods. The function takes three arguments: a data frame to process, a date (or dates) and labels for each period. If there was interest in looking at `mydata` before and after the 1st Jan 2003, it would be split into *two* periods (before that date and after). In other words, there will always be one more label than there is date. We have made the function easier to use for supplying dates. Dates can be accepted in the form 'dd/mm/yyyy' e.g. 13/04/1999 or as 'yyyy-mm-dd' e.g. '1999-04-13'.

The example below chops the data set up into three sections, called 'before', 'during' and 'after'. This is done to show how more than one date can be supplied to the function.

```
polarAnnulus(mydata, pollutant = "no2", type = "split.by", period = "hour",
             layout = c(3, 1))
```

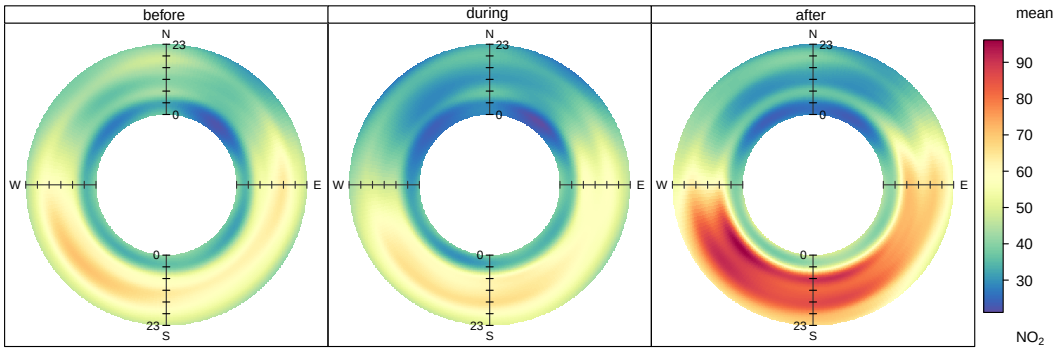


FIGURE 31.3 Example of processing data for use in the `polarAnnulus` function by time period for NO_2 concentrations at Marylebone Road.

```
mydata <- splitByDate(mydata, dates = c("1/1/2000", "1/3/2003"),
                      labels = c("before", "during", "after"))
head(mydata)
```

```
##                  date  ws  wd nox no2 o3 pm10 so2   co pm25 split.by
## 1 1998-01-01 00:00:00 0.6 280 285  39  1   29 4.7  3.4   NA   before
## 2 1998-01-01 01:00:00 2.2 230  NA  NA NA   37  NA   NA   NA   before
## 3 1998-01-01 02:00:00 2.8 190  NA  NA  3   34 6.8  9.6   NA   before
## 4 1998-01-01 03:00:00 2.2 170 493  52  3   35 7.7 10.2   NA   before
## 5 1998-01-01 04:00:00 2.4 180 468  78  2   34 8.1  8.9   NA   before
## 6 1998-01-01 05:00:00 3.0 190 264  42  0   16 5.5  3.1   NA   before
```

```
tail(mydata)
```

```
##                      date  ws  wd nox no2 o3 pm10 so2  co pm25 split.by
## 65528 2005-06-23 07:00:00 1.5 250 404 156  4   49  NA 1.8   28    after
## 65529 2005-06-23 08:00:00 1.5 260 388 145  6   48  NA 1.6   26    after
## 65530 2005-06-23 09:00:00 1.5 210 404 168  7   58  NA 1.3   34    after
## 65531 2005-06-23 10:00:00 2.6 240 387 175 10   55  NA 1.3   34    after
## 65532 2005-06-23 11:00:00 3.1 220 312 125 15   52  NA 1.3   33    after
## 65533 2005-06-23 12:00:00 3.1 220 287 119 17   55  NA 1.3   35    after
```

As can be seen, there is a new field `split.by` (although the name can be set by the user), where at the beginning of the time series it is labelled 'before' and at the end it is labelled 'after'. Now let us make a polar annulus plot showing the diurnal variation of NO_2 by wind direction:

In some cases it would make sense to have labels that refer to dates. Here is an example:

```
mydata <- splitByDate(mydata, dates = c("1/1/2000", "1/3/2003"),
                      labels = c("before Jan. 2000", "Jan. 2000 - Mar. 2003",
                      "after Mar. 2003"))
```

31.7.2 Combining lots of sites

A typical example is that imported data have a date field and one or more pollutant fields from one of more sites in a series of columns. The aim would be, for example, to produce a series of plots by site for the same pollutant. If the data contains multiple

pollutants and multiple sites, it makes sense to subset the data first.[20] For example, if a data frame **mydata** has fields 'date', 'nox.site1', 'so2.site1', 'nox.site2', 'so2.site2', then just working with the NO_x data can be done by:

```
subdata <- subset(mydata, select = c(date, nox.site1, nox.site2))
```

Rather than import new data, the code below first makes an artificial data set from which to work. In a real situation, the first few lines would not be needed.

```
## load reshape2 package if it is not already loaded...
library(reshape2)
## first make a subset of the data: date and nox, ws, wd
siteData <- subset(mydata, select = c(date, ws, wd, nox))

## look at the first few lines
head(siteData)

##                  date  ws  wd nox
## 1 1998-01-01 00:00:00 0.6 280 285
## 2 1998-01-01 01:00:00 2.2 230  NA
## 3 1998-01-01 02:00:00 2.8 190  NA
## 4 1998-01-01 03:00:00 2.2 170 493
## 5 1998-01-01 04:00:00 2.4 180 468
## 6 1998-01-01 05:00:00 3.0 190 264

## rename the nox field to "site1"
names(siteData)[4] <- "site1"

## now make another field "site2" to be equal to half of site1
siteData$site2 <- siteData$site1 * 0.5
## end of making new data, now let's process it
head(siteData)

##                  date  ws  wd site1 site2
## 1 1998-01-01 00:00:00 0.6 280   285   142
## 2 1998-01-01 01:00:00 2.2 230    NA    NA
## 3 1998-01-01 02:00:00 2.8 190    NA    NA
## 4 1998-01-01 03:00:00 2.2 170   493   246
## 5 1998-01-01 04:00:00 2.4 180   468   234
## 6 1998-01-01 05:00:00 3.0 190   264   132

## now we need to "stack" the data, ready for openair functions
## use the melt function in the reshape2 package (loaded with openair)
siteData <- melt(siteData, measure.vars = c("site1", "site2"))
head(siteData)

##                  date  ws  wd variable value
## 1 1998-01-01 00:00:00 0.6 280    site1   285
## 2 1998-01-01 01:00:00 2.2 230    site1    NA
## 3 1998-01-01 02:00:00 2.8 190    site1    NA
## 4 1998-01-01 03:00:00 2.2 170    site1   493
## 5 1998-01-01 04:00:00 2.4 180    site1   468
## 6 1998-01-01 05:00:00 3.0 190    site1   264

## change the variable names (one of them has to be "site")
names(siteData)[4:5] <- c("site", "nox")
```

Now it is possible to run many **openair** functions on this dataset. In this case, let

[20] Note that if you are able to use data from the AURN archive using the `importAURN` function, the data will already be in the correct format for direct use by many of the functions—although it may well be necessary to merge some meteorological data first.

```
polarPlot(siteData, pollutant = "nox", type = "site")
```

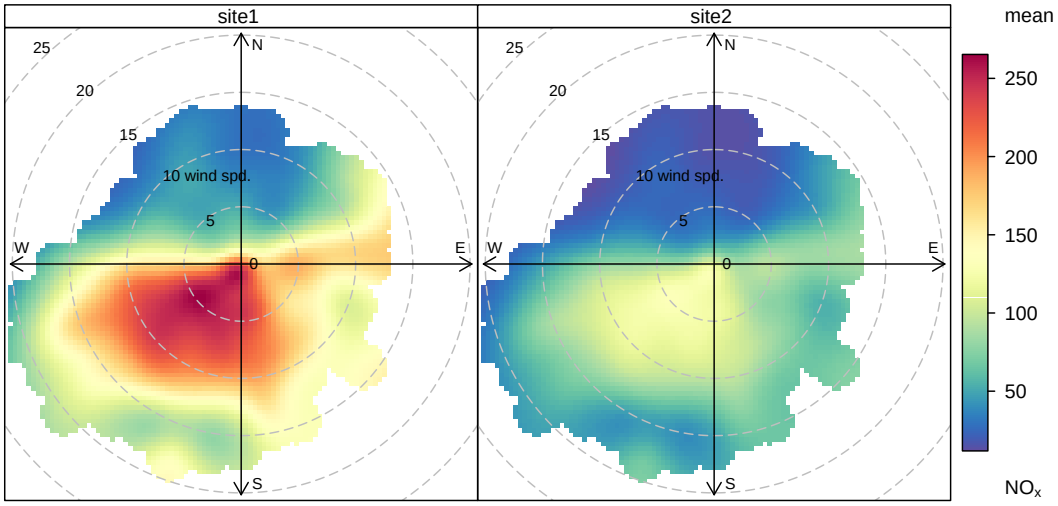


FIGURE 31.4 Example of processing data for use in the `polarPlot` function by site.

us consider a `polarPlot`. Note that this process would work with many more sites than shown here. Note, however, many functions such as `polarPlot` accept multiple pollutants and the `importAURN` and `importKCL` format multiple site data directly and no additional work is required by the user.

Acknowledgements

We are very grateful to a range of organisations that have so far seen this initiative as something worth supporting financially. These include:

- Dr Karl Ropkins of the Institute for Transport Studies at the University of Leeds for his contributions during the NERC project.
- The initial funding supplied by the Faculty of Environment at the University of Leeds to help develop a course on R to analyse and understand air pollution data.
- Sefton Council for their direct funding of data analysis in their borough as part of the Beacon air quality scheme.
- AEA.
- North Lincolnshire Council.
- Defra as part of their work through the AURN.
- The Natural Environment Research Council (NERC) Knowledge Transfer grant NE/G001081/1.

This work would not be possible without the incredible individuals who have given their time freely to develop the R system. This includes those in the R-Core Development Team and all those that contribute to its development (R Core Team 2013).

Further information

For any enquiries related to this document or the **openair** package, please use the contact details below. Please contact us regarding any of the following: bug reports, suggestions for existing functions, suggestions for new functions and offer of code contributions. When reporting potential bugs, it is helpful (sometimes essential) to submit a reproducible example, which would normally require sending a description of the problem and the data set used. Also, we are interested in developing further funded case studies.

David Carslaw
King's College London
Environmental Research Group
Franklin Wilkins Building
150 Stamford Street
London
SE1 9NH
UK

e-mail: mailto:david.carslaw@kcl.ac.uk

References

Applequist, S. (2012). "Wind Rose Bias Correction". In: *Journal of Applied Meteorology and Climatology* 51.7, pp. 1305–1309 (cit. on pp. 104, 108).

AQEG (2008). *Trends in primary nitrogen dioxide in the UK*. Air Quality Expert Group. Report prepared by the Air Quality Expert Group for the Department for Environment, Food, Rural Affairs; Scottish Executive; Welsh Assembly Government; and Department of the Environment in Northern Ireland. (cit. on p. 252).

Ara Begum, B., E. Kim, C.-H. Jeong, D.-W. Lee and P. K. Hopke (2005). "Evaluation of the potential source contribution function using the 2002 Quebec forest fire episode". In: *Atmospheric Environment* 39.20, pp. 3719–3724. DOI: 10.1016/j.atmosenv.2005.03.008 (cit. on p. 222).

Ashbaugh, L. L., W. C. Malm and W. Z. Sadeh (1985). "A residence time probability analysis of sulfur concentrations at grand Canyon National Park". In: *Atmospheric Environment (1967)* 19.8, pp. 1263–1270. DOI: 10.1016/0004-6981(85)90256-2 (cit. on pp. 116, 135).

Carslaw, D. C. (2005). "Evidence of an increasing NO_2/NO_x, emissions ratio from road traffic emissions". In: *Atmospheric Environment* 39.26, pp. 4793–4802 (cit. on p. 251).

Carslaw, D. C. and S. D. Beevers (2004). "Investigating the potential importance of primary NO_2 emissions in a street canyon". In: *Atmospheric Environment* 38.22, pp. 3585–3594 (cit. on p. 251).

– (2005). "Estimations of road vehicle primary NO_2 exhaust emission fractions using monitoring data in London". In: *Atmospheric Environment* 39.1, pp. 167–177 (cit. on p. 252).

Carslaw, D. C., S. D. Beevers, K. Ropkins 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". In: *Atmospheric Environment* 40.28, pp. 5424–5434 (cit. on p. 125).

Carslaw, D. C., S. D. Beevers and J. E. Tate (2007). "Modelling and assessing trends in traffic-related emissions using a generalised additive modelling approach". In: *Atmospheric Environment* 41.26, pp. 5289–5299 (cit. on p. 170).

Carslaw, D. C. and N. Carslaw (2007). "Detecting and characterising small changes in urban nitrogen dioxide concentrations". In: *Atmospheric Environment* 41.22, pp. 4723–4733 (cit. on p. 251).

Carslaw, D. C. and S. D. Beevers (2013). "Characterising and understanding emission sources using bivariate polar plots and k-means clustering". In: *Environmental Modelling & Software* 40, pp. 325–329. doi: 10.1016/j.envsoft.2012.09.005 (cit. on pp. 125, 138).

Carslaw, D. C. and K. Ropkins (2012). "openair — An R package for air quality data analysis". In: *Environmental Modelling & Software* 27–28, pp. 52–61. doi: 10.1016/j.envsoft.2011.09.008 (cit. on p. 8).

Chambers, J. M. (2007). *Software for Data Analysis: Programming with R*. ISBN 978-0-387-75935-7. New York: Springer (cit. on p. 9).

Chatfield, C. (2004). *The analysis of time series : an introduction / Chris Chatfield*. 6th ed. Boca Raton, FL ; London : Chapman & Hall/CRC (cit. on p. 279).

Clapp, L. J. and M. E. Jenkin (2001). "Analysis of the relationship between ambient levels of O_3, NO_2 and NO as a function of NO_X in the UK". In: *Atmospheric Environment* 35.36, pp. 6391–6405 (cit. on p. 252).

Cleveland, W. S. (1985). *The elements of graphing data*. Wadsworth Publ. Co. Belmont, CA, USA (cit. on p. 58).

Cleveland, W. (1993). *Visualizing Data*. Hobart Press, Summit, NJ (cit. on p. 58).

COMEAP (2011). *Review of the UK Air Quality Index: A report by the Committee on the Medical Effects of Air Pollutants* (cit. on p. 161).

Crawley, M. (2007). *The R book*. Wiley (cit. on p. 13).

Dalgaard, P. (2008). *Introductory Statistics with R*. 2nd. ISBN 978-0-387-79053-4. Springer, p. 380 (cit. on p. 13).

Davison, A. C. and D. Hinkley (1997). *Bootstrap methods and their application*. (Anthony Christopher). Cambridge ; New York, NY, USA : Cambridge University Press (cit. on p. 275).

Droppo, J. G. and B. A. Napier (2008). "Wind direction bias in generating wind roses and conducting sector-based air dispersion modeling". In: *Journal of the Air & Waste Management Association* 58.7, pp. 913–918 (cit. on p. 104).

Efron, B. and R. Tibshirani (1993). *An Introduction to the Bootstrap*. Chapman & Hall (cit. on p. 275).

Fleming, Z. L., P. S. Monks and A. J. Manning (2012). "Review: Untangling the influence of air-mass history in interpreting observed atmospheric composition". In: *Atmospheric Research* 104-105, pp. 1–39. doi: 10.1016/j.atmosres.2011.09.009 (cit. on pp. 220, 222).

Friendly, M. (2002). "Corrgrams: Exploratory Displays for Correlation Matrices". In: *The American Statistician* 56.4, pp. 316–325 (cit. on p. 264).

Hastie, T. J. and R. Tibshirani (1990). *Generalized additive models*. London: Chapman and Hall (cit. on p. 126).

Helsel, D. and R. Hirsch (2002). *Statistical Methods in Water Resources*. US Geological Survey (cit. on p. 14).

Henry, R., G. A. Norris, R. Vedantham and J. R. Turner (2009). "Source Region Identification Using Kernel Smoothing". In: *Environmental Science & Technology* 43.11, 4090–4097. doi: {10.1021/es8011723} (cit. on p. 108).

Hirsch, R. M., J. R. Slack and R. A. Smith (1982). "Techniques Of Trend Analysis For Monthly Water-Quality Data". In: *Water Resources Research* 18.1. ISI Document Delivery No.: NC504, pp. 107–121 (cit. on p. 162).

Hsu, Y.-K., T. M. Holsen and P. K. Hopke (2003). "Comparison of hybrid receptor models to locate PCB sources in Chicago". In: *Atmospheric Environment* 37.4, pp. 545–562. doi: 10.1016/S1352-2310(02)00886-5 (cit. on pp. 223, 224).

Kunsch, H. R. (1989). "The jackknife and the bootstrap for general stationary observations". In: *Annals of Statistics* 17.3, pp. 1217–1241 (cit. on pp. 163, 276).

Legates, D. R. and G. J. McCabe Jr (1999). "Evaluating the use of "goodness-of-fit" measures in hydrologic and hydroclimatic model validation". In: *Water Resources Research* 35.1, pp. 233–241 (cit. on p. 231).

Legates, D. R. and G. J. McCabe (2012). "A refined index of model performance: a rejoinder". In: *International Journal of Climatology* (cit. on p. 231).

Lupu, A. and W. Maenhaut (2002). "Application and comparison of two statistical trajectory techniques for identification of source regions of atmospheric aerosol species". In: *Atmospheric Environment* 36, pp. 5607–5618 (cit. on p. 224).

Maindonald, J. and J. Braun (2007). *Data Analysis and Graphics Using R: An Example-based Approach*. Cambridge University Press (cit. on p. 13).

Matloff, N. (2011). *The Art of R Programming*. No Starch Press (cit. on p. 14).

McHugh, C. A., D. J. Carruthers and H. A. Edmunds (1997). "ADMS and ADMS-Urban". In: *International Journal of Environment and Pollution* 8.3-6, pp. 438–440 (cit. on p. 70).

Monks, P. S. (2000). "A review of the observations and origins of the spring ozone maximum". In: *Atmospheric Environment* 34.21. ISI Document Delivery No.: 333VG, pp. 3545–3561 (cit. on p. 207).

Pekney, N. J., C. I. Davidson, L. Zhou and P. K. Hopke (2006). "Application of PSCF and CPF to PMF-Modeled Sources of PM 2.5 in Pittsburgh". In: *Aerosol Science and Technology* 40.10, pp. 952–961. doi: 10.1080/02786820500543324 (cit. on p. 222).

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria (cit. on p. 269).

Sarkar, D. (2007). *Lattice Multivariate Data Visualization with R*. ISBN 978-0-387-75968-5. New York: Springer (cit. on pp. 13, 264).

Seibert, P., H. Kromp-Kolb, U. Baltensperger and D. Jost (1994). "Trajectory analysis of high-alpine air pollution data". In: *NATO Challenges of Modern Society* 18, pp. 595–595 (cit. on p. 223).

Sen, P. K. (1968). "Estimates of regression coefficient based on Kendall's tau". In: *Journal of the American Statistical Association* 63(324) (cit. on p. 162).

Spector, P. (2008). *Data Manipulation with R*. ISBN 978-0-387-74730-9. New York: Springer (cit. on p. 13).

Taylor, K. E. (2001). "Summarizing multiple aspects of model performance in a single diagram". In: *Journal of Geophysical Research* 106.D7, pp. 7183–7192 (cit. on p. 237).

Theil, H. (1950). "A rank invariant method of linear and polynomial regression analysis, I, II, III". In: *Proceedings of the Koninklijke Nederlandse Akademie Wetenschappen, Series A – Mathematical Sciences* 53, pp. 386–392, 521–525, 1397–1412 (cit. on p. 162).

Tukey, J. (1977). *Exploratory data analysis*. Addison-Wesley Series in Behavioral Science: Quantitative Methods, Reading, Mass.: Addison-Wesley, 1977 (cit. on p. 10).

Uria-Tellaetxe, I. and D. C. Carslaw (2014). "Conditional bivariate probability function for source identification". In: *Environmental Modelling & Software* 59, pp. 1–9. doi: 10.1016/j.envsoft.2014.05.002 (cit. on pp. 125, 135, 136).

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". In: *Atmospheric Environment* 41.39, pp. 9195–9205 (cit. on p. 125).

Wilcox, R. R. (2010). *Fundamentals of Modern Statistical Methods: Substantially Improving Power and Accuracy*. 2nd. Springer New York (cit. on p. 162).

Wilks, D. S. (2005). *Statistical Methods in the Atmospheric Sciences, Volume 91, Second Edition (International Geophysics)*. 2nd ed. Academic Press (cit. on p. 243).

Willmott, C. J., S. M. Robeson and K. Matsuura (2011). "A refined index of model performance". In: *International Journal of Climatology* (cit. on p. 232).

Wood, S. N. (2006). *Generalized Additive Models: An Introduction with R*. Chapman and Hall/CRC (cit. on pp. 126, 278).

Xie, Y. (2013a). *Dynamic Documents with R and knitr*. ISBN 978-1482203530. Chapman and Hall/CRC (cit. on p. 10).

– (2013b). *knitr: A general-purpose package for dynamic report generation in R*. R package version 1.1 (cit. on p. 10).

Yu, K., Y. Cheung, T. Cheung and R. Henry (2004). "Identifying the impact of large urban airports on local air quality by nonparametric regression". In: *Atmospheric Environment* 38.27, pp. 4501–4507 (cit. on p. 125).

A Installing and maintaining R

A.1 Downloading and installing R

R can be downloaded from `http://www.r-project.org`, shown in Figure A.1. On the left hand side there is a link to the *Comprehensive R Archive Network* (CRAN), which provides links to local repositories where the software can be downloaded from. The one most relevant to the UK is hosted by the University of Bristol. Follow the link and choose to download 'Precompiled binary distributions' from the Download and Install R section. Most likely you will want the Windows version, but versions for Linux and Apple Mac are also available. This will provide an executable file that can be downloaded (something like R-2.12.0-win.exe). A direct link from the UK to the Windows download file is `http://www.stats.bris.ac.uk/R/bin/windows/base/`

Important Information — Internet connections for Windows users

Note that R set up and maintenance works best when users have direct Internet access. Many users that use R through their organisation's computers may need to install R slightly differently due to the proxy settings used. Rather than accept all the defaults during installation, at the Setup screen choose not to accept the defaults and when offered, choose 'Internet2' as the Internet option. This will force R to use the same proxy settings used by Internet Explorer. The defaults for all other options can be accepted.

The other issue on Windows systems is Administrator rights. When it comes to installing or updating packages it may be necessary to 'Run as Administrator'. To do this, go to R on the Windows menu and right-click, then choose to 'Run as administrator'.

The installation of R is straightforward. Most users can happily accept all the defaults. However, see box for installation information for users installing R on an organisation's computer system. Choose where you want to install it (the default is usually appropriate for most systems). It is a good idea to install it where you have write privileges — probably more of an issue for Windows Vista/7 users. Many more details are provided on the R website regarding installation on specific systems.

...then accept all the defaults unless you have a preference for an alternative option.

A.2 Maintenance

One of the advantages of R compared with most commercially available software is that it is regularly updated; fixing any bugs and adding new features. New versions tend to be released every six months and it is wise to keep the version as up to date as possible. You can check the website occasionally, or sign up to be alerted to new releases (recommended). Often, as new versions of the base system are released, certain R packages will also require updating. If there is incompatibility between the base system and an R package you will likely receive a warning message. To keep the packages up to date select Packages | Update packages ..., which will prompt you to choose a CRAN mirror (choose the UK one again). This will check to see if all you packages are up to date, and if not, automatically download and install more recent versions. Note the information about R in the box where it may be necessary to run R as an administrator.

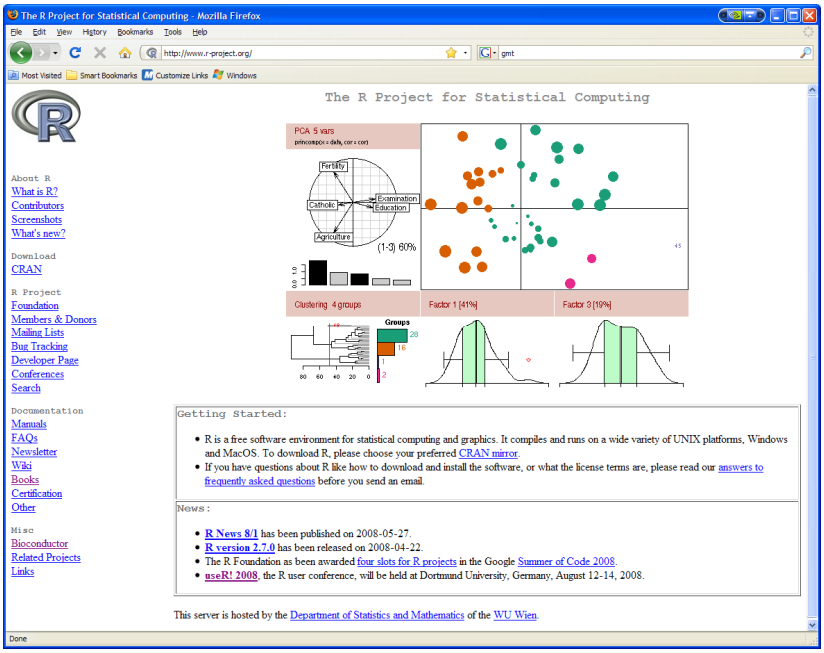


FIGURE A.1 The R-project web pages.

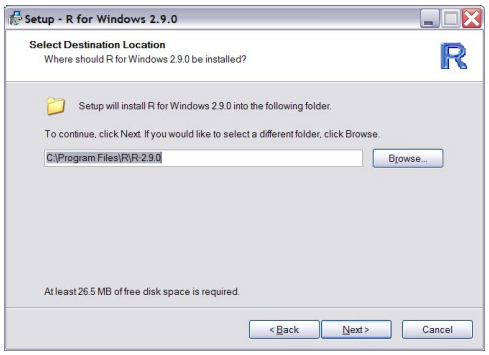


FIGURE A.2 Choose where R should be installed.

B Bootstrap estimates of uncertainty

B.1 The bootstrap

The *bootstrap* is a data-based simulation method for analysing data, including hypothesis testing, standard error and confidence interval estimation. It involves repeatedly drawing random samples from the original data, with replacement (see EFRON and TIBSHIRANI (1993) and DAVISON and HINKLEY (1997) for a detailed history and examples of use of the bootstrap). Each *bootstrap sample* is the same size as the original sample. The 'with replacement' bit is important. Sampling with replacement means that after we randomly draw an observation from the original sample we put it back before drawing the next observation i.e. it is possible to draw the same sample more than once. In fact, on average, 37 % of data will not be sampled each time. If one sampled without replacement it would be equivalent to just shuffling the data and no new information is available. Typically, 100s or 1000s of samples are required in order to derive reliable statistics.

The term bootstrap derives from the phrase "to pull oneself up by one's bootstraps". The phrase is based on one of the eighteenth century *Adventures of Baron Mun-*

chausen by Rudolph Erich Raspe. The Baron had fallen to the bottom of a deep lake. Just when it looked like all was lost, he thought to pick himself up by his own bootstraps! In a statistical sense it is meant to convey the idea of generating 'new' data from the original data set itself, which seems like an implausible thing to do, but has been shown to be valid.

When the bootstrap was discovered in the 1970s it was difficult to apply to many practical problems because computers were not powerful enough to carry out such repetitive and intensive calculations. However, computers are now sufficiently powerful to allow these methods to be used (in most circumstances) easily. This section does not aim to provide an in-depth consideration of statistics and justify the use of these methods, but rather aims to provide some background in their use in **openair**.

When used to estimate confidence intervals, the bootstrap sampling will yield, say, 1000 estimates of the statistic of interest e.g. the slope of a trend. This distribution could be highly skewed and this is one of the principal advantages of the technique: normality is not required. We now have a 1000 bootstrap samples, and 1000 estimates of the statistic of interest, one from each bootstrap sample. If these 1000 bootstrap samples are ordered in increasing value, a bootstrap 95 % confidence interval for the mean would be from the 25th to the 975th largest values. Sometimes, uncertainty estimates are not symmetrical. For example, it may not be possible to report an uncertainty as 100 $\pm$ 12, but 100 [87, 121], where 87 and 121 are the lower and upper 95 % confidence intervals, respectfully.

B.2 The block bootstrap

The basic bootstrap assumes that data are independent. However, in time series this is rarely the case due to *autocorrelation* when consecutive points in time are related to one another. For example, for data with a strong seasonal effect, the month of January may tend to have higher values than other months. These effects can, however, be difficult to characterise and model. The motivation for accounting for autocorrelation in this project is mostly to ensure that uncertainty estimates in trends and other statistics are not overly optimistic, which would generally be the case if autocorrelation was not accounted for. These effects can be accounted for by ensuring that the random sampling captures the correlation structure of the data using a *block bootstrap* (Kunsch 1989). The idea is that if data (or residuals from a model) are sampled in small blocks, the correlation structure is retained, provided there is not significant correlation between the blocks.

The following Figures highlight the importance of accounting for autocorrelation using the `TheilSen` function. Figure B.1 shows a simulated time series comprising of a linear trend + some random noise with no autocorrelation. The Sen-Theil slope and the slope uncertainties are shown (and the slope statistics upper left). By contrast, Figure B.2 shows a similar series but with a high autocorrelation value of 0.8. In this time series it is possible to see that the values seem to fall and rise in 'chunks', indicative of autocorrelation. In this plot, no account has been taken of autocorrelation and the uncertainty in the slope is very similar to Figure B.1.

However, if autocorrelation is accounted for using the data shown in Figure B.2, the uncertainty in the slope increases markedly, as shown in Figure B.3. Instead of the 95 % confidence intervals ranging from 0.42–0.54, they now range from 0.36–0.61 — approximately double the uncertainty of the case where no account is taken of autocorrelation.

The block bootstrap has also been applied to models e.g. the Generalized Additive Model (GAM) used in the `smoothTrend` function. There are two options here: the observations can be sampled and the models run many times (called *case resampling*),

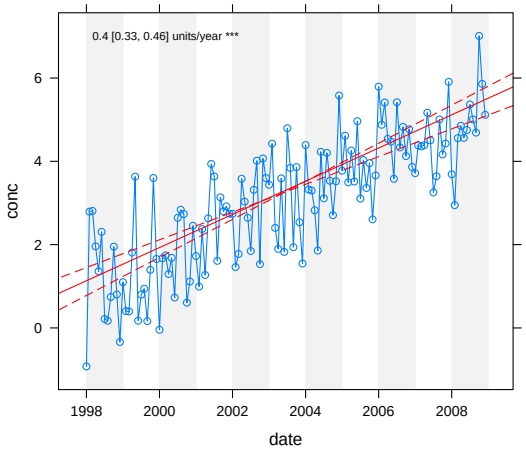


FIGURE B.1 AR1 time random series where the autocorrelation coefficient is zero.

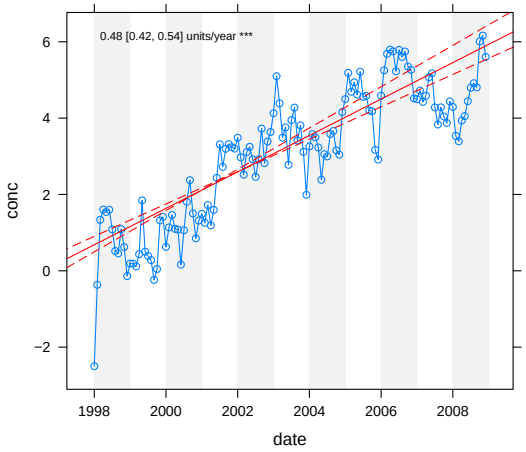


FIGURE B.2 AR1 time random series where the autocorrelation coefficient is 0.8. The uncertainty in the Sen-Theil slope estimate does not account for autocorrelation.

or the residuals from the model can be sampled and added to the model predictions to make 'new' input data and run many times (called *residual resampling*). There are pros and cons with each approach, but often the two methods yield similar results. In the case of a GAM (or specifically the *mgcv* package), which uses cross-validation for model fitting, having duplicate samples through bootstrapping would seem to make little sense. The approach adopted here therefore is to use residual resampling. The effect of taking account of autocorrelation often (but not always) is an increase in the predicted uncertainty intervals, and a smooth function that is less 'wiggly' than that derived by not accounting for autocorrelation.

A more 'robust' approach is outlined in Appendix C, where models are used to describe the correlation structure of the data.

Clearly, the importance of these issues is data-dependent and there are other issues to consider too. However, if one is interested in drawing important inferences from data, then it would seem wise to account for these effects. It should be noted that these issues are an area of active research and will be revisited from time to time with the aim of improving the robustness of the techniques used.

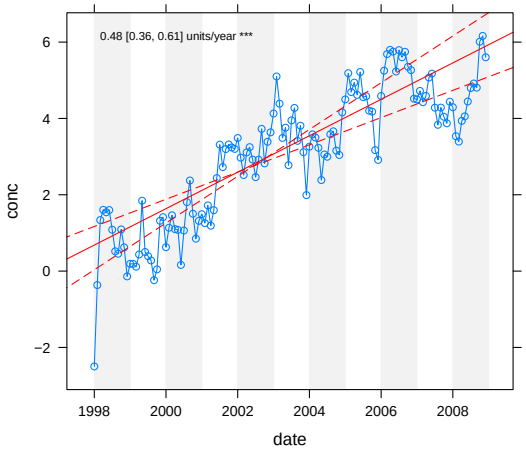


FIGURE B.3 AR1 time random series where the autocorrelation coefficient is 0.8. The uncertainty in the Sen-Theil slope estimate does account for autocorrelation.

C A closer look at trends

Understanding trends is a core component of air quality and the atmospheric sciences in general. **openair** provides two main functions for considering trends (`smoothTrend` and `TheilSen`), the latter useful for linear trend estimates. Understanding trends and quantifying them robustly is not so easy and careful analysis would treat each time series individually and consider a wide range of diagnostics. In this section we take advantage of some of the excellent capabilities that R has to consider fitting trend models. Experience with real atmospheric composition data shows that trends are rarely linear, which is unfortunate given how much of statistics has been built around the linear model.

Generalized Additive Models (GAMs) offer a flexible approach to calculating trends and in particular, the `mgcv` package contains many functions that are very useful for such modelling. Some of the details of this type of model are presented in WOOD (2006) and the `mgcv` package itself.

The example considered is 23 years of O_3 measurements at Mace Head on the West coast of Ireland. The example shows the sorts of steps that might be taken to build a model to explain the trend. The data are first imported and then the year, month and 'trend' estimated. Note that 'trend' here is simply a decimal date that can be used to construct various explanatory models.

First we import the data:

```
## Loading required package: nlme
## This is mgcv 1.8-4. For overview type 'help("mgcv-package")'.
```

```
library(mgcv)
dat <- importAURN(site = "mh", year = 1988:2010)
```

```
## calculate monthly means
monthly <- timeAverage(dat, avg.time = "month")
## now calculate components for the models
monthly$year <- as.numeric(format(monthly$date, "%Y"))
monthly$month <- as.numeric(format(monthly$date, "%m"))
monthly <- transform(monthly, trend = year + (month - 1) / 12)
```

```
timePlot(monthly, pollutant = "o3")
```

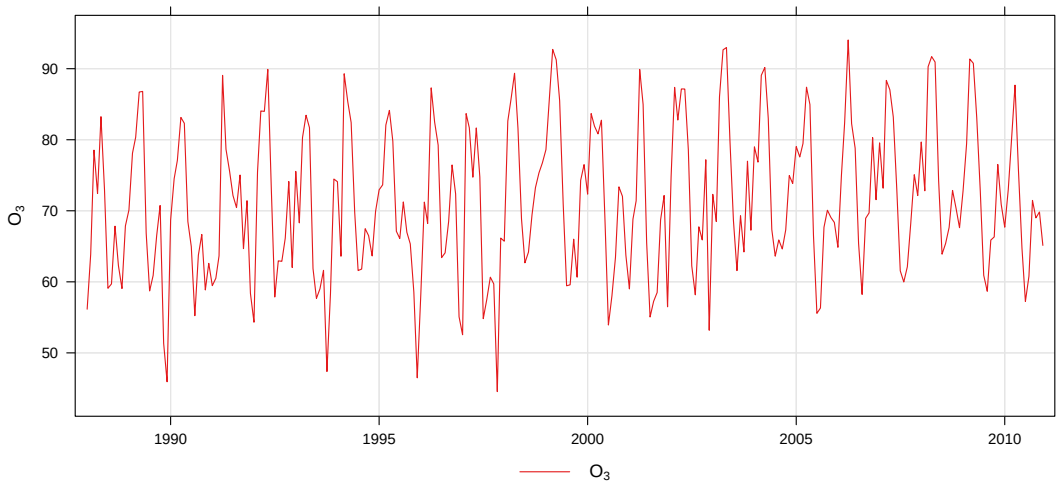


FIGURE C.1 Monthly mean O_3 concentrations at Mace Head, Ireland (1998–2010).

It is *always* a good idea to plot the data first:

Figure C.1 shows that there is a clear seasonal variation in O_3 concentrations, which is certainly expected. Less obvious is whether there is a trend.

Even though it is known there is a seasonal signal in the data, we will first of all ignore it and build a simple model that only has a trend component (model M0).

```
M0 <- gam(o3 ~ s(trend), data = monthly)
summary(M0)

##
## Family: gaussian
## Link function: identity
##
## Formula:
## o3 ~ s(trend)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    71.34       0.62     115   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##          edf Ref.df    F p-value
## s(trend)   1      1 6.96  0.0088 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.0212   Deviance explained = 2.48%
## GCV = 106.82  Scale est. = 106.04    n = 276
```

This model only explains about 2% of the variation as shown by the adjusted r^2. More of a problem however is that no account has been taken of the seasonal variation. An easy way of seeing the effect of this omission is to plot the autocorrelation function (ACF) of the residuals, shown in Figure C.2. This Figure clearly shows the residuals have a strong seasonal pattern. CHATFIELD (2004) provides lots of useful information

```
acf(residuals(M0))
```

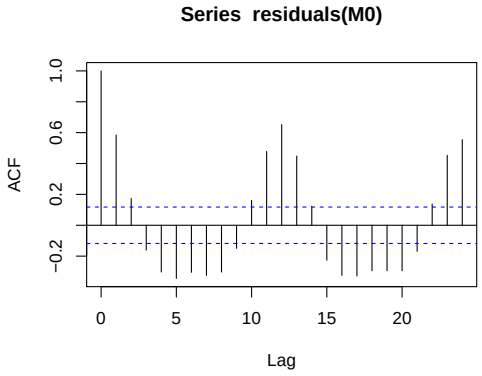


FIGURE C.2 ACF for the residuals of model M0.

on time series modelling.

A refined model should therefore take account of the seasonal variation in O_3 concentrations. Therefore, we add a term taking account of the seasonal variation. Note also that we choose a cyclic spline for the monthly component (`bs = "cc"`), which joins the first and last points i.e. January and December.

```
M1 <- gam(o3 ~ s(trend) + s(month, bs = "cc"), data = monthly)
summary(M1)

##
## Family: gaussian
## Link function: identity
##
## Formula:
## o3 ~ s(trend) + s(month, bs = "cc")
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   71.343      0.374     191   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##           edf Ref.df    F p-value
## s(trend) 1.22    1.4 15.8 1.7e-05 ***
## s(month) 6.11    8.0 59.4 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.644   Deviance explained = 65.4%
## GCV = 39.766  Scale est. = 38.566    n = 276
```

Now we have a model that explains much more of the variation with an r^2 of 0.65. Also, the p-values for the trend and seasonal components are both highly statistically significant. Let's have a look at the separate components for trend and seasonal variation:

The seasonal component shown in Figure C.4 clearly shows the strong seasonal effect on O_3 at this site (peaking in April). The trend component is actually linear in this case and could be modelled as such. This model looks much better, but as is often the case autocorrelation could remain important. The ACF is shown in Figure C.5 and

```
plot.gam(M1, select = 1, shade = TRUE)
```

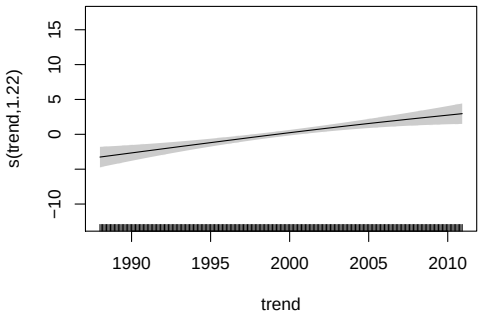


FIGURE C.3 The trend component of model M1.

```
plot.gam(M1, select = 2, shade = TRUE)
```

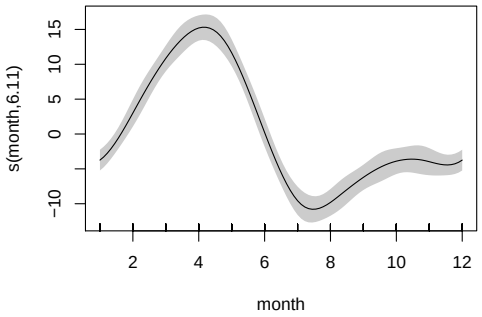


FIGURE C.4 The seasonal component of model M1.

shows there is still some short-term correlation in the residuals.

Note also that there are other diagnostic tests one should consider when comparing these models that are not shown here e.g. such as considering the normality of the residuals. Indeed a consideration of the residuals shows that the model fails to some extent in explaining the very low values of O_3, which can be seen in Figure C.1. These few points (which skew the residuals) may well be associated with air masses from the polluted regions of Europe. Better and more useful models would likely be possible if the data were split my airmass origin, which is something that will be returned to when **openair** includes a consideration of back trajectories.

Further tests, also considering the partial autocorrelation function (PACF) suggest that an AR1 model is suitable for modelling this short-term autocorrelation. This is where modelling using a GAMM (Generalized Additive Mixed Model) comes in because it is possible to model the short-term autocorrelation using a linear mixed model. The `gamm` function uses the package `nlme` and the Generalized Linear Mixed Model (GLMM) fitting routine. In the M2 model below the correlation structure is considered explicitly.

```
acf(residuals(M1))
```

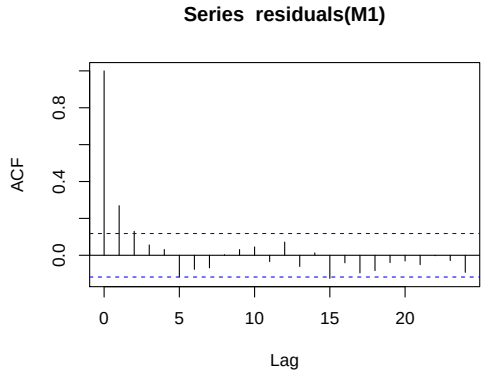


FIGURE C.5 ACF for the residuals of model M1.

```
M2 <- gamm(o3 ~ s(month, bs = "cc")  + s(trend), data = monthly,
           correlation = corAR1(form = ~ month | year))
summary(M2$gam)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## o3 ~ s(month, bs = "cc") + s(trend)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   71.316      0.493     145   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##           edf Ref.df    F p-value
## s(month) 6.91      8 42.2 < 2e-16 ***
## s(trend) 1.00      1 15.1 0.00013 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.643
##   Scale est. = 38.863    n = 276
```

The ACF plot is shown in Figure C.6 and shows that the autocorrelation has been dealt with and we can be rather more confident about the trend component (not plotted). Note that in this case we need to use the normalized residuals to get residuals that take account of the fitted correlation structure.

Note that model M2 assumes that the trend and seasonal terms vary independently of one another. However, if the seasonal amplitude and/or phase change over time then a model that accounts for the interaction between the two may be better. Indeed, this does seem to be the case here, as shown by the improved fit of the model below. This model uses a tensor product smooth (`te`) and the reason for doing this and not using an isotropic smooth (`s`) is that the trend and seasonal components are essentially on different scales. We would not necessarily want to apply the same level of smoothness to both components. An example of covariates on the same scale would be latitude and longitude.

```
acf(residuals(M2$lme, type = "normalized"))
```

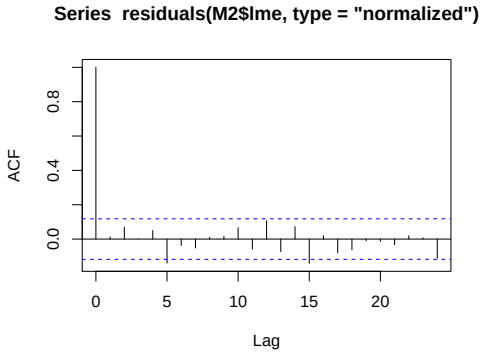


FIGURE C.6 ACF for the residuals of model M2.

```
M3 <- gamm(o3 ~ s(month, bs = "cc")  + te(trend, month), data = monthly,
           correlation = corAR1(form = ~ month | year))
summary(M3$gam)

##
## Family: gaussian
## Link function: identity
##
## Formula:
## o3 ~ s(month, bs = "cc") + te(trend, month)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   71.321      0.458     156   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                  edf Ref.df     F p-value
## s(month)        6.89   8.00 22.16 < 2e-16 ***
## te(trend,month) 4.17   4.17  5.74 0.00016 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.667
##   Scale est. = 36.121    n = 276
```

It becomes a bit more difficult to plot the two-way interaction between the trend and the month, but it is possible with a surface as shown in Figure C.7. This plot shows for example that during summertime the trends component varies little. However for the autumn and winter months there has been a much greater increase in the trend compnent for O_3.

While there have been many steps involved in this short analysis, the data at Mace Head are not typical of most air quality data observed, say in urban areas. Much of the data considered in these areas does not appear to have significant autocorrelation in the residuals once the seasonal variation has been accounted for, therefore avoiding the complexities of taking account of the correlation structure of the data. It may be for example that sites like Mace Head and a pollutant such as O_3 are much more prone to larger scale atmospheric processes that are not captured by these models.

```
plot(M3$gam, select = 2, pers = TRUE, theta = 225, phi = 10,ticktype = "detailed")
```

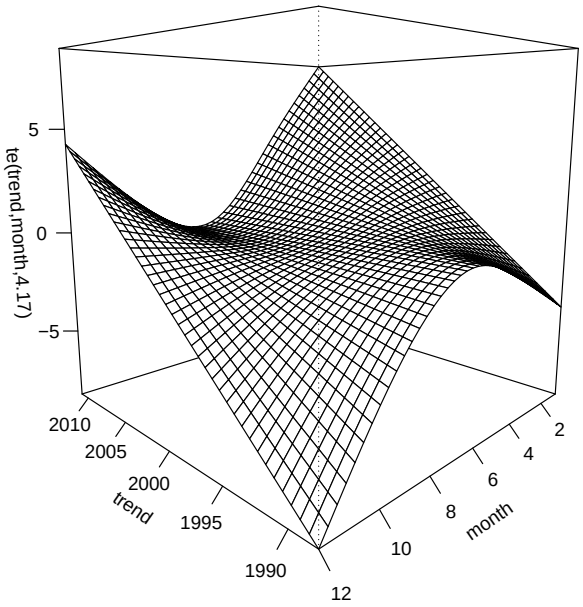


FIGURE C.7 Plot showing the two-way interaction between the trend and seasonal components.

D Production of HYSPLIT trajectory files

As discussed in Section 26, **openair** can import pre-calculated trajectory data for specified locations. The data are stored on a King's College London webserver to make it easy to import 96-hour back trajectory data. Several users have requested how they can run HYSPLIT themselves e.g. for different trajectory start heights or for many locations. This section provides the code necessary to run the HYSPLIT model. The code below assumes that full years are run, but it could be adopted for shorter periods. There are three main parts to producing trajectory files:

1. Download and install the NOAA Hysplit model, somewhere with write access (see below).
2. Download the monthly meteorological (.gbl) files also from the NOAA website.
3. Copy and paste the three functions shown below into R (`read.files`, `add.met` and `procTraj`), taking note of the locations of the meteorological input files and the output locations.

This section assumes users are using a Windows operating system. There is a slightly modified version of the code below that will work on Mac OS X. Please contact David Carslaw directly should you require this code.

To run back trajectories it is necessary to download the meteorological data files. The easiest way to download the meteorological files is using the function below.

```
getMet <- function (year = 2013, month = 1, path_met = "~/TrajData/") {

    for (i in seq_along(year)) {
        for (j in seq_along(month)) {
            download.file(url = paste0("ftp://arlftp.arlhq.noaa.gov/archives/reanalysis/RP",
                                       year[i], sprintf("%02d", month[j]), ".gbl"),
                          destfile = paste0(path_met, "RP", year[i],
                                       sprintf("%02d", month[j]), ".gbl"), mode = "wb")
        }
    }
}
```

The function will download monthly met files (each about 120 MB) to the chosen directory. Note that the met data files only need be downloaded once. For example, to download files for 2013:

```
getMet(year = 2013, month = 1:12)
```

Three functions need to be loaded as shown below. To make the trajectory files code such as that shown below should be run. Note the use of full paths.

```
for (i in 2010:2011) {
    procTraj(lat = 36.134, lon = -5.347, year = i,
             name = "gibraltar", hours = 96,
             met = "c:/users/david/TrajData/", out = "c:/users/david/TrajProc/",
             hy.path = "c:/users/david/hysplit4/")
}
```

In this example, 2010 and 2011 will be run. The latitude (`lat`) and longitude (`lon`) are provided in decimal form (the above is for Gibraltar). The processed files will be given a name determined by the name option in the form name_Year.RData automatically. The `hours` option is for the length of the back trajectories, so 96 is for 4 days. It can be longer or shorter — but more days will take more time to run and generate bigger files. Note however if longer periods of time are run e.g. 10 days, **openair** can easily subset the data for shorter periods. The `met` option gives the full directory path to where the meteorological data you downloaded are stored. The `out` option gives the full directory path where the processed output is stored. This is what **openair** reads and is the type of file that is put on a server for remote reading. There is also a `height` option which is 10 m by default which controls the start height of the back trajectories.

The default install location for Hysplit on Windows is c:/hysplit4. One reason to install it elsewhere is to have write access because Hysplit needs to write files to that location. It is recommended therefore that Hysplit is installed somewhere with write access. For example, on my Windows machine I have installed it at C:/users/david/hysplit4/. There is an option `hy.path` to choose the location of Hysplit.

Once the met files are downloaded it should all run easily. Typically it takes about 1 to 2 hours to run a year.

It will then be necessary to store the RData files somewhere convenient. **openair** can use `importTraj` to read a local file rather than from the King's College web server. There is a `local` option that should be the full directory path to where the processed trajectory files are stored. For example, local pre-calculated back trajectories can be imported by:

```
traj <- importTraj(site = "london", year = 2010, local = "~/TrajProc/")
```

```
read.files <- function(hours = 96, hy.path) {
    ## find tdump files
    files <- Sys.glob("tdump*")
    output <- file('Rcombined.txt', 'w')

    ## read through them all, ignoring 1st 7 lines
    for (i in files){
        input <- readLines(i)
        input <- input[-c(1:7)]  # delete header
        writeLines(input, output)
    }
    close(output)

    ## read the combined txt file
    traj <- read.table(paste0(hy.path, "working/Rcombined.txt"), header = FALSE)
    traj <- subset(traj, select = -c(V2, V7, V8))

    traj <- rename(traj, c(V1 = "receptor", V3 = "year", V4 = "month", V5 = "day",
                           V6 = "hour", V9 = "hour.inc", V10 = "lat", V11 = "lon",
                           V12 = "height", V13 = "pressure"))

    ## hysplit uses 2-digit years ...
    year <- traj$year[1]
    if (year < 50) traj$year <- traj$year + 2000 else traj$year <- traj$year + 1900

    traj$date2 <- with(traj, ISOdatetime(year, month, day, hour,  min = 0, sec = 0,
                                         tz = "GMT"))
    ## arrival time
    traj$date <- traj$date2 - 3600 * traj$hour.inc
    traj
}


add.met <- function(month, Year, met, bat.file) {

    ## if month is one, need previous year and month = 12
    if (month == 0) {
        month <- 12
        Year <- as.numeric(Year) - 1
    }

    if (month < 10) month <- paste("0", month, sep = "")
    ## add first line

    write.table(paste("echo", met, "      >>CONTROL"),
                bat.file, col.names = FALSE,
                row.names = FALSE, quote = FALSE, append = TRUE)

    x <- paste("echo RP", Year, month, ".gbl       >>CONTROL", sep = "")
    write.table(x, bat.file, col.names = FALSE,
                row.names = FALSE, quote = FALSE, append = TRUE)
}
```

```
procTraj <- function(lat = 51.5, lon = -0.1, year = 2010, name = "london",
                     met = "c:/users/david/TrajData/", out = "c:/users/david/TrajProc/",
                     hours = 96, height = 10, hy.path = "c:/users/david/hysplit4/") {
    ## hours is the back trajectory time e.g. 96 = 4-day back trajectory
    ## height is start height (m)
    lapply(c("openair", "plyr", "reshape2"), require, character.only = TRUE)

    ## function to run 12 months of trajectories
    ## assumes 96 hour back trajectories, 1 receptor
    setwd(paste0(hy.path, "working/"))

    ## remove existing "tdump" files
    path.files <- paste0(hy.path, "working/")
    bat.file <- paste0(hy.path, "working/test.bat") ## name of BAT file to add to/run
    files <- list.files(path = path.files, pattern = "tdump")
    lapply(files, function(x) file.remove(x))

    start <- paste(year, "-01-01", sep = "")
    end <- paste(year, "-12-31 18:00", sep = "")
    dates <- seq(as.POSIXct(start, "GMT"), as.POSIXct(end, "GMT"), by = "3 hour")

    for (i in 1:length(dates)) {

        year <- format(dates[i], "%y")
        Year <- format(dates[i], "%Y") # Long format
        month <- format(dates[i], "%m")
        day <- format(dates[i], "%d")
        hour <- format(dates[i], "%H")

        x <- paste("echo", year, month, day, hour, "        >CONTROL")
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE)

        x <-  "echo 1                 >>CONTROL"
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        x <- paste("echo", lat, lon, height, "     >>CONTROL")
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        x <- paste("echo ", "-", hours, "               >>CONTROL", sep = "")
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        x <- "echo 0                >>CONTROL
              echo 10000.0          >>CONTROL
              echo 3                >>CONTROL"

        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        ## processing always assumes 3 months of met for consistent tdump files
        months <- as.numeric(unique(format(dates[i], "%m")))
        months <- c(months, months + 1:2)
        months <- months - 1 ## to make sure we get the start of the previous year
        months <- months[months <= 12]
        if (length(months) == 2) months <- c(min(months) - 1, months)

        for (i in 1:3)
            add.met(months[i], Year, met, bat.file)

        x <-  "echo ./                >>CONTROL"
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        x <- paste("echo tdump", year, month, day, hour, "          >>CONTROL", sep = "")
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        x <- "c:\\users\\david\\hysplit4\\exec\\hyts_std"
        write.table(x, bat.file, col.names = FALSE,
                    row.names = FALSE, quote = FALSE, append = TRUE)

        ## run the file
        system(paste0(hy.path, 'working/test.bat'))
    }

    ## combine files and make data frame

    traj <- read.files(hours, hy.path)

    ## write R object to file
    file.name <- paste(out, name, Year, ".RData", sep = "")
    save(traj, file = file.name)
}
```