Displaying Time Series, Spatial, and Space-Time Data with R

Oscar Perpiñán Lamigueiro

Contents

Chapter 1

Introduction

1.1 What This Book Is About

A data graphic is not only a static image but also tells a story about the data. It activates cognitive processes that are able to detect patterns and discover information not readily available with the raw data. This is particularly true for time series, spatial, and space-time datasets.

There are several excellent books about data graphics and visual perception theory, with guidelines and advice for displaying information, including visual examples. Let's mention *The Elements of Graphical Data* (Cleveland 1994) and *Visualizing Data* (Cleveland 1993) by W. S. Cleveland, *Envisioning Information* (Tufte 1990) and *The Visual Display of Quantitative Information* (Tufte 2001) by E. Tufte, *The Functional Art* by A. Cairo (Cairo 2012), and *Visual Thinking for Design* by C. Ware (Ware 2008). Ordinarily, they do not include the code or software tools to produce those graphics.

On the other hand, there is a collection of books that provides code and detailed information about the graphical tools available with R. Commonly they do not use real data in the examples and do not provide advice for improving graphics according to visualization theory. Three books are the unquestioned representatives of this group: *R Graphics* by P. Murrell (Murrell 2011), *Lattice: Multivariate Data Visualization with R* by D. Sarkar (Sarkar 2008), and *ggplot2: Elegant Graphics for Data Analysis* by H. Wickham (Wickham 2009).

This book proposes methods to display time series, spatial, and space-time data using R, and aims to be a synthesis of both groups providing code and detailed information to produce high-quality graphics with practical examples.

1.2 What You Will *Not* Find in This Book

- **This is not a book to learn R.**

  Readers should have a fair knowledge of programming with R to understand the book. In addition, previous experience with the `zoo`, `sp`, `raster`, `lattice`, `ggplot2`, and `grid` packages is helpful.

  If you need to improve your R skills, consider these information sources:

  - Introduction to R[1].
  - Official manuals[2].
  - Contributed documents[3].
  - Mailing lists[4].
  - R-bloggers[5].
  - Books related to R[6] and particularly *Software for Data Analysis* by John M. Chambers (Chambers 2008).

- **This book does not provide an exhaustive collection of visualization methods**.

  Instead, it illustrates what I found to be the most useful and effective methods. Notwithstanding, each part includes a section titled "Further Reading" with bibliographic proposals for additional information.

- **This book does not include a complete review or discussion of R packages**.

[1] http://cran.r-project.org/doc/manuals/R-intro.html
[2] http://cran.r-project.org/manuals.html
[3] http://cran.r-project.org/other-docs.html
[4] http://www.r-project.org/mail.html
[5] http://www.r-bloggers.com
[6] http://www.r-project.org/doc/bib/R-books.html

Their most useful functions, classes, and methods regarding data and graphics are outlined in the introductory chapter of each part, and conveniently illustrated with the help of examples. However, if you need detailed information about a certain aspect of a package, you should read the correspondent package manual or vignette. Moreover, if you want to know additional alternatives, you can navigate through the CRAN Task Views about Time Series[7], Spatial Data[8], Spatiotemporal Data[9], and Graphics[10].

- **Finally, this book is not a handbook of data analysis, geostatistics, point pattern analysis, or time series theory**.

  Instead, this book is focused on the exploration of data with visual methods, so it may be framed in the Exploratory Data Analysis approach. Therefore, this book may be a useful complement for superb bibliographic references where you will find plenty of information about those subjects. For example, (Chatfield 2003), (Cressie and Wikle 2011), (Slocum 2005) and (R. S. Bivand, E. J. Pebesma, and Gomez-Rubio 2008).

1.3 How to Read This Book

This book is organized into three parts, each devoted to different types of data. Each part comprises several chapters according to the various visualization methods or data characteristics. The chapters are structured as independent units so readers can jump directly to a certain chapter according to their needs. Of course, there are several dependencies and redundancies between the sets of chapters that have been conveniently signaled with cross-references.

The content of each chapter illustrates how to display a dataset starting with an easy and direct approach. Often this first result is not entirely satisfactory so additional improvements are progressively added. Each step involves additional complexity which, in some cases, can be overwhelming during a first reading. Thus, some sections, marked with the sign ⚽, can be safely skipped for later reading.

Although I have done my best to help readers understand the methods and code, you should not expect to understand it after one reading. The

[7] http://cran.r-project.org/web/views/TimeSeries.html
[8] http://cran.r-project.org/web/views/Spatial.html
[9] http://cran.r-project.org/web/views/SpatioTemporal.html
[10] http://cran.r-project.org/web/views/Graphics.html

key is practical experience, and the best way is to try out the code with the provided data **and** modify it to suit your needs with your own data. There is a website and a code repository to help you in this task.

1.3.1 Website and Code Repository

The book website with the main graphics of this book is located at

http://oscarperpinan.github.com/spacetime-vis/

The full code is freely available from the repository:

https://github.com/oscarperpinan/spacetime-vis

On the other hand, the datasets used in the examples are either available at the repository or can be freely obtained from other websites. It must be underlined that the combination of code and data freely available allows this book to be fully reproducible.

I have chosen the datasets according to two main criteria:

- They are freely available without restrictions for public use.
- They cover different scientific and professional fields (meteorology and climate research, economy and social sciences, energy and engineering, environmental research, epidemiology, etc.).

The repository and the website can be downloaded as compressed files[11], and if you use `git`, you can clone the repository with

```
git clone https://github.com/oscarperpinan/spacetime-vis.git
```

1.4 R Graphics

There are two distinct graphics systems built into R, referred to as traditional and grid graphics. Grid graphics are produced with the `grid` package (Murrell 2011), a flexible low-level graphics toolbox. Compared with the traditional graphics model, it provides more flexibility to modify or

[11] Repository: https://github.com/oscarperpinan/spacetime-vis/archive/master.zip, Website: https://github.com/oscarperpinan/spacetime-vis/archive/gh-pages.zip

add content to an existent graphical output, better support for combining different outputs easily, and more possibilities for interaction. All the graphics in this book have been produced with the grid graphics model.

Other packages are constructed over it to provide high-level functions, most notably the `lattice` and `ggplot2` packages.

1.4.1 lattice

The `lattice` package (Sarkar 2008) is an independent implementation of Trellis graphics, which were mostly influenced by *The Elements of Graphing Data* (Cleveland 1994). Trellis graphics often consist of a rectangular array of panels. The `lattice` package uses a *formula* interface to define the structure of the array of panels with the specification of the variables involved in the plot. The result of a `lattice` high-level function is a `trellis` object.

For bivariate graphics, the formula is generally of the form `y ~ x` representing a single panel plot with `y` versus `x`. This formula can also involve expressions. The main function for bivariate graphics is `xyplot`.

Optionally, the formula may be `y ~ x | g1 * g2` and `y` is represented against `x` conditional on the variables `g1` and `g2`. Each unique combination of the levels of these conditioning variables determines a subset of the variables `x` and `y`. Each subset provides the data for a single panel in the Trellis display, an array of panels laid out in columns, rows, and pages.

For example, in the following code, the variable `wt` of the dataset `mtcars` is represented against the `mpg`, with a panel for each level of the categorical variable `am`. The points are grouped by the values of the `cyl` variable.

```
xyplot(wt ~ mpg | am, data = mtcars, groups = cyl)
```

For trivariate graphics, the formula is of the form `z ~ x * y`, where `z` is a numeric response, and `x` and `y` are numeric values evaluated on a rectangular grid. Once again, the formula may include conditioning variables, for example `z ~ x * y | g1 * g2`. The main function for these graphics is `levelplot`.

The plotting of each panel is performed by the panel function, specified in a high-level function call as the `panel` argument. Each high-level `lattice` function has a default panel function, although the user can create new Trellis displays with custom panel functions.

`lattice` is a member of the recommended packages list so it is commonly distributed with R itself. There are more than 250 packages depending on it, and the most important packages for our purposes (`zoo`, `sp`, and `raster`) define methods to display their classes using `lattice`.

On the other hand, the `latticeExtra` package (Sarkar and Andrews 2012) provides additional flexibility for the somewhat rigid structure of the Trellis framework implemented in `lattice`. This package complements the `lattice` with the implementation of layers via the `layer` function, and superposition of `trellis` objects and layers with the `+.trellis` function. Using both packages, you can define a graphic with the formula interface (under the `lattice` model) and overlay additional content as layers (following the `ggplot2` model).

1.5 ggplot2

The `ggplot2` package (Wickham 2009) is an implementation of the system proposed in *The Grammar of Graphics* (Wilkinson 1999), a general scheme for data visualization that breaks up graphs into semantic components such as scales and layers. Under this framework, the definition of the graphic with `ggplot2` is done with a combination of several functions that provides the components, instead of the formula interface of `lattice`.

With `ggplot2`, a graphic is composed of:

- A dataset, `data`, and a set of mappings from variables to aesthetics, `aes`.
- One or more layers, each composed of: a geometric object, `geom_*`, to control the type of plot you create (points, lines, etc.); a statistical transformation, `stat_*`; and a position adjustment (and optionally, additional dataset and aesthetic mappings).
- A scale, `scale_*`, to control the mapping from data to aesthetic attributes. Scales are common across layers to ensure a consistent mapping from data to aesthetics.
- A coordinate system, `coords_*`.
- Optionally, a faceting specification, `facet_*`, the equivalent of Trellis graphics with panels.

The function `ggplot` is typically used to construct a plot incrementally, using the `+` operator to add layers to the existing ggplot object. For instance, the following code (equivalent to the previous `lattice` example) uses `mtcars` as the dataset, and maps the `mpg` variable on the x-axis and the `wt` variable on the y-axis. The geometric object is the point using the `cyl` variable to control the color. Finally, the levels of the `am` variable define the panels of the graphic.

```
ggplot(mtcars, aes(mpg, wt)) +
   geom_point(aes(colour=factor(cyl))) +
   facet_grid(. ~ am)
```

This package is increasingly popular, with a list of more than ninety packages depending on it. On the other hand, few packages provide method definitions based on `ggplot2` to display their classes. In our context, only the `zoo` package defines the `autoplot` function based on it.

1.5.1 Comparison between lattice and ggplot2

Which package to choose is, for a wide range of datasets, a question of personal preferences. You may be interested in a comparison between them published in a series of blog posts[12]. However, the major drawback of `ggplot2` is its considerably slower speed when dealing with large datasets[13], so you should be cautious with large spatial and spatiotemporal data. Consequently, most of the code in Part **??** contains alternatives defined both with `lattice` and with `ggplot2`. However, because of the speed problem and the absence of `ggplot2` functions in the corresponding packages, only a minor fraction of the code in Parts **??** and **??** contains graphics defined with `ggplot2`.

1.6 Packages

Throughout the book, several R packages are used. All of them are available from CRAN, and you must install them before using the code. Most of them are loaded at the start of the code of each chapter, although some of them are loaded later if they are used only inside optional sections (marked with ✿). You should install the last version available at CRAN to ensure correct functioning of the code.

Although the introductory chapter of each part includes a section with an outline of the most relevant packages, some of them deserve to be highlighted here:

- `zoo` (Zeileis and Grothendieck 2005) provides infrastructure for time series using arbitrary classes for the time stamps (Section 2.1.1).

[12] http://learnr.wordpress.com/2009/06/28/ggplot2-version-of-figures-in-lattice-multivari

[13] Take a look at the time comparison published as the final result of the previous series of blog posts, http://learnr.files.wordpress.com/2009/08/latbook.pdf

- `sp` (E. Pebesma 2012) provides a coherent set of classes and methods for the major spatial data types: points, lines, polygons, and grids (Section **??**). `spacetime` (E. Pebesma 2012) defines classes and methods for spatiotemporal data, and methods for plotting data as map sequences or multiple time series (Section **??**).

- `raster` (R. J. Hijmans 2013) is a major extension of gridded spatial data classes. It provides a unified access method to different raster formats, permitting large objects to be analyzed with the definition of basic and high-level processing functions (Sections **??** and **??**). `rasterVis` (Oscar Perpiñán and R. Hijmans 2013) provides enhanced visualization of raster data with methods for spatiotemporal rasters (Sections **??** and **??**).

- `gridSVG` (Murrell and Potter 2013) converts any grid scene to an SVG document. The `grid.hyperlink` function allows a hyperlink to be associated with any component of the scene, the `grid.animate` function can be used to animate any component of a scene, and the `grid.garnish` function can be used to add SVG attributes to the components of a scene. By setting event handler attributes on a component, plus possibly using the `grid.script` function to add JavaScript to the scene, it is possible to make the component respond to user input such as mouse clicks.

1.7 Software Used to Write This Book

This book has been written using different computers running Debian GNU Linux and using several gems of open-source software:

- org-mode (Schulte et al. 2012), LaTeX, and AUCTeX, for authoring text and code.

- R (R Development Core Team 2013) with Emacs Speaks Statistics (Rossini et al. 2004).

- GNU Emacs as development environment.

1.8 About the Author

During the past 15 years, my main area of expertise has been photovoltaic solar energy systems, with a special interest in solar radiation. Initially I

worked as an engineer for a private company and I was involved in several commercial and research projects. The project teams were partly integrated by people with low technical skills who relied on the input from engineers to complete their work. I learned how a good visualization output eased the communication process.

Now I work as a professor and researcher at the university. Data visualization is one of the most important tools I have available. It helps me embrace and share the steps, methods, and results of my research. With students, it is an inestimable partner in helping them understand complex concepts.

I have been using R to simulate the performance of photovoltaic energy systems and to analyze solar radiation data, both as time series and spatial data. As a result, I have developed packages that include several graphical methods to deal with multivariate time series (namely, `solaR` (Oscar Perpiñán 2012)) and space-time data (`rasterVis`).

1.9 Acknowledgments

Writing a book is often described as a solitary activity. It is certainly difficult to write when you are with friends or spending time with your family,... although with three little children at home I have learned to write prose and code while my baby wants to learn typing and my daughters need help to share a family of dinosaurs.

Seriously speaking, solitude is the best partner of a writer. But when I am writing or coding I feel I am immersed in a huge collaborative network of past and present contributors. Piotr Kropotkin described it with the following words (Kropotkin 1906):

> Thousands of writers, of poets, of scholars, have laboured to increase knowledge, to dissipate error, and to create that atmosphere of scientific thought, without which the marvels of our century could never have appeared. And these thousands of philosophers, of poets, of scholars, of inventors, have themselves been supported by the labour of past centuries. They have been upheld and nourished through life, both physically and mentally, by legions of workers and craftsmen of all sorts.

And Lewis Mumford claimed (Mumford 1934):

> Socialize Creation! What we need is the realization that the creative life, in all its manifestations, is necessarily a social product.

I want to express my deepest gratitude and respect to all those women and men who have contributed and contribute to strengthening the communities of free software, open data, and open science. My special thanks go to the people of the R community: users, members of the R Core Development Team, and package developers.

With regard to this book in particular, I would like to thank John Kimmel for his constant support, guidance, and patience.

Last, and most importantly, thanks to Candela, Marina, and Javi, my crazy little shorties, my permanent source of happiness, imagination, and love. Thanks to María, *mi amor, mi cómplice y todo*.

Part I

Time Series

Chapter 2

Displaying Time Series: Introduction

A time series is a sequence of observations registered at consecutive time instants. When these time instants are evenly spaced, the distance between them is called the sampling interval. The visualization of time series is intended to reveal changes of one or more quantitative variables through time, and to display the relationships between the variables and their evolution through time.

The standard time series graph displays the time along the horizontal axis. Several variants of this approach can be found in Chapter 3. On the other hand, time can be conceived as a grouping or conditioning variable (Chapter 4). This solution allows several variables to be displayed together with a scatterplot, using different panels for subsets of the data (time as a conditioning variable) or using different attributes for groups of the data (time as a grouping variable). Moreover, time can be used as a complementary variable that adds information to a graph where several variables are confronted (Chapter 5).

These chapters provide a variety of examples to illustrate a set of useful techniques. These examples make use of several datasets (available at the book website) described in Chapter 6.

2.1 Packages

The CRAN Tasks View "Time Series Analysis" [1] summarizes the packages for reading, vizualizing, and analyzing time series. This section provides a brief introduction to the `zoo` and `xts` packages. Most of the information has been extracted from their vignettes, webpages, and help pages. You should read them for detailed information.

Both packages extensively use the time classes defined in `R`. The interested reader will find an overview of the different time classes in `R` in (Ripley and Hornik 2001) and (Grothendieck and Petzoldt 2004).

2.1.1 zoo

The `zoo` package (Zeileis and Grothendieck 2005) provides an `S3` class with methods for indexed totally ordered observations. Its key design goals are independence of a particular index class and consistency with base `R` and the `ts` class for regular time series.

Objects of class `zoo` are created by the function `zoo` from a numeric vector, matrix, or a factor that is totally ordered by some index vector. This index is usually a measure of time but every other numeric, character, or even more abstract vector that provides a total ordering of the observations is also suitable. It must be noted that this package defines two new index classes, `yearmon` and `yearqtr`, for representing monthly and quarterly data, respectively.

The package defines several methods associated with standard generic functions such as `print`, `summary`, `str`, `head`, `tail`, and `[` (subsetting). In addition, standard mathematical operations can be performed with `zoo` objects, although only for the intersection of the indexes of the objects.

On the other hand, the data stored in `zoo` objects can be extracted with `coredata`, which drops the index information, and can be replaced by `coredata<-`. The index can be extracted with `index` or `time`, and can be modified by `index<-`. Finally, the `window` and `window<-` methods extract or replace time windows of `zoo` objects.

Two `zoo` objects can be merged by common indexes with `merge` and `cbind`. The `merge` method combines the columns of several objects along the union or the intersection of the indexes. The `rbind` method combines the indexes (rows) of the objects.

[1] http://CRAN.R-project.org/view=TimeSeries

The aggregate method splits a zoo object into subsets along a coarser index grid, computes a function (sum is the default) for each subset, and returns the aggregated zoo object.

This package provides four methods for dealing with missing observations:

1. na.omit removes incomplete observations.
2. na.contiguous extracts the longest consecutive stretch of non-missing values.
3. na.approx replaces missing values by linear interpolation.
4. na.locf replaces missing observations by the most recent non-NA prior to it.

The package defines interfaces to read.table and write.table for reading, read.zoo, and writing, write.zoo, zoo series from or to text files. The read.zoo function expects either a text file or connection as input or a data.frame. write.zoo first coerces its argument to a data.frame, adds a column with the index, and then calls write.table.

2.1.2 xts

The xts package (Ryan and Ulrich 2013) extends the zoo class definition to provide a general time-series object. The index of an xts object must be of a time or date class: Date, POSIXct, chron, yearmon, yearqtr, or timeDate. With this restriction, the subset operator [is able to extract data using the ISO:8601 [2] time format notation CCYY-MM-DD HH:MM:SS. It is also possible to extract a range of times with a from/to notation, where both from and to are optional. If either side is missing, it is interpreted as a request to retrieve data from the beginning, or through the end of the data object.

Furthermore, this package provides several time-based tools:

- endpoints identifies the endpoints with respect to time.
- to.period changes the periodicity to a coarser time index.
- The functions period.* and apply.* evaluate a function over a set of non-overlapping time periods.

[2] http://en.wikipedia.org/wiki/ISO_8601

2.2 Further Reading

- (Wills 2011) provides a systematic analysis of the visualization of time series, and a section of (Jeffrey Heer, Bostock, and Ogievetsky 2010) summarizes the main techniques to display time series.
- (Cleveland 1994) includes a section about time series visualization with a detailed discussion of the banking to 45° technique and the cut-and-stack method. (J. Heer and Agrawala 2006) propose the multi-scale banking, a technique to identify trends at various frequency scales.
- (Few 2008; J. Heer, Kong, and Agrawala 2009) explain in detail the foundations of the horizon graph (Section 3).
- The *small multiples* concept (Sections 3.2 and 4.1) is illustrated in (Tufte 2001; Tufte 1990).
- Stacked graphs are analyzed in (Byron and Wattenberg 2008), and the ThemeRiver technique is explained in (Havre et al. 2002).
- (Cleveland 1994; Friendly and Denis 2005) study the scatterplot matrices (Section 4.1), and (D. B. Carr et al. 1987) provide information about hexagonal binning.
- (Harrower and Fabrikant 2008) discuss the use of animation for the visualization of data. (Few 2007) exposes a software tool resembling the Trendalyzer.
- The `D3` gallery [3] shows several great examples of time-series visualizations using the JavaScript library `D3.js`.

[3] https://github.com/mbostock/d3/wiki/Gallery

Chapter 3

Time on the Horizontal Axis

The most frequent visualization method of a time series uses the horizontal axis to depict the time index. This chapter illustrates several variants to display multivariate time series: multiple time series with different scales, variables with the same scale, and stacked graphs.

3.1 Time Graph of Variables with Different Scales

There is a variety of scientific research interested in the relationship among several meteorological variables. A suitable approach is to display the time evolution of all of them using a panel for each of the variables. The superposition of variables with different characteristics is not very useful (unless their values were previously rescaled), so this option is postponed for Section 3.2.

For this example we will use the 8 years of daily data from the SIAR meteorological station located at Aranjuez (Madrid). This multivariate time series can be displayed with the `xyplot` method of `lattice` for `zoo` objects with a panel for each variable (Figure 3.1).

```
library(zoo)
load('data/aranjuez.RData')
```

```
## The layout argument arranges panels in rows
xyplot(aranjuez, layout=c(1, ncol(aranjuez)))
```

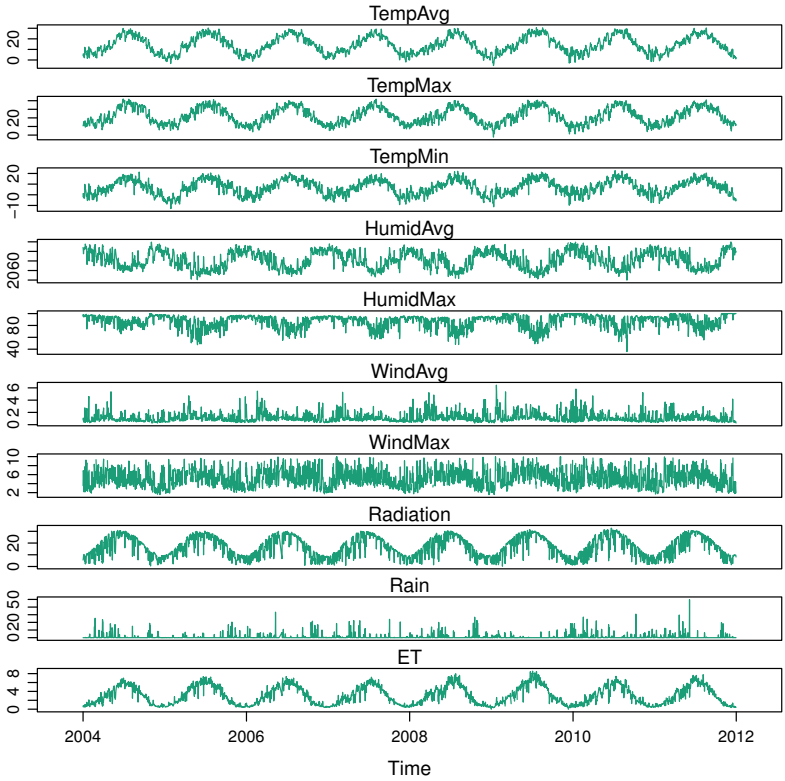

FIGURE 3.1: Time plot of the collection of meteorological time series of the Aranjuez station (`lattice` version).

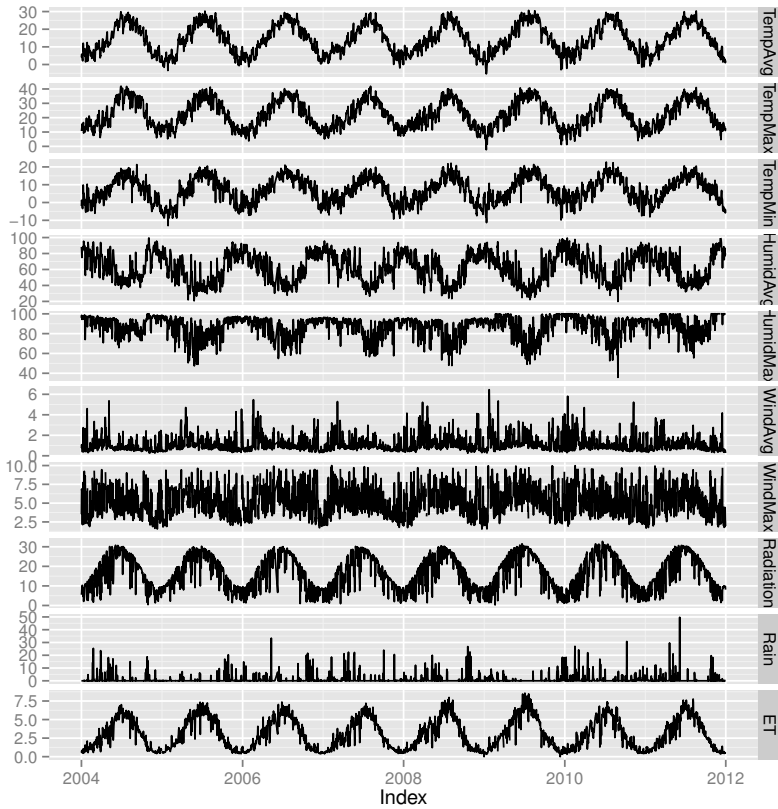

FIGURE 3.2: Time plot of the collection of meteorological time series of the Aranjuez station (`ggplot2` version).

The package `ggplot2` provides the generic method `autoplot` to automate the display of certain classes with a simple command. The package `zoo` provides an `autoplot` method for the `zoo` class with a result similar to that obtained with `xyplot` (Figure 3.2)

```
autoplot(aranjuez) + facet_free()
```

3.1.1 Annotations to Enhance the Time Graph

These first attempts can be improved with a custom panel function that generates the content of each panel using the information processed by

xyplot, or overlaying additional layers with autoplot. One of the main enhancements is to highlight certain time regions that fulfill certain conditions. The package latticeExtra provides a nice solution for xyplot with panel.xblocks. The result is displayed in Figure 3.3:

- The label of each time series is displayed with text inside each panel instead of using the strips mechanism. The panel.text prints the name of each variable with the aid of panel.number.
- The alternating of years is displayed with blocks of gray and white color using the panel.xblocks function from latticeExtra. The year is extracted (as character) from the time index of the zoo object with format.POSIXlt.
- Those values below the mean of each variable are highlighted with short red color blocks at the bottom of each panel, again with the panel.xblocks function.
- The maxima and minima are highlighted with small blue triangles.

Because the functions included in the panel function are executed consecutively, their order determines the superposition of graphical layers.

```
library(grid)
library(latticeExtra)

## Auxiliary function to extract the year value of a POSIXct time
## index
Year <- function(x)format(x, "%Y")

xyplot(aranjuez, layout=c(1, ncol(aranjuez)), strip=FALSE,
      scales=list(y=list(cex=0.6, rot=0)),
      panel=function(x, y, ...){
        ## Alternation of years
        panel.xblocks(x, Year,
                      col = c("lightgray", "white"),
                      border = "darkgray")
        ## Values under the average highlighted with red regions
        panel.xblocks(x, y<mean(y, na.rm=TRUE),
                      col = "indianred1",
                      height=unit(0.1, 'npc'))
        ## Time series
```

```
        panel.lines(x, y, col='royalblue4', lwd=0.5, ...)
        ## Label of each time series
        panel.text(x[1], min(y, na.rm=TRUE),
                 names(aranjuez)[panel.number()],
                 cex=0.6, adj=c(0, 0), srt=90, ...)
        ## Triangles to point the maxima and minima
        idxMax <- which.max(y)
        panel.points(x[idxMax], y[idxMax],
                   col='black', fill='lightblue', pch=24)
        idxMin <- which.min(y)
        panel.points(x[idxMin], y[idxMin],
                   col='black', fill='lightblue', pch=25)
      })
```

There is no equivalent `panel.xblocks` function that can be used with `ggplot2`. Therefore, the `ggplot2` version must explicitly compute the corresponding bands (years and regions below the average values):

- The first step in working with `ggplot` is to transform the `zoo` object into a `data.frame` in long format. `fortify` returns a `data.frame` with three columns: the time `Index`, a factor indicating the `Series`, and the corresponding `Value`.

```
timeIdx <- index(aranjuez)

long <- fortify(aranjuez, melt=TRUE)
```

- The bands of values below the average can be easily extracted with `scale` because these regions are negative when the `data.frame` is centered.

```
## Values below mean are negative after being centered
scaled <- fortify(scale(aranjuez, scale=FALSE), melt=TRUE)
## The 'scaled' column is the result of the centering.
## The new 'Value' column store the original values.
scaled <- transform(scaled, scaled=Value, Value=long$Value)
underIdx <- which(scaled$scaled <= 0)
## 'under' is the subset of values below the average
under <- scaled[underIdx,]
```

- The years bands are defined with the function `endpoints` from the `xts` package:

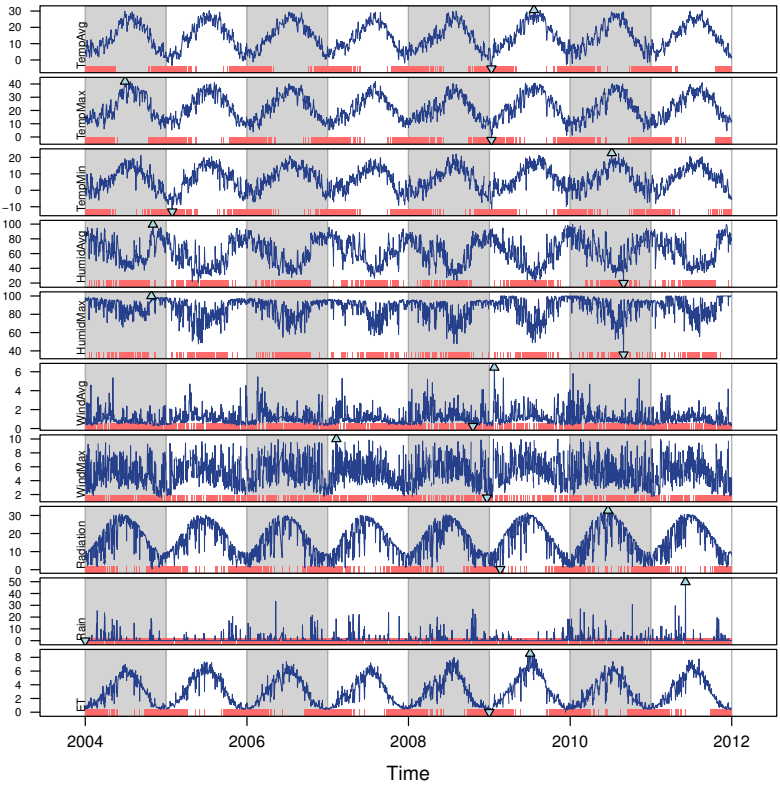

FIGURE 3.3: Enhanced time plot of the collection of meteorological time series of the Aranjuez station.

```
library(xts)
ep <- endpoints(timeIdx, on='years')
N <- length(ep[-1])
## 'tsp' is start and 'tep' is the end of each band
tep <- timeIdx[ep]
tsp <- timeIdx[ep[-(N+1)]+1]
## 'cols' is a vector with the color of each band
cols <- rep_len(c('gray', 'white'), N)
```

- The minima and maxima points of each variable are extracted with `apply`:

```
minIdx <- timeIdx[apply(aranjuez, 2, which.min)]
minVals <- apply(aranjuez, 2, min, na.rm=TRUE)
mins <- data.frame(Index=minIdx,
                   Value=minVals,
                   Series=names(aranjuez))

maxIdx <- timeIdx[apply(aranjuez, 2, which.max)]
maxVals <- apply(aranjuez, 2, max, na.rm=TRUE)
maxs <- data.frame(Index=maxIdx,
                   Value=maxVals,
                   Series=names(aranjuez))
```

- With `ggplot` we define the canvas, and the layers of information are added successively:

```
ggplot(data=long, aes(Index, Value)) +
   ## Time series of each variable
   geom_line(colour = "royalblue4", lwd = 0.5) +
   ## Year bands
   annotate(geom='rect', ymin = -Inf, ymax = Inf,
            xmin=tsp, xmax=tep,
            fill = cols, alpha = 0.4) +
   ## Values below average
   geom_rug(data=under,
          sides='b', col='indianred1') +
   ## Minima
   geom_point(data=mins, pch=25) +
   ## Maxima
   geom_point(data=maxs, pch=24) +
```

```
## Axis labels and theme definition
labs(x='Time', y=NULL) +
theme_bw() +
## Each series is displayed in a different panel with an
## independent y scale
facet_free()
```

Some messages from Figure 3.3:

- The radiation, temperature, and evotranspiration are quasi-periodic and are almost synchronized between them. Their local maxima appear in the summer and the local minima in the winter. Obviously, the summer values are higher than the average.
- The average humidity varies in oposition to the temperature and radiation cycle, with local maxima located during winter.
- The average and maximum wind speed, and rainfall vary in a more erratic way and do not show the evident periodic behavior of the radiation and temperature.
- The rainfall is different from year to year. The remaining variables do not show variations between years.
- The fluctuations of solar radiation are more apparent than the temperature fluctuations. There is hardly any day with temperatures below the average value during summer, while it is not difficult to find days with radiation below the average during this season.

3.2 Time Series of Variables with the Same Scale

As an example of time series of variables with the same scale, we will use measurements of solar radiation from different meteorological stations.

The first attempt to display this multivariate time series makes use of the `xyplot.zoo` method. The objective of this graphic is to display the behavior of the collection as a whole: the series are superposed in the same panel (`superpose=TRUE`) without legend (`auto.key=TRUE`), using thin lines and partial transparency[1]. Transparency softens overplotting problems and reveals density clusters because regions with more overlapping lines are darker. Figure 3.4 displays the variations around the time average (`avRad`).

[1] A similar result can be obtained with `autoplot` using `facets=NULL`.

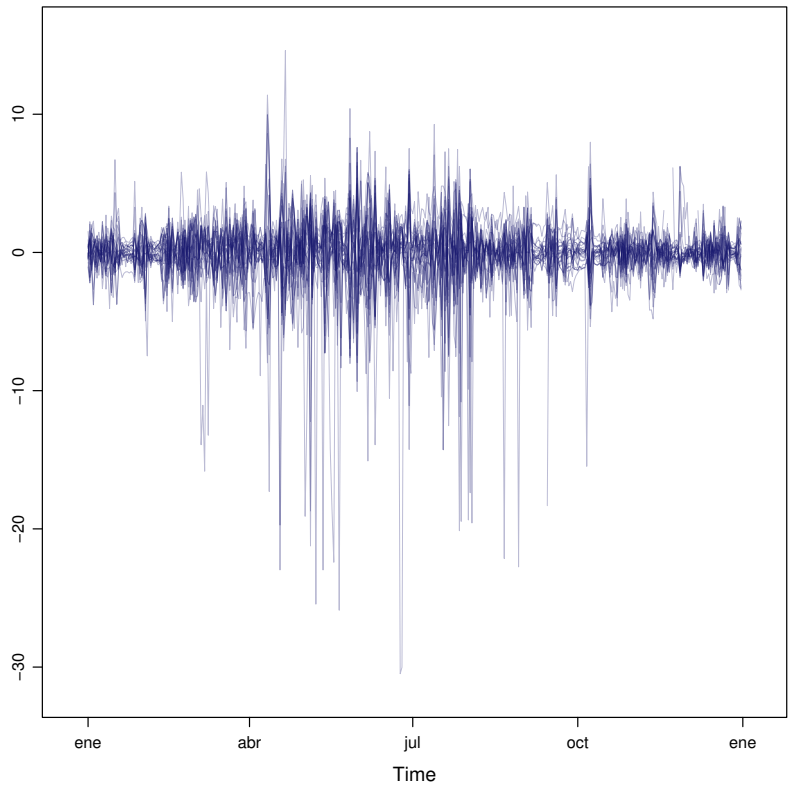

FIGURE 3.4: Time plot of the variations around time average of solar radiation measurements from the meteorological stations of Navarra.

```
load('data/navarra.RData')
```

```
avRad <- zoo(rowMeans(navarra, na.rm=1), index(navarra))
pNavarra <- xyplot(navarra - avRad,
                   superpose=TRUE, auto.key=FALSE,
                   lwd=0.5, alpha=0.3, col='midnightblue')
pNavarra
```

This result can be improved with different methods: the cut-and-stack method, and the horizon graph with `horizonplot`.

3.2.1 Aspect Ratio and Rate of Change

When a graphic is intended to inform about the rate of change, special attention must be paid to the aspect ratio of the graph, defined as the ratio of the height to the width of the graphical window. Cleveland analyzed the importance of the aspect ratio for judging rate of change. He concluded that we visually decode the information about the relative local rate of change of one variable with another by comparing the orientations of the local line segments that compose the polylines. The recommendation is to choose the aspect ratio so that the absolute values of the orientations of the segments are centered on 45° (banking to 45°).

The problem with banking to 45° is that the resulting aspect ratio is frequently too small. A suitable solution to minimize wasted space is the cut-and-stack method. The `xyplot.ts` method implement this solution with the combination of the arguments `aspect` and `cut`. The version of Figure 3.4 using banking to 45° and the cut-and-stack method is produced with

```
xyplot(navarra - avRad,
      aspect='xy', cut=list(n=3, overlap=0.1),
      strip=FALSE,
      superpose=TRUE, auto.key=FALSE,
      lwd=0.5, alpha=0.3, col='midnightblue')
```

3.2.2 The Horizon Graph

The horizon graph is useful in examining how a large number of series changes over time, and does so in a way that allows both comparisons between the individual time series and and independent analysis of each series. Moreover, extraordinary behaviors and predominant patterns are easily distinguished (J. Heer, Kong, and Agrawala 2009; Few 2008).

This graph displays several stacked series collapsing the y-axis to free vertical space:

- Positive and negative values share the same vertical space. Negative values are inverted and placed above the reference line. Sign is encoded using different hues (positive values in blue and negative values in red).
- Differences in magnitude are displayed as differences in color intensity (darker colors for greater differences).

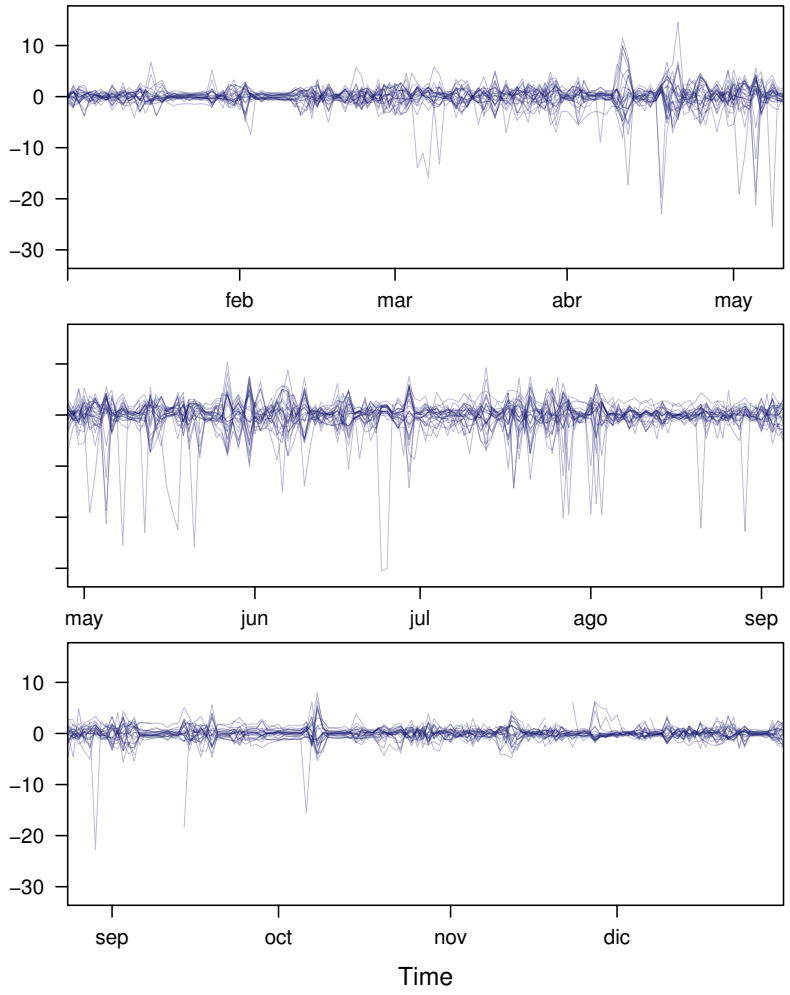

FIGURE 3.5: Cut-and-stack plot with banking to 45°.

- The color bands share the same baseline and are superposed, with darker bands in front of the ligther ones.

Because the panels share the same design structure, once this technique is understood, it is easy to establish comparisons or spot extraordinary events. This method is what Tufte described as small multiples (Tufte 1990).

Figure 3.6 displays the variations of solar radiation around the time average with an horizon graph using a row for each time series.

```
library(latticeExtra)

horizonplot(navarra-avRad,
          layout=c(1, ncol(navarra)),
          origin=0, colorkey=TRUE)
```

Figure 3.6 allows several questions to be answered:

- Which stations consistently measure above and below the average?
- Which stations resemble more closely the average time series?
- Which stations show erratic and uniform behavior?
- In each of the stations, is there any day with extraordinary measurements?
- Which part of the year is associated with more intense absolute fluctuations across the set of stations?

3.2.3 Time Graph of the Differences between a Time Series and a Reference

The horizon graph is also useful in revealing the differences between a univariate time series and another reference. For example, we might be interested in the departure of the observed temperature from the long-term average, or in other words, the temperature change over time.

Let's illustrate this approach with the time series of daily average temperatures measured at the meteorological station of Aranjuez. The reference is the long-term daily average calculated with `ave`.

```
Ta <- aranjuez$TempAvg
timeIndex <- index(aranjuez)
longTa <- ave(Ta, format(timeIndex, '%j'))
diffTa <- (Ta - longTa)
```

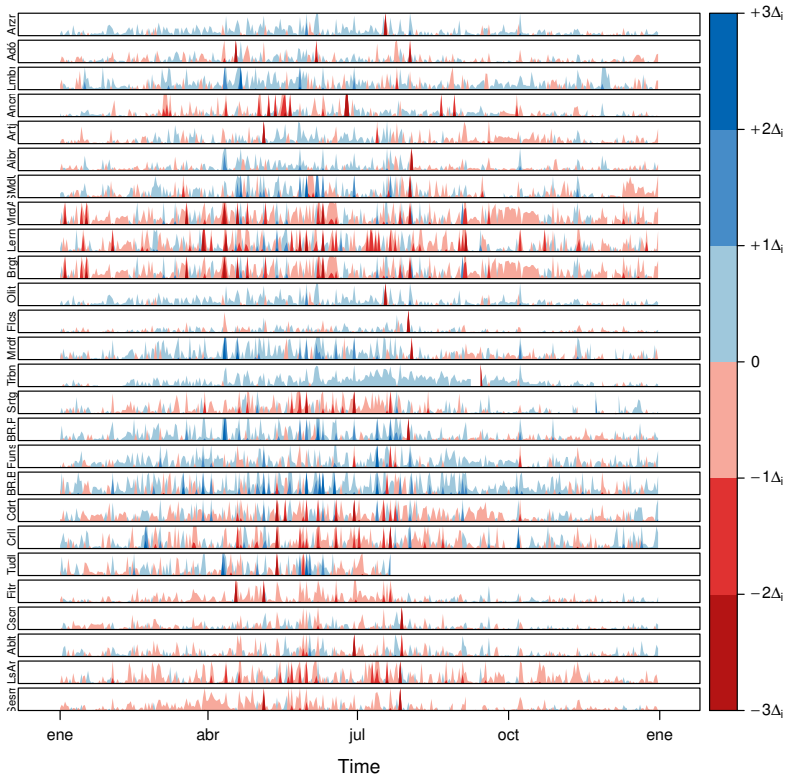

FIGURE 3.6: Horizon plot of variations around time average of solar radiation measurements from the meteorological stations of Navarra.

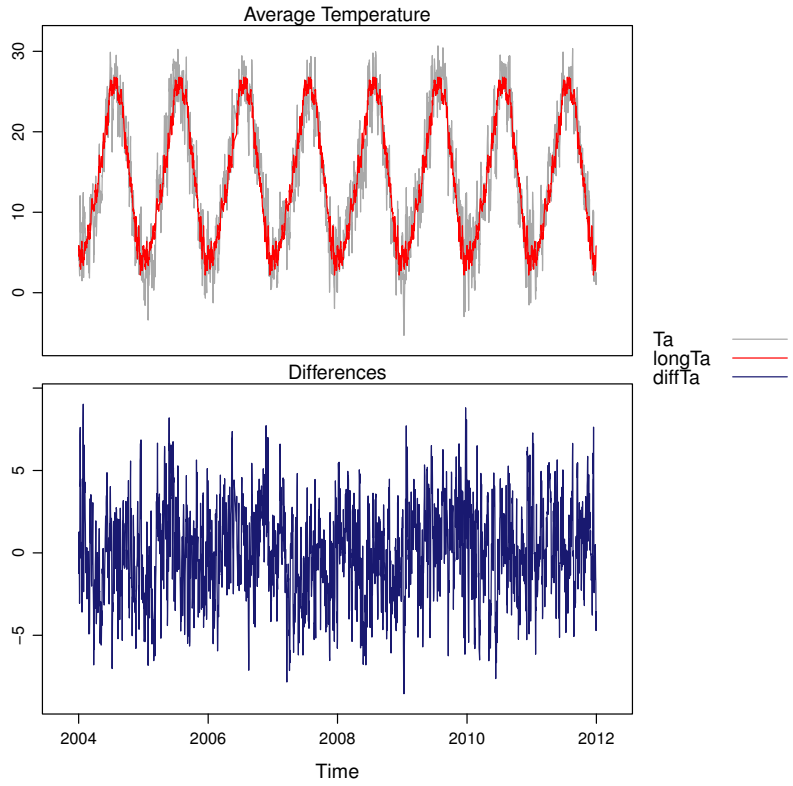

FIGURE 3.7: Daily temperature time series, its long-term average and the differences between them.

The temperature time series, the long-term average and the differences between them can be displayed with the `xyplot` method, now using `screens` to use a different panel for the differences time series (Figure 3.7)

```
xyplot(cbind(Ta, longTa, diffTa),
      col=c('darkgray', 'red', 'midnightblue'),
      superpose=TRUE, auto.key=list(space='right'),
      screens=c(rep('Average Temperature', 2), 'Differences'))
```

The horizon graph is better suited for displaying the differences. The next code again uses the cut-and-stack method (Figure 3.5) to distinguish between years. Figure 3.8 shows that 2004 started clearly above the aver-

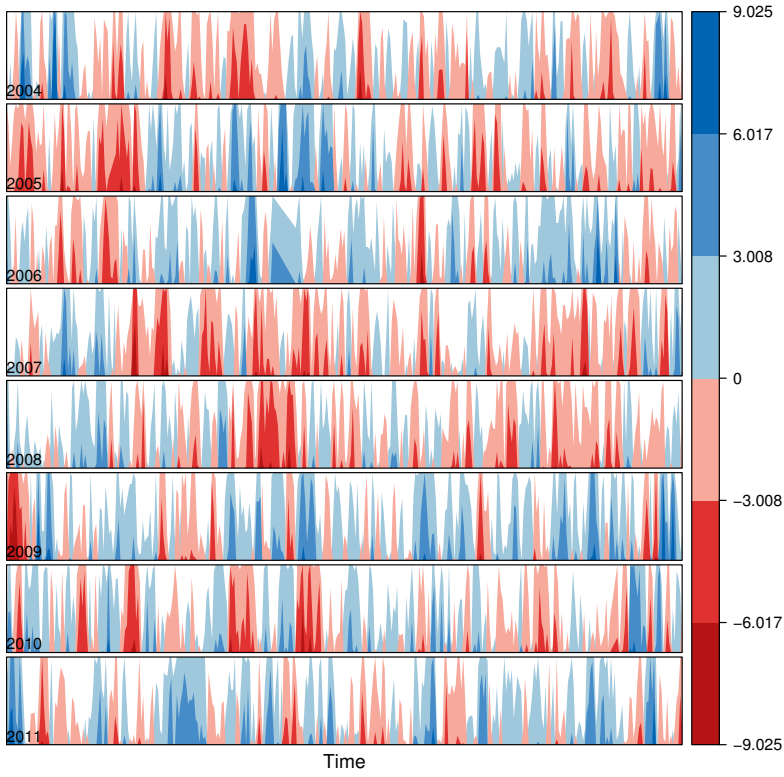

FIGURE 3.8: Horizon graph displaying differences between a daily temperature time series and its long-term average.

age while 2005 and 2009 did the contrary. Year 2007 was frequently below the long-term average but 2011 was more similar to that reference.

```
years <- unique(format(timeIndex, '%Y'))

horizonplot(diffTa, cut=list(n=8, overlap=0),
          colorkey=TRUE, layout=c(1, 8),
          scales=list(draw=FALSE, y=list(relation='same')),
          origin=0, strip.left=FALSE) +
   layer(grid.text(years[panel.number()], x = 0, y = 0.1,
                gp=gpar(cex=0.8),
                just = "left"))
```

A different approach to display this information is to produce a level plot displaying the time series using parts of its time index as independent and conditioning variables[2]. The following code displays the differences with the day of month on the horizontal axis and the year on the vertical axis, with a different panel for each month number. Therefore, each cell of Figure 3.9 corresponds to a certain day of the time series. If you compare this figure with the horizon plot, you will find the same previous findings but revealed now in more detail. On the other hand, while the horizon plot of Figure 3.8 clearly displays the yearly evolution, the combination of variables of the level plot focuses on the comparison between years in a certain month.

```
year <- function(x)as.numeric(format(x, '%Y'))
day <- function(x)as.numeric(format(x, '%d'))
month <- function(x)as.numeric(format(x, '%m'))
```

```
myTheme <- modifyList(custom.theme(region=brewer.pal(9, 'RdBu')),
                            list(
                              strip.background=list(col='gray'),
                              panel.background=list(col='gray')))

maxZ <- max(abs(diffTa))

levelplot(diffTa ~ day(timeIndex) * year(timeIndex) | factor(month(
    timeIndex)),
        at=pretty(c(-maxZ, maxZ), n=8),
        colorkey=list(height=0.3),
        layout=c(1, 12), strip=FALSE, strip.left=TRUE,
        xlab='Day', ylab='Month',
        par.settings=myTheme)
```

The ggplot version of the Figure 3.9 requires a data.frame with the day, year, and month arranged in different columns.

```
df <- data.frame(Vals = diffTa,
             Day = day(timeIndex),
             Year = year(timeIndex),
             Month = month(timeIndex))
```

The values (Vals column of this data.frame) are displayed as a level plot thanks to the geom_raster function.

[2]This approach was inspired by the strip function of the metvurst package https://metvurst.wordpress.com/2013/03/04/visualising-large-amounts-of-hourly-environmental-data-2/.

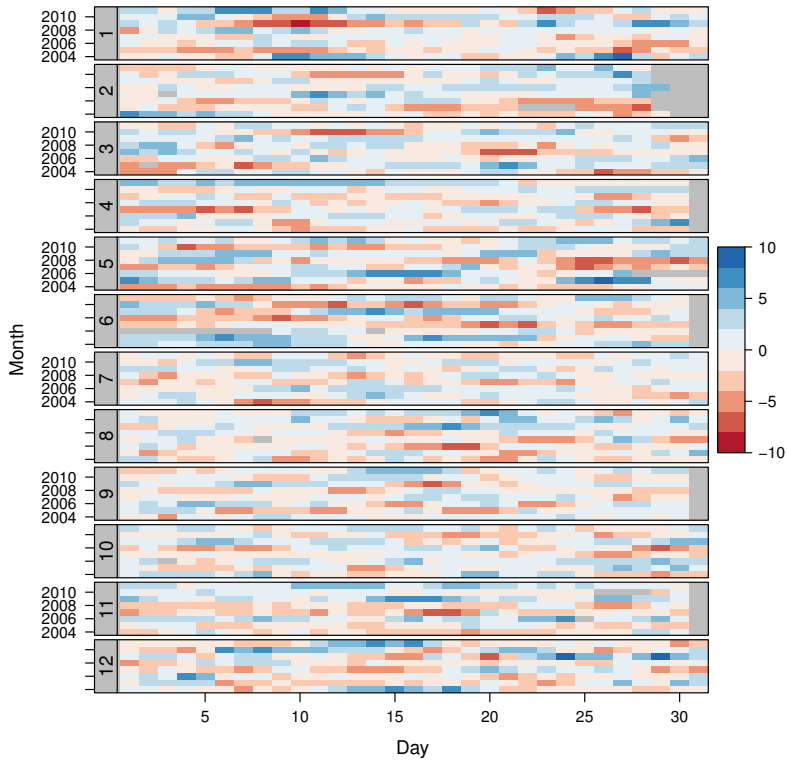

FIGURE 3.9: Level plot of differences between a daily temperature time series and its long-term average.

```
library(scales)
## The packages scales is needed for the pretty_breaks function.

ggplot(data = df,
      aes(fill = Vals,
          x = Day,
          y = Year)) +
   facet_wrap(~ Month, ncol = 1, strip.position = 'left') +
   scale_y_continuous(breaks = pretty_breaks()) +
   scale_fill_distiller(palette = 'RdBu', direction = 1) +
   geom_raster() +
   theme(panel.grid.major = element_blank(),
```

```
        panel.grid.minor = element_blank())
```

3.3 Stacked Graphs

If the variables of a multivariate time series can be summed to produce a meaningful global variable, they may be better displayed with stacked graphs. For example, the information on unemployment in the United States provides data of unemployed persons by industry and class of workers, and can be summed to give a total unemployment time series.

```
load('data/unemployUSA.RData')
```

The time series of unemployment can be directly displayed with the `xyplot.zoo` method (Figure 3.10).

```
xyplot(unemployUSA, superpose=TRUE, par.settings=custom.theme,
      auto.key=list(space='right'))
```

This graphical output is not very useful: the legend is confusing, with too many items; the vertical scale is dominated by the largest series, with several series buried in the lower part of the scale; the trend, variations and structure of the total and individual contributions cannot be deduced from this graph.

A suitable improvement is to display the multivariate time series as a set of stacked colored polygons to follow the macro/micro principle proposed by Tufte (Tufte 1990): Show a collection of individual time series and also display their sum. A traditional stacked graph is easily obtained with `geom_area` (Figure 3.11):

```
library(scales) ## scale_x_yearmon needs scales::pretty_breaks
autoplot(unemployUSA, facets=NULL, geom='area') +
   geom_area(aes(fill=Series)) +
   scale_x_yearmon()
```

Traditional stacked graphs have their bottom on the x-axis which makes the overall height at each point easy to estimate. On the other hand, with this layout, individual layers may be difficult to distinguish. The *ThemeRiver* (Havre et al. 2002) (also named *streamgraph* in (Byron and Wattenberg 2008)) provides an innovative layout method in which layers are symmetrical around the x-axis at their center. At a glance, the pattern of the global sum and individual variables, their contribution to conform the global sum, and the interrelation between variables can be perceived.

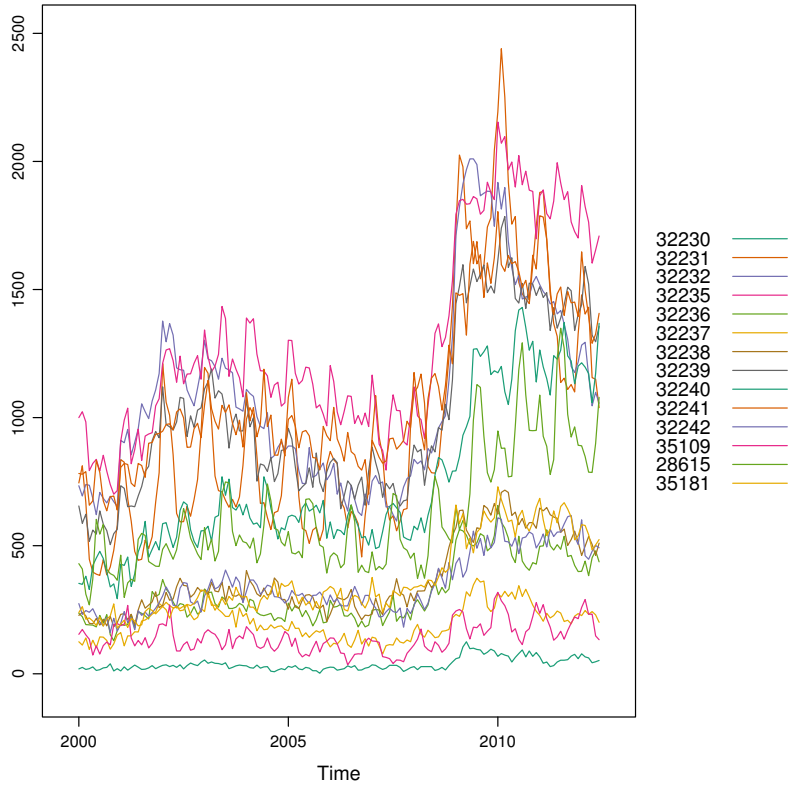

FIGURE 3.10: Time series of unemployment with `xyplot` using the default panel function.

FIGURE 3.11: Time series of unemployment with stacked areas using `geom_area`.

I have defined a panel and prepanel functions[3] to implement a ThemeRiver with `xyplot`. The result is displayed in Figure 3.12 with a vertical line to indicate one of main milestones of the financial crisis, whose effect on the overall unemployment results is clearly evident.

```
library(colorspace)
## We will use a qualitative palette from colorspace
nCols <- ncol(unemployUSA)
pal <- rainbow_hcl(nCols, c=70, l=75, start=30, end=300)
myTheme <- custom.theme(fill=pal, lwd=0.2)

sep2008 <- as.numeric(as.yearmon('2008-09'))

xyplot(unemployUSA, superpose=TRUE, auto.key=FALSE,
      panel=panel.flow, prepanel=prepanel.flow,
      origin='themeRiver', scales=list(y=list(draw=FALSE)),
      par.settings=myTheme) +
   layer(panel.abline(v=sep2008, col='gray', lwd=0.7))
```

This figure can help answer several questions. For example:

- What is the industry or class of worker with the lowest/highest unemployment figures during this time period?
- What is the industry or class of worker with the lowest/highest unemployment increases due to the financial crisis?
- There are a number of local maxima and minima of the total unemployment numbers. Are all the classes contributing to the maxima/minima? Do all the classes exhibit the same fluctuation behavior as the global evolution?

More questions and answers can be found in the "Current Employment Statistics" reports from the Bureau of Labor Statistics[4].

3.3.1 Panel and Prepanel Functions to Implement the ThemeRiver with `xyplot`

The `xyplot` function displays information according to the class of its first argument (methods) and to the `panel` function. We will use the `xyplot.zoo` method (equivalent to the `xyplot.ts` method) with a new custom `panel` function. This new panel function has four main arguments,

[3] The code of these panel and prepanel functions is explained in Section 3.3.1.

[4] The March 2012 highlights report is available at http://www.bls.gov/ces/highlights032012.pdf.

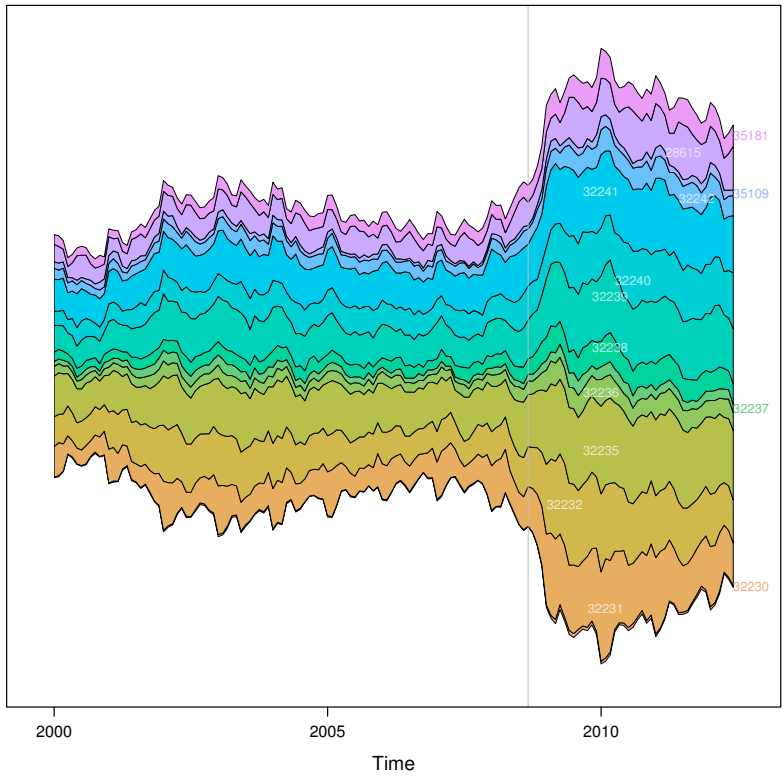

FIGURE 3.12: ThemeRiver of unemployment in the United States.

three of them calculated by `xyplot` (`x`, `y` and `groups`) and a new one, `origin`. Of course, it includes the `...` argument to provide additional arguments.

The first step is to create a `data.frame` with coordinates and with the `groups` factor. The value and number of the levels will be used in the main step of this `panel` function. With this `data.frame` we have to calculate the `y` and `x` coordinates for each group to get a stacked set of polygons.

This `data.frame` is in the *long* format, with a row for each observation, and where the `group` column identifies the variable. Thus, it must be transformed to the *wide* format, with a column for each variable. With the `unstack` function, a new `data.frame` is produced whose columns are defined according to the formula `y ~ groups` and with a row for each time

position. The stack of polygons is the result of the cumulative sum of each row (`apply(yWide, 1, cumsum)`). The origin of this sum is defined with the corresponding `origin` argument: with `themeRiver`, the polygons are arranged in a symmetric way.

Each column of this matrix of cumulative sums defines the `y` coordinate of each variable (where `origin` is now the first variable). The polygon of each variable is between this curve (`iCol+1`) and the one of the previous variable (`iCol`). In order to get a closed polygon, the coordinates of the inferior limit are in reverse order. This new `data.frame` (`Y`) is in the *wide* format, but `xyplot` requires the information in the *long* format: the `y` coordinates of the polygons are extracted from the `values` column of the *long* version of this `data.frame`.

The `x` coordinates are produced in an easier way. Again, `unstack` produces a `data.frame` with a column for each variable and a row for each time position, but now, because the `x` coordinates are the same for the set of polygons, the corresponding vector is constructed directly using a combination of concatenation and repetition.

Finally, the `groups` vector is produced, repeating each element of the columns of the original `data.frame` (`dat$groups`) twice to account for the forward and reverse curves of the corresponding polygon.

The final step before displaying the polygons is to acquire the graphical settings. The information retrieved with `trellis.par.get` is transferred to the corresponding arguments of `panel.polygon`.

Everything is ready for constructing the polygons. With a `for` loop, the coordinates of the corresponding group are extracted from the `x` and `y` vectors, and a polygon is displayed with `panel.polygon`. The labels of each polygon (the `levels` of the original `groups` variable, `groupLevels`) are printed inside the polygon if there is enough room for the text (`hChar>1`) or at the right if the polygon is too small, or if it is the first or last variable of the set. Both the polygons and the labels share the same color (`col[i]`).

```
panel.flow <- function(x, y, groups, origin, ...){
  dat <- data.frame(x=x, y=y, groups=groups)
  nVars <- nlevels(groups)
  groupLevels <- levels(groups)

  ## From long to wide
  yWide <- unstack(dat, y~groups)
  ## Where are the maxima of each variable located? We will use
  ## them to position labels.
  idxMaxes <- apply(yWide, 2, which.max)
```

```
##Origin calculated following Havr.eHetzler.ea2002
if (origin=='themeRiver') origin= -1/2*rowSums(yWide)
else origin=0
yWide <- cbind(origin=origin, yWide)
## Cumulative sums to define the polygon
yCumSum <- t(apply(yWide, 1, cumsum))
Y <- as.data.frame(sapply(seq_len(nVars),
                    function(iCol)c(yCumSum[,iCol+1],
                                    rev(yCumSum[,iCol]))))
names(Y) <- levels(groups)
## Back to long format, since xyplot works that way
y <- stack(Y)$values

## Similar but easier for x
xWide <- unstack(dat, x~groups)
x <- rep(c(xWide[,1], rev(xWide[,1])), nVars)
## Groups repeated twice (upper and lower limits of the polygon)
groups <- rep(groups, each=2)

## Graphical parameters
superpose.polygon <- trellis.par.get("superpose.polygon")
col = superpose.polygon$col
border = superpose.polygon$border
lwd = superpose.polygon$lwd

## Draw polygons
for (i in seq_len(nVars)){
  xi <- x[groups==groupLevels[i]]
  yi <- y[groups==groupLevels[i]]
  panel.polygon(xi, yi, border=border,
              lwd=lwd, col=col[i])
}

## Print labels
for (i in seq_len(nVars)){
  xi <- x[groups==groupLevels[i]]
  yi <- y[groups==groupLevels[i]]
  N <- length(xi)/2
  ## Height available for the label
  h <- unit(yi[idxMaxes[i]], 'native') -
    unit(yi[idxMaxes[i] + 2*(N-idxMaxes[i]) +1], 'native')
  ##...converted to "char" units
  hChar <- convertHeight(h, 'char', TRUE)
```

```
    ## If there is enough space and we are not at the first or
    ## last variable, then the label is printed inside the polygon.
    if((hChar >= 1) && !(i %in% c(1, nVars))){
      grid.text(groupLevels[i],
                xi[idxMaxes[i]],
                (yi[idxMaxes[i]] +
                 yi[idxMaxes[i] + 2*(N-idxMaxes[i]) +1])/2,
                gp = gpar(col='white', alpha=0.7, cex=0.7),
                default.units='native')
    } else {
      ## Elsewhere, the label is printed outside

      grid.text(groupLevels[i],
                xi[N],
                (yi[N] + yi[N+1])/2,
                gp=gpar(col=col[i], cex=0.7),
                just='left', default.units='native')
    }
  }
}
```

With this panel function, `xyplot` displays a set of stacked polygons corresponding to the multivariate time series (Figure 3.13). However, the graphical window is not large enough, and part of the polygons fall out of it. Why?

```
xyplot(unemployUSA, superpose=TRUE, auto.key=FALSE,
       panel=panel.flow, origin='themeRiver',
       par.settings=myTheme, cex=0.4, offset=0,
       scales=list(y=list(draw=FALSE)))
```

The problem is that `lattice` makes a preliminary estimate of the window size using a default `prepanel` function that is unaware of the internal calculations of our new `panel.flow` function. The solution is to define a new `prepanel.flow` function.

The input arguments and first lines are the same as in `panel.flow`. The output is a list whose elements are the limits for each axis (`xlim` and `ylim`), and the sequence of differences (`dx` and `dy`) that can be used for the aspect and banking calculations.

The limits of the x-axis are defined with the range of the time index, while the limits of the y-axis are calculated with the minimum of the first column of `yCumSum` (the origin line) and with the maximum of its last column (the upper line of the cumulative sum).

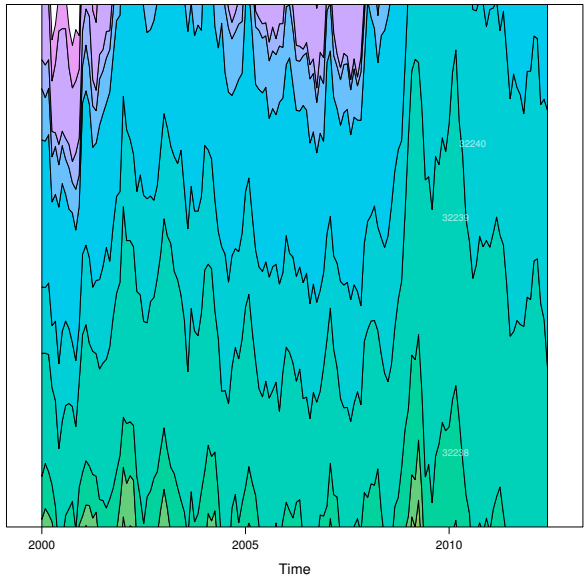

FIGURE 3.13: First attempt of ThemeRiver.

```
prepanel.flow <- function(x, y, groups, origin,...){
  dat <- data.frame(x=x, y=y, groups=groups)
  nVars <- nlevels(groups)
  groupLevels <- levels(groups)
  yWide <- unstack(dat, y~groups)
  if (origin=='themeRiver') origin= -1/2*rowSums(yWide)
  else origin=0
  yWide <- cbind(origin=origin, yWide)
  yCumSum <- t(apply(yWide, 1, cumsum))

  list(xlim=range(x),
       ylim=c(min(yCumSum[,1]), max(yCumSum[,nVars+1])),
       dx=diff(x),
       dy=diff(c(yCumSum[,-1])))
}
```

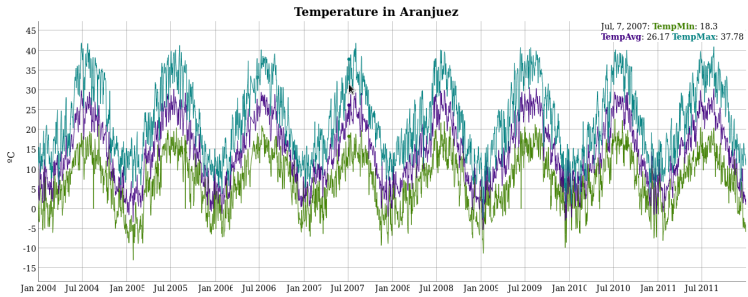

FIGURE 3.14: dygraphs

3.4 Interactive graphics

This section describes the interactive alternatives of the static figures included in the previous sections with four packages: `dygraphs`, `highcharter`, `plotly`, and `gridSVG`.

`dygraphs`, `highcharter`, and `plotly` are `R` interfaces to JavaScript libraries based on the `htmlwidgets` package, while the `gridSVG` package converts a `grid` graphic object into an SVG file.

3.4.1 Dygraphs

The `dygraphs` package is an interface to the `dygraphs` JavaScript library, and provides facilities for charting time-series. It works automatically with `xts` time series objects, or with objects than can be coerced to this class. The result is an interactive graph, where values are displayed according to the mouse position over the time series. Regions can be selected to zoom into a time period. The figure is an snapshot of the interactive graph.

```
library(dygraphs)

dyTemp <- dygraph(aranjuez[, c("TempMin", "TempAvg", "TempMax")],
                  main = "Temperature in Aranjuez",
                  ylab = "ºC")

dyTemp
```

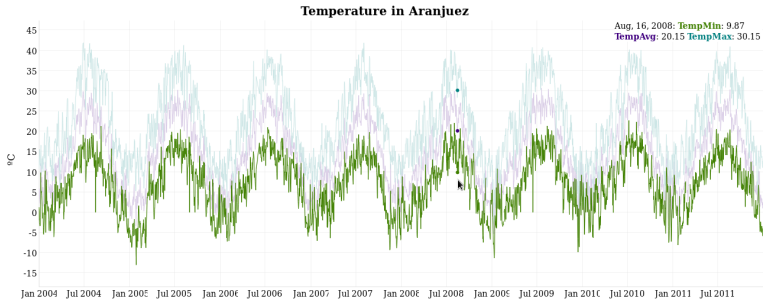

FIGURE 3.15: dygraphs selected

You can customize dygraphs by piping additional commands onto the original dygraph object.

```
dyTemp %>%
   dyHighlight(highlightSeriesBackgroundAlpha = 0.2)
```

```
dyTemp %>%
   dyHighlight(highlightSeriesOpts = list(strokeWidth = 2))
```

```
dygraph(aranjuez[, c("TempMin", "TempAvg", "TempMax")],
      main = "Temperature in Aranjuez",
      ylab = "ºC") %>%
   dySeries(c("TempMin", "TempAvg", "TempMax"),
         label = "Temperature")
```

3.4.2 Highcharter

The `highcharter` package is an interface to the `highcharts` JavaScript library, with a wide spectrum of graphics solutions. Displaying time series with this package can be achieved with the combination of the generic `highchart` function and several calls to the `hc_add_series_xts` function through the pipe %>% operator. Once again, the result is an interactive graph with selection and zoom capabilities. Figure is an snapshot of the interactive graph.

```
library(highcharter)
library(xts)
```

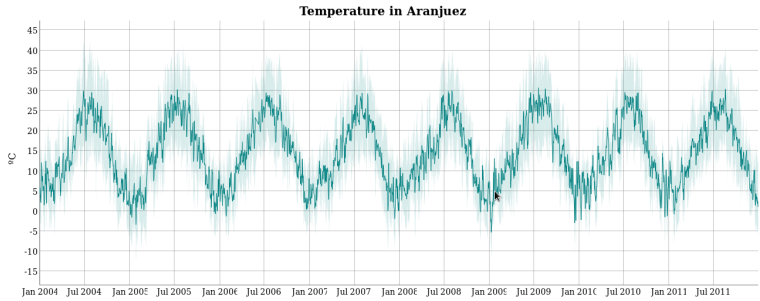

FIGURE 3.16: dygraphs maxmin

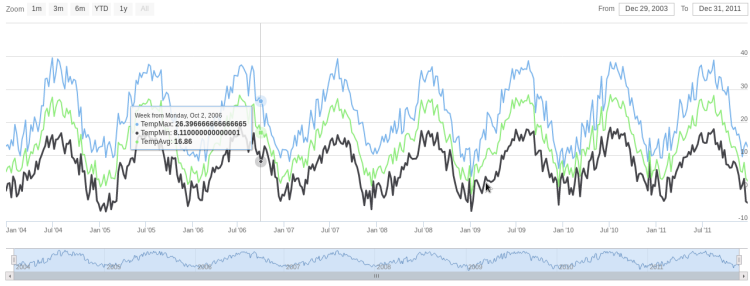

FIGURE 3.17: highcharter

```
aranjuezXTS <- as.xts(aranjuez)

highchart() %>%
   hc_add_series_xts(name = 'TempMax',
                     aranjuezXTS[, "TempMax"]) %>%
   hc_add_series_xts(name = 'TempMin',
                     aranjuezXTS[, "TempMin"]) %>%
   hc_add_series_xts(name = 'TempAvg',
                     aranjuezXTS[, "TempAvg"])
```

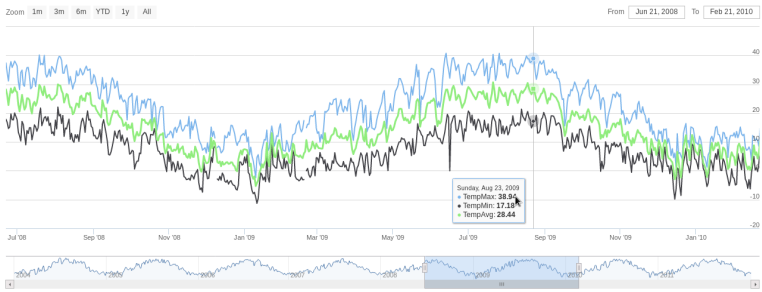

FIGURE 3.18: highcharter zoom

3.4.3 plotly

The `plotly` package is an interface to the `plotly` JavaScript library, also with a wide spectrum of graphics solutions. This package does not provide any function specifically focused on time series. Thus, the time series object has to be transformed in a `data.frame` including a column for the time index. If the `data.frame` is in *wide* format (one column per variable), each variable will be represented with a call to the `add_lines` function. However, if the `data.frame` is in *long* format (a column for values, and a column for variable names) only one call to `add_lines` is required. The next code follows this approach using the combination of `fortify`, to convert the `zoo` object into a `data.frame`, and `melt`, to transform from wide to long format.

```
library(reshape2)

aranjuezDF <- fortify(aranjuez[,
                          c("TempMax",
                            "TempAvg",
                            "TempMin")])

aranjuezDF <- melt(aranjuezDF, id.vars = 'Index')
```

Figure 3.19 is a snapshot of the interactive graphic produce with the generic function `plot_ly` connected with `add_lines` through the pipe operator, %>%.

```
library(plotly)
```

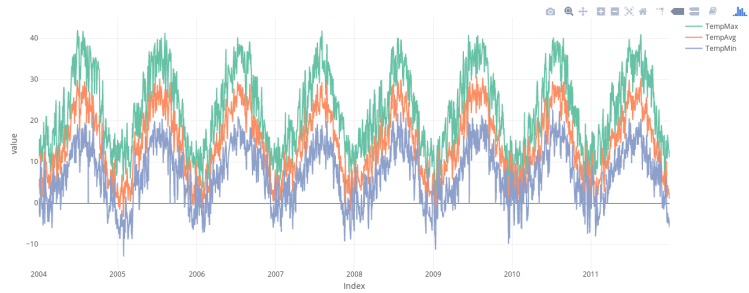

FIGURE 3.19: plotly

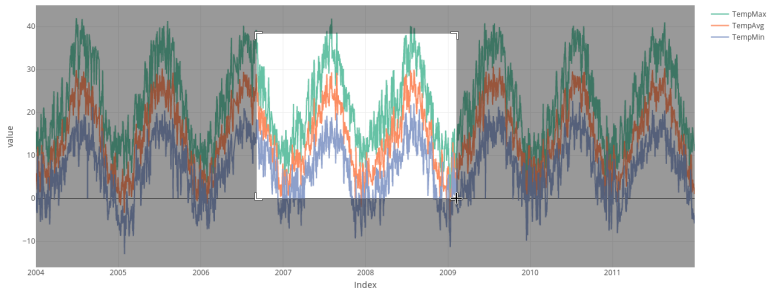

FIGURE 3.20: plotly zoom

```
plot_ly(aranjuezDF) %>%
   add_lines(x = ~ Index,
             y = ~ value,
             color = ~ variable)
```

3.4.4 Interaction with `gridSVG`

The `gridSVG` package provides functions to convert `grid`-based `R` graphics to an SVG format. It provides several functions to add dynamic and interactive capabilities to `R` graphics. In this section we will use `grid.script`, a function to add JavaScript code to a plot.

The first step is to specify which component of the scene will run the JavaScript code. The `grid.ls` function returns a listing of the names of grobs or viewports included in the graphic output: only the lines will be connected with the JavaScript code.

```
library(gridSVG)
## grobs in the graphical output
pNavarra
grobs <- grid.ls(print=FALSE)
## only interested in some of them
nms <- grobs$name[grobs$type == "grobListing"]
idxNames <- grep('lines', nms)
IDs <- nms[idxNames]
```

The second step is to modify each `grob` (graphical object) to add attributes that specify when it will call JavaScript code. For each line identified with the elements of the `IDs` vector and associated to a meteorological station, the `navarra` object is accessed to extract the annual mean value of the daily radiation and the abbreviated name of the corresponding station (`info`). The `grid.garnish` function adds attributes to the `grob` of each line so that when the mouse moves over a `grob`, the line is highlighted and colored in red (`highlight`). When the mouse hovers out of the `grob`, the `hide` function sets back the default values of line width and transparency, but uses the green color to denote that this line has been already visited. In addition, because the browsers display the content of the title attribute with a default tooltip, `grid.garnish` sets this attribute to `info`.

```
for (id in unique(IDs)){
  ## extract information from the data
  ## according to the ID value
  i <- strsplit(id, '\\.')
  i <- sapply(i, function(x)as.numeric(x[5]))
  ## Information to be attached to each line: annual mean of daily
  ## radiation and abbreviated name of the station
  dat <- round(mean(navarra[,i], na.rm=TRUE), 2)
  info <- paste(names(navarra)[i], paste(dat, collapse=','),
              sep=': ')
  ## attach SVG attributes
  grid.garnish(id,
               onmouseover="highlight(evt)",
               onmouseout="hide(evt)",
               title=info)
}
```

These JavaScript functions are included in a script file named `highlight.js` (available at the website of the book). It can be added as an additional object with `grid.script`.

```
grid.script(filename="highlight.js")
```

This script is easy to understand, even without previous JavaScript knowledge:

```
highlight = function(evt){',
    evt.target.setAttribute('opacity', '1');
    evt.target.setAttribute('stroke', 'red');
    evt.target.setAttribute('stroke-width', '1');
}

hide = function(evt){
    evt.target.setAttribute('opacity', '0.3');
    evt.target.setAttribute('stroke', green');
    evt.target.setAttribute('stroke-width', '0.3');
}
```

Finally, `gridToSVG` exports the whole scene to SVG.

```
grid.export('figs/navarraRadiation.svg')
```

A snapshot of the result, as viewed in a browser with a line highlighted, is shown in Figure 3.21. Open the SVG file with your browser, explore it using the horizon graph (Figure 3.6) as a reference, and try to answer the questions raised with that graphic.

3.4.5 streamgraph

The `streamgraph` package[5] creates interactive stream graphs based on the `htmlwidgets` package and the `D3.js` JavaScript library. Its main function, `streamgraph`, requires a `data.frame` as the first argument. Besides, its three next arguments, `key`, `value`, and `date`, makes this function a good candidate to work together with `fortify` and `melt`.

```
## remotes::install_github("hrbrmstr/streamgraph")
library(streamgraph)
library(reshape2)
```

[5]The `streamgraph` package, http://hrbrmstr.github.io/streamgraph/, is not available in CRAN. It can be installed using the `devtools` or the `remotes` package.

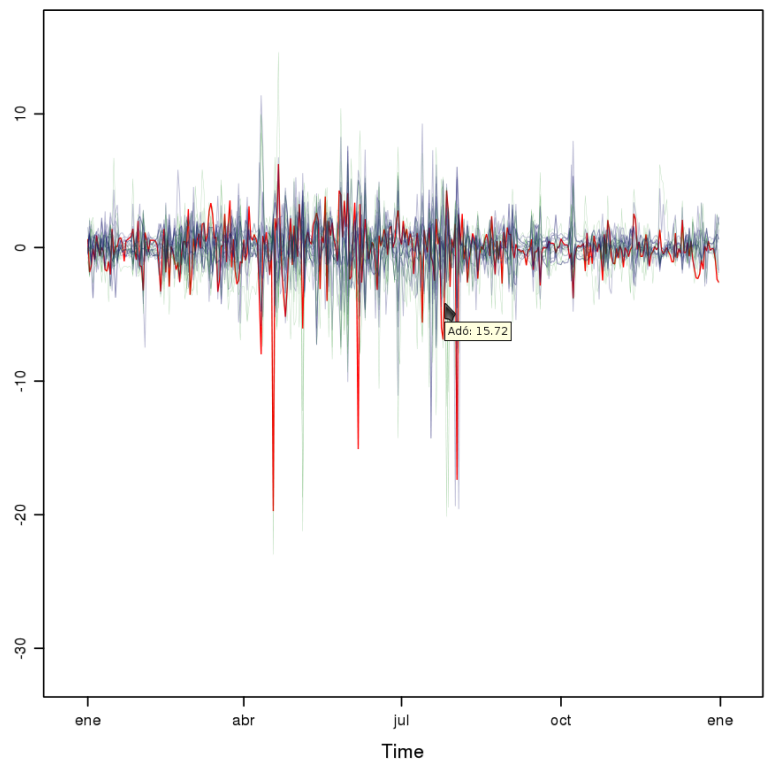

FIGURE 3.21: Snapshot of an SVG graphic produced with `gridSVG`.

```
unemployDF <- fortify(unemployUSA)

unemployDF <- melt(unemployDF,
                id.vars = 'Index')
```

Figures 3.22 and 3.23 are snapshots of the interactive graphic created with the functions `streamgraph`, `sg_axis`, and `sg_fill_brewer`, connected through the pipe operator, `%>%`.

```
streamgraph(unemployDF,
          key = "variable",
          value = "value",
          date = "Index") %>%
   sg_axis_x(1, "year", "%Y") %>%
```

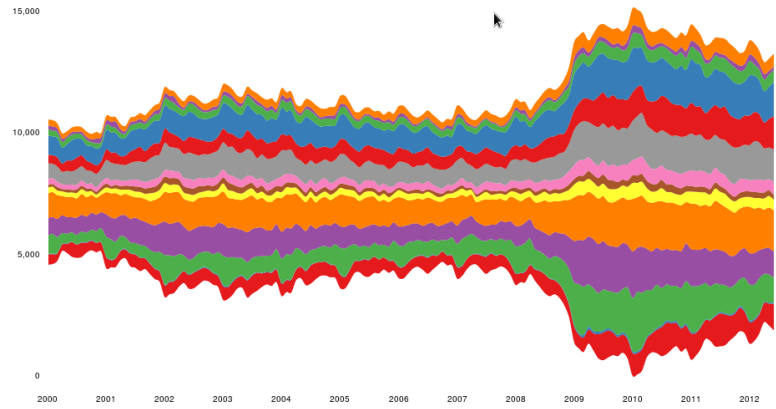

FIGURE 3.22: Streamgraph created with the `streamgraph` package, without selection.

```
  sg_fill_brewer("Set1")
```

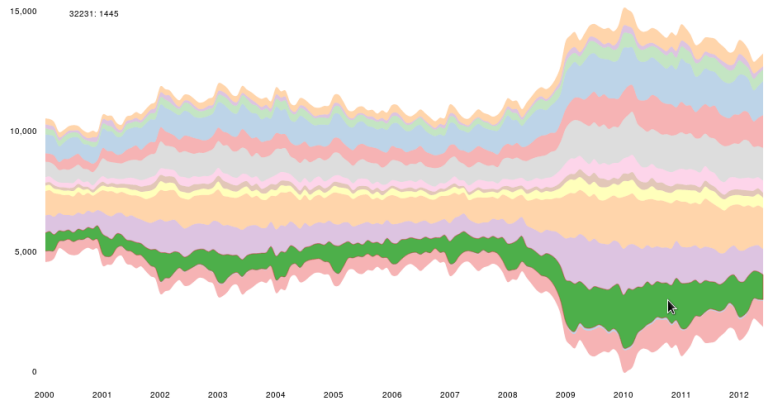

FIGURE 3.23: Streamgraph created with the `streamgraph` package, with a selection.

Chapter 4

Time as a Conditioning or Grouping Variable

In Section 3.1 we learned to display the time evolution of multiple time series with different scales. But, what if instead of displaying the time evolution we want to confront the variables between them? Section 4.1 proposes the scatterplot matrix solution with time as a grouping variable. Section 4.2 uses an enhanced scatterplot with time as a conditioning variable. Section 4.1.1 includes a digression about the hexagonal binning for large datasets.

4.1 Scatterplot Matrix: Time as a Grouping Variable

The scatterplot matrices are based on the technique of small multiples (Tufte 1990): small, thumbnail-sized representations of multiple images displayed all at once, which allows the reader to immediately, and in parallel, compare the inter-frame differences. A scatterplot matrix is a display of all pairwise bivariate scatterplots arranged in a $p \times p$ matrix for p variables. Each subplot shows the relation between the pair of variables at the intersection of the row and column indicated by the variable names in the diagonal panels (Friendly and Denis 2005).

This graphical tool is implemented in the `splom` function. The following code displays the relation between the set of meteorological variables

using a sequential palette from the ColorBrewer catalog (`RbBu`, with black added to complete a twelve-color palette) to encode the month. The order of colors of this palette is chosen in order to display summer months with intense colors and to distinguish between the first and second half of the year with red and blue, respectively (Figure 4.1).

```
load('data/aranjuez.RData')

## Red-Blue palette with black added (12 colors)
colors <- c(brewer.pal(n=11, 'RdBu'), '#000000')
## Rearrange according to months (darkest for summer)
colors <- colors[c(6:1, 12:7)]
```

```
splom(~as.data.frame(aranjuez),
      groups=format(index(aranjuez), '%m'),
     auto.key=list(space='right',
        title='Month', cex.title=1),
     pscale=0, varname.cex=0.7, xlab='',
      par.settings=custom.theme(symbol=colors,
         pch=19), cex=0.3, alpha=0.1)
```

ggplot version

```
library(GGally)

df <- as.data.frame(aranjuez)
df$Month <- format(index(aranjuez), '%m')

p <- ggpairs(df,
          columns = 1:10, ## Do not include "Month"
          upper = list(continuous = "points"),
          mapping = aes(colour = Month, alpha = 0.1))
```

Let's explore Figure 4.1. For example,

- The highest values of ambient temperature (average, maximum, and mimimun), solar radiation, and evotranspiration can be found during the summer.
- These variables are almost linearly related. The relation between radiation and temperature is different during both halves of the year (red and blue regions can be easily distinguished).
- The humidity reaches its highest values during winter without appreciable differences between the first and second half of the year.

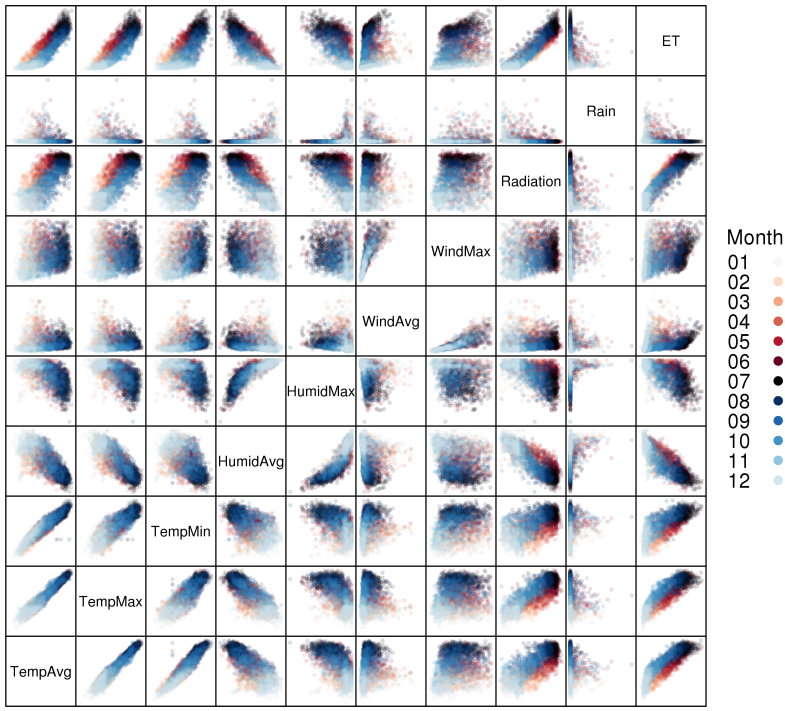

FIGURE 4.1: Scatter plot matrix of the collection of meteorological time series of the Aranjuez station.

The temperature and humidity may be related with an exponential function.

A bit of interactivity can be added to this plot with the identification of some points. This task is easy with `panel.link.splom`. The points are selected via mouse clicks (and highlighted in green). Clicks other than left-clicks terminate the procedure. The output of this function is the index of chosen points.

```
trellis.focus('panel', 1, 1)
idx <- panel.link.splom(pch=13, cex=0.6, col='green')
aranjuez[idx,]
```

4.1.1 Hexagonal Binning

For large datasets, the display of a large number of points in a scatterplot produces hidden point density, long computation times, and slow displays. These problems can be circumvented with the estimation and representation of points densities. A common encoding uses gray scales, pseudo colors or partial transparency. An improved scheme encodes density as the size of hexagon symbols inscribed within hexagonal binning regions (D. B. Carr et al. 1987).

The `hexbin` package (D. Carr, Lewin-Koh, and Maechler 2013) includes several functions for hexagonal binning. The `panel.hexbinplot` is a good substitute for the default panel function. In addition, our first attempt with `splom` can be improved with several modifications (Figure 4.2):

- The scale's ticks and labels are suppressed with `pscale=0`.
- The panels of the lower part of the matrix (`lower.panel`) will include a locally weighted scatterplot smoothing (loess) with `panel.loess`.
- The diagonal panels (`diag.panel`) will display the kernel density estimate of each variable. The `density` function computes this estimate. The result is adjusted to the panel limits (calculated with `current.panel.limits`). The kernel density is plotted with `panel.lines` and the `diag.panel.splom` function completes the content of each diagonal panel.
- The point density is encoded with the palette `BTC` (ligther colors for high density values and darker colors for almost empty regions, with a gradient of blue hues for intermediate values).

```
library(hexbin)

splom(~as.data.frame(aranjuez),
         panel=panel.hexbinplot, xlab='',
         colramp=BTC,
         diag.panel = function(x, ...){
           yrng <- current.panel.limits()$ylim
           d <- density(x, na.rm=TRUE)
           d$y <- with(d, yrng[1] + 0.95 * diff(yrng) * y / max(y))
           panel.lines(d)
           diag.panel.splom(x, ...)
         },
         lower.panel = function(x, y, ...){
```

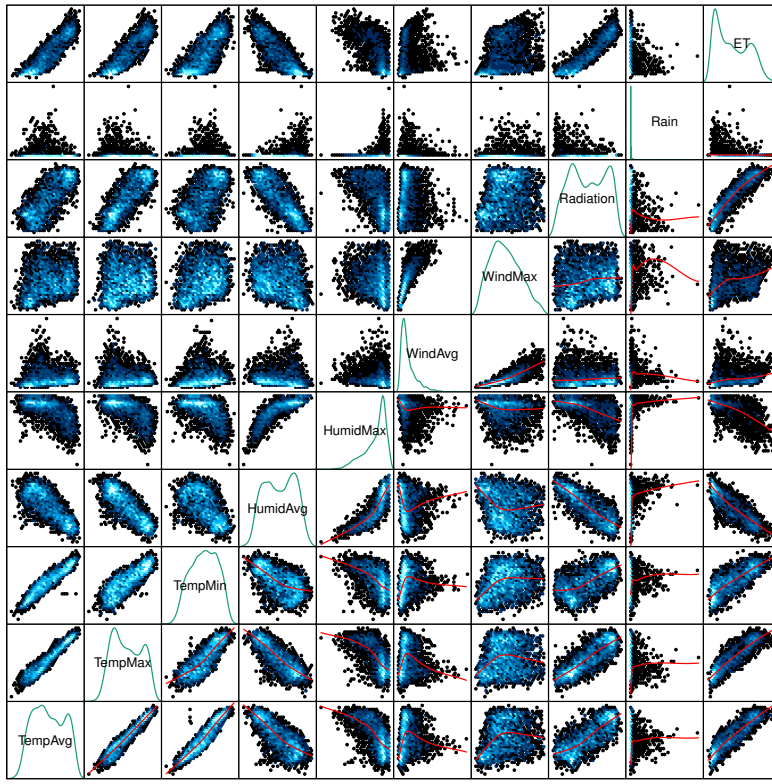

FIGURE 4.2: Scatterplot matrix of the collection of meteorological time series of the Aranjuez station using hexagonal binning.

```
        panel.hexbinplot(x, y, ...)
        panel.loess(x, y, ..., col = 'red')
    },
    pscale=0, varname.cex=0.7
    )
```

A drawback of the matrix of scatterplots with hexagonal binning is that each panel is drawn independently, so it is impossible to compute a common color key for all of them. In other words, two cells with exactly the same color in different panels encode different point densities.

It is possible to display a reduced set of variables against another one and generate a common color key using the `hexbinplot` function. First,

the dataset must be reshaped from the wide format (one colum for each variable) to the long format (only one column for the values with one row for each observation).

The `reshape` function needs several arguments to perform the conversion. The most important is the `data.frame` to be transformed. Then there are the names of variables to be mapped to a single variable in the long dataset (the three ambient temperatures). The name of this variable can be set with `v.names`. Finally, `timevar` is the name of the column in long format that differentiates multiple observations from the same variable. The values of this column are defined with the `times` argument.

```
aranjuezDF <- data.frame(aranjuez,
                     month=format(index(aranjuez), '%m'))
aranjuezRshp <- reshape(aranjuezDF, direction='long',
                    varying=list(names(aranjuez)[1:3]),
                    v.names='Temperature',
                    times=names(aranjuez)[1:3],
                    timevar='Statistic')
```

```
head(aranjuezRshp)
```

The `hexbinplot` displays this dataset with a different panel for each type of temperature (average, maximum, and minimum) but with a common color key encoding the point density (Figure 4.3). Now, two cells with the same color in different panels encode the same value.

```
hexbinplot(Radiation~Temperature|Statistic, data=aranjuezRshp,
         layout=c(1, 3), colramp=BTC) +
   layer(panel.loess(..., col = 'red'))
```

The ggplot2 version uses `stat_binhex`.

```
ggplot(data=aranjuezRshp, aes(Temperature, Radiation)) +
   stat_binhex(ncol=1) +
   stat_smooth(se=FALSE, method='loess', col='red') +
   facet_wrap(~Statistic, ncol=1) +
   theme_bw()
```

4.2 Scatterplot with Time as a Conditioning Variable

After discussing the hexagonal binning, let's recover the time variable. Figure 4.1 uses colors to encode months. Instead, we will now display separate scatterplots with a panel for each month. In addition, the statistic

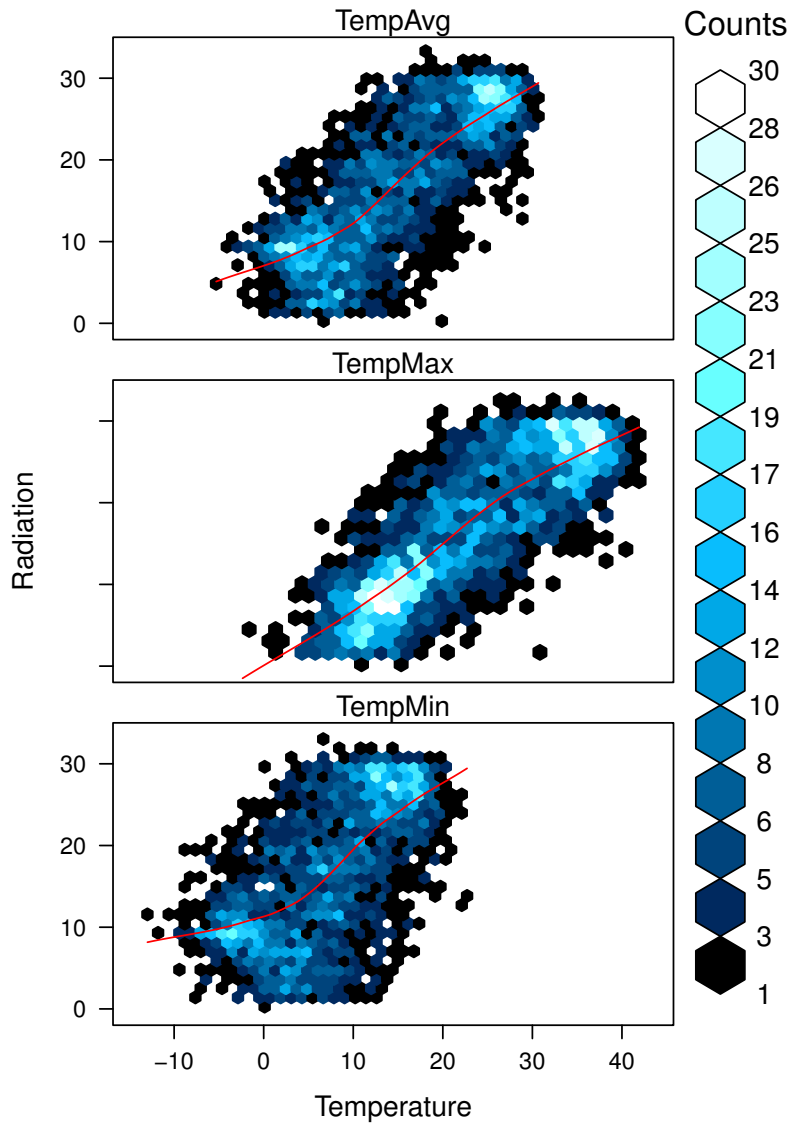

FIGURE 4.3: Scatterplot with hexagonal binning of temperature versus solar radiation using data of the Aranjuez station (`lattice` version).

type (average, maximum, minimum) is included as an additional conditioning variable.

This matrix of panels can be displayed with `ggplot` using `facet_grid`. The code of Figure 4.4 uses partial transparency to cope with overplotting, small horizontal and vertical segments (`geom_rug`) to display points density on both variables, and a smooth line in each panel.

```
ggplot(data=aranjuezRshp, aes(Radiation, Temperature)) +
   facet_grid(Statistic ~ month) +
   geom_point(col='skyblue4', pch=19, cex=0.5, alpha=0.3) +
   geom_rug() +
   stat_smooth(se=FALSE, method='loess', col='indianred1', lwd
       =1.2) +
   theme_bw()
```

The version with `lattice` needs the `useOuterStrips` function from the `latticeExtra` package, which prints the names of the conditioning variables on the top and left outer margins (Figure 4.5).

```
useOuterStrips(xyplot(Temperature ~ Radiation | month * Statistic,
                  data=aranjuezRshp,
                  between=list(x=0),
                  col='skyblue4', pch=19,
                  cex=0.5, alpha=0.3)) +
   layer({
      panel.rug(..., col.line='indianred1', end=0.05, alpha=0.6)
      panel.loess(..., col='indianred1', lwd=1.5, alpha=1)
   })
```

These figures show the typical seasonal behavior of solar radiation and ambient temperature. Additionally, it displays in more detail the same relations between radiation and temperature already discussed with Figure 4.3.

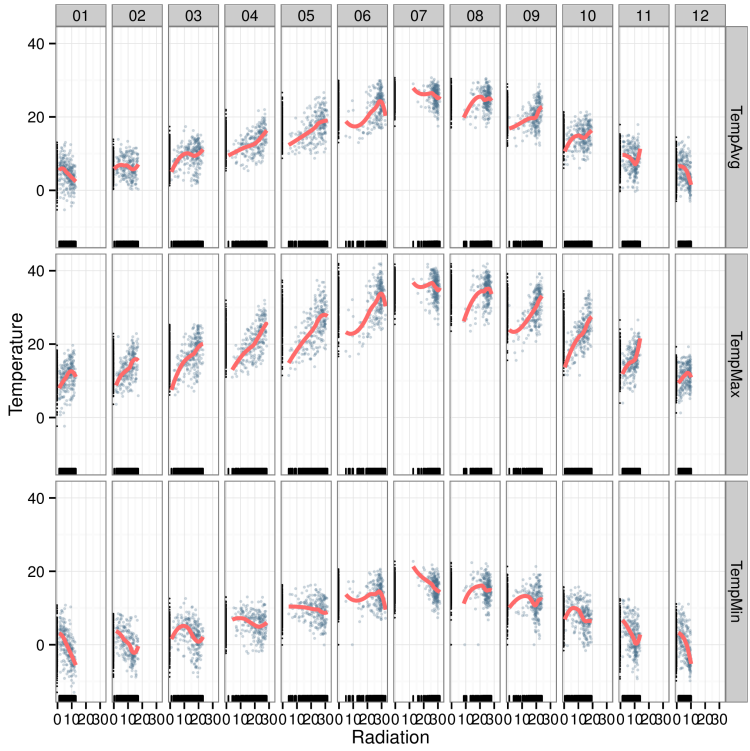

FIGURE 4.4: Scatterplot of temperature versus solar radiation for each month using data of the Aranjuez station (`ggplot2` version).

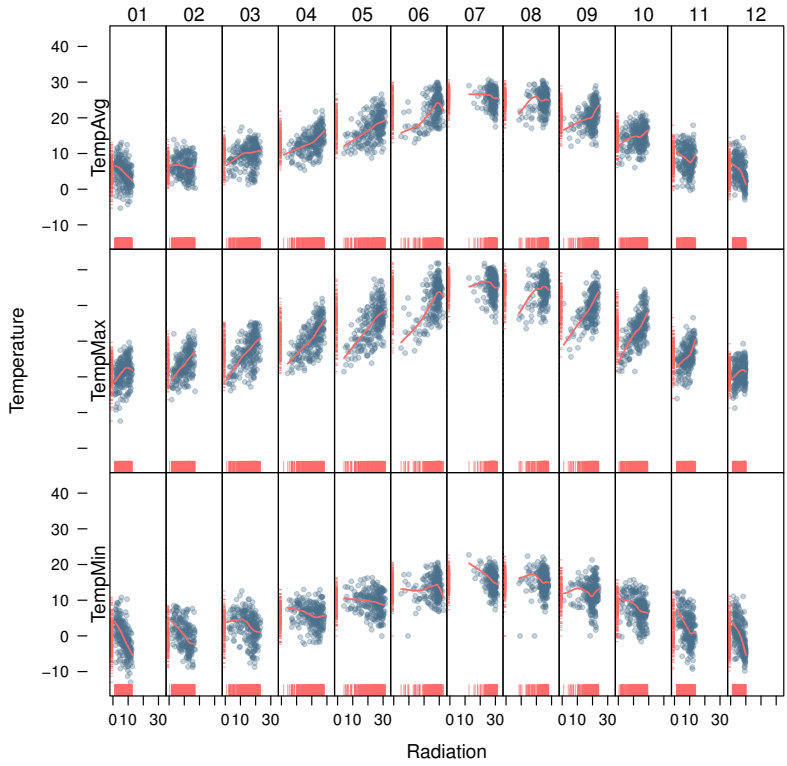

FIGURE 4.5: Scatterplot of temperature versus solar radiation for each month using data of the Aranjuez station (lattice version).

Chapter 5

Time as a Complementary Variable

Gapminder [1] is an independent foundation based in Stockholm, Sweden. Its mission is "to debunk devastating myths about the world by offering free access to a fact-based world view." They provide free online tools, data, and videos "to better understand the changing world." The initial development of Gapminder was the Trendalyzer software, used by Hans Rosling in several sequences of his documentary "The Joy of Stats."

The information visualization technique used by Trendalyzer is an interactive bubble chart. By default it shows five variables: two numeric variables on the vertical and horizontal axes, bubble size and color, and a time variable that may be manipulated with a slider. The software uses brushing and linking techniques for displaying the numeric value of a highlighted country.

This software was acquired by Google in 2007, and is now available as a Motion Chart gadget and as the Public Data Explorer.

In this chapter, time will be used as a complementary variable which adds information to a graph where several variables are confronted. We will illustrate this approach with the evolution of the relationship between Gross National Income (GNI) and carbon dioxide (CO_2) emissions for a set

[1] http://www.gapminder.org/

of countries extracted from the database of the World Bank Open Data. We will try several solutions to display the relationship between CO_2 emissions and GNI over the years using time as a complementary variable. The final method will produce an animated plot resembling the Trendalyzer solution.

5.1 Polylines

```
load('data/CO2.RData')
```

Our first approach is to display the entire data in a panel with a scatterplot using country names as the grouping factor. Points of each country are connected with polylines to reveal the time evolution (Figure 5.1).

```
## lattice version
xyplot(GNI.capita ~ CO2.capita, data=CO2data,
      xlab="Carbon dioxide emissions (metric tons per capita)",
      ylab="GNI per capita, PPP (current international $)",
      groups=Country.Name, type='b')
```

```
## ggplot2 version
ggplot(data=CO2data, aes(x=CO2.capita, y=GNI.capita,
         color=Country.Name)) +
   xlab("Carbon dioxide emissions (metric tons per capita)") +
   ylab("GNI per capita, PPP (current international $)") +
   geom_point() + geom_path() + theme_bw()
```

Three improvements can be added to this graphical result:

1. Define a better palette to enhance visual discrimination between countries.
2. Display time information with labels to show year values.
3. Label each polyline with the country name instead of a legend.

5.2 Choosing Colors

The `Country.Name` categorical variable will be encoded with a qualitative palette, namely the first five colors of `Set1` palette[2] from the `RColorBrewer` package (Neuwirth 2011). Because there are more countries than colors,

[2] http://colorbrewer2.org/

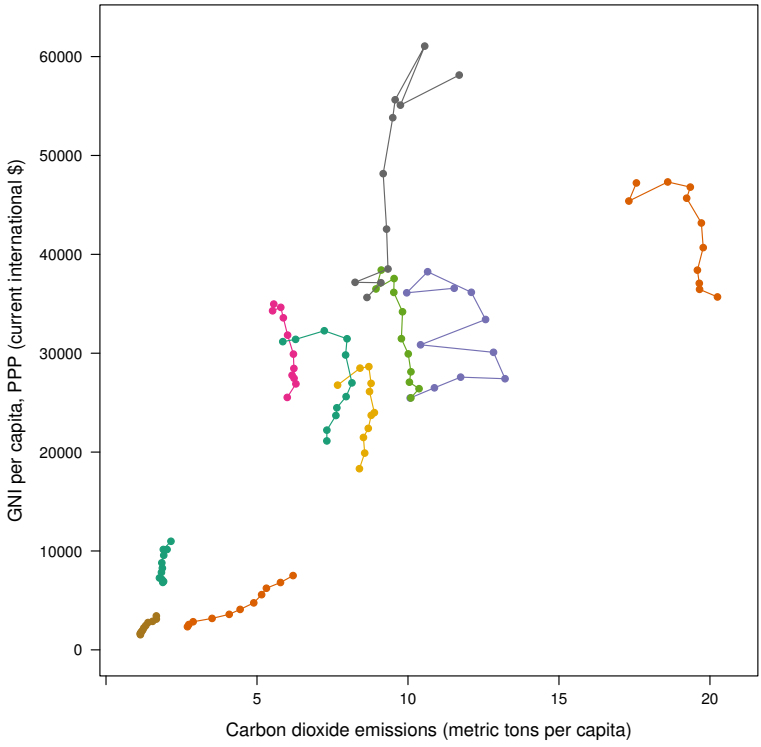

FIGURE 5.1: GNI per capita versus CO_2 emissions per capita (`lattice` version).

we have to repeat some colors to complete the number of levels of the variable `Country.Name`. The result is a palette with non-unique colors, and thus some countries will share the same color. This is not a problem because the curves will be labeled, and countries with the same color will be displayed at enough distance.

```
library(RColorBrewer)

nCountries <- nlevels(CO2data$Country.Name)
pal <- brewer.pal(n=5, 'Set1')
pal <- rep(pal, length = nCountries)
```

Adjacent colors of this palette are chosen to be easily distinguishable. Therefore, the connection between colors and countries must be in such a way that nearby lines are encoded with adjacent colors of the palette.

A simple approach is to calculate the annual average of the variable to be represented along the x-axis (`CO2.capita`), and extract colors from the palette according to the order of this value.

```
## Rank of average values of CO2 per capita
CO2mean <- aggregate(CO2.capita ~ Country.Name, data=CO2data, FUN=
    mean)
palOrdered <- pal[rank(CO2mean$CO2.capita)]
```

A more sophisticated solution is to use the ordered results of a hierarchical clustering of the time evolution of the CO_2 per capita values (Figure 5.2). The data is extracted from the original CO_2 `data.frame`.

```
CO2capita <- CO2data[, c('Country.Name', 'Year', 'CO2.capita')]
CO2capita <- reshape(CO2capita, idvar='Country.Name', timevar='Year
    ', direction='wide')
hCO2 <- hclust(dist(CO2capita[, -1]))

oldpar <- par(mar=c(0, 2, 0, 0) + .1)
plot(hCO2, labels=CO2capita$Country.Name,
    xlab='', ylab='', sub='', main='')
par(oldpar)
```

The colors of the palette are assigned to each country with `match`, which returns a vector of the positions of the matches of the country names in alphabetical order in the country names ordered according to the hierarchical clustering.

```
idx <- match(levels(CO2data$Country.Name),
          CO2capita$Country.Name[hCO2$order])
palOrdered <- pal[idx]
```

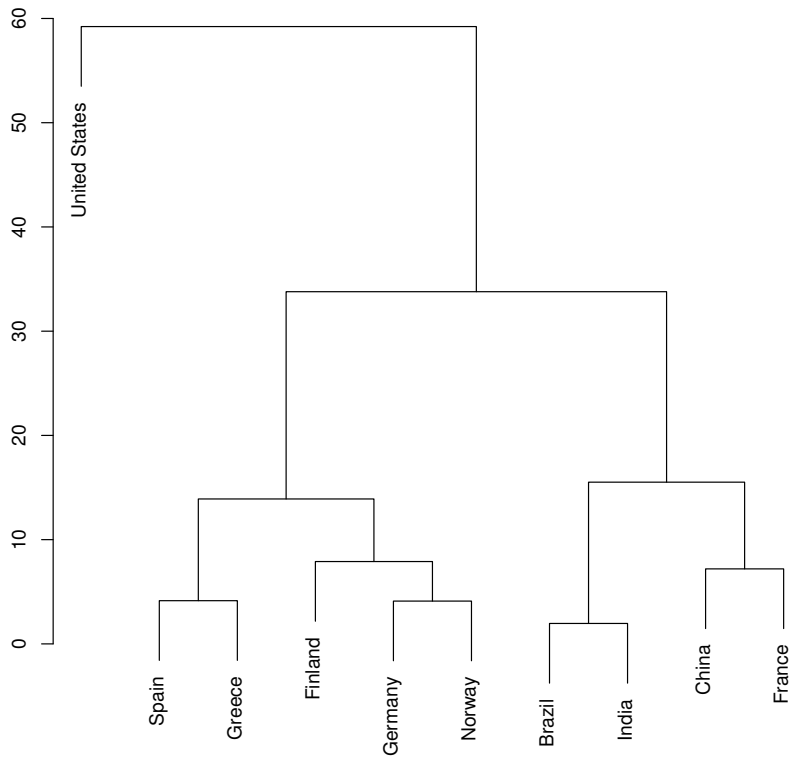

FIGURE 5.2: Hierarchical clustering of the time evolution of CO_2 per capita values.

It must be highlighted that this palette links colors with the levels of `Country.Name` (country names in alphabetical order), which is exactly what the `groups` argument provides. The following code produces a curve for each country using different colors to distinguish them.

```
## simpleTheme encapsulates the palette in a new theme for xyplot
myTheme <- simpleTheme(pch=19, cex=0.6, col=palOrdered)

pCO2.capita <- xyplot(GNI.capita ~ CO2.capita,
                  xlab="Carbon dioxide emissions (metric tons per
                      capita)",
                  ylab="GNI per capita, PPP (current international
                      $)",
```

```
                groups=Country.Name, data=CO2data,
                par.settings=myTheme,
                type='b')
```

```
gCO2.capita <- ggplot(data=CO2data, aes(x=CO2.capita, y=GNI.capita,
                color=Country.Name)) +
   geom_point() + geom_path() +
   scale_color_manual(values=palOrdered, guide=FALSE) +
   xlab('CO2 emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

5.3 Labels to Show Time Information

This result can be improved with labels displaying the years to show the time evolution. A panel function with `panel.text` to print the year labels and `panel.superpose` to display the lines for each group is a solution. In the panel function, `subscripts` is a vector with the integer indices representing the rows of the `data.frame` to be displayed in the panel.

```
xyplot(GNI.capita ~ CO2.capita,
     xlab="Carbon dioxide emissions (metric tons per capita)",
     ylab="GNI per capita, PPP (current international $)",
     groups=Country.Name, data=CO2data,
     par.settings=myTheme,
     type='b',
     panel=function(x, y, ..., subscripts, groups){
       panel.text(x, y, ...,
                labels=CO2data$Year[subscripts],
                pos=2, cex=0.5, col='gray')
       panel.superpose(x, y, subscripts, groups,...)
     }
     )
```

The same result with a clearer code is obtained with the combination of `+.trellis`, `glayer_` and `panel.text`. Using `glayer_` instead of `glayer`, we ensure that the labels are printed below the lines.

```
pCO2.capita <- pCO2.capita +
   glayer_(panel.text(..., labels=CO2data$Year[subscripts],
                pos=2, cex=0.5, col='gray'))
```

```
gCO2.capita <- gCO2.capita + geom_text(aes(label=Year),
                                colour='gray',
                                size=2.5,
                                hjust=0, vjust=0)
```

5.4 Country Names: Positioning Labels

The common solution to link each curve with the group value is to add a legend. However, a legend can be confusing with too many items. In addition, the reader must carry out a complex task: Choose the line, memorize its color, search for it in the legend, and read the country name.

A better approach is to label each line using nearby text with the same color encoding. A suitable method is to place the labels close to the end of each line (Figure 5.3). Labels are placed with the `panel.pointLabel` function from the `maptools` package. This function use optimization routines to find locations without overlaps.

```
library(maptools)
## group.value provides the country name; group.number is the
## index of each country to choose the color from the palette.
pCO2.capita +
   glayer(panel.pointLabel(mean(x), mean(y),
                       labels= group.value,
                       col=pal0rdered[group.number],
                       cex=.8,
                       fontface=2, fontfamily='Palatino'))
```

However, this solution does not solve the overlapping between labels and lines. The package `directlabels` (Hocking 2013) includes a wide repertory of positioning methods to cope with this problem. The main function, `direct.label`, is able to determine a suitable method for each plot, although the user can choose a different method from the collection or even define a custom method. For the `pCO2.capita` object, I have obtained the best results with `extreme.grid` (Figure 5.4).

```
library(directlabels)
direct.label(pCO2.capita, method='extreme.grid')
```

```
direct.label(gCO2.capita, method='extreme.grid')
```

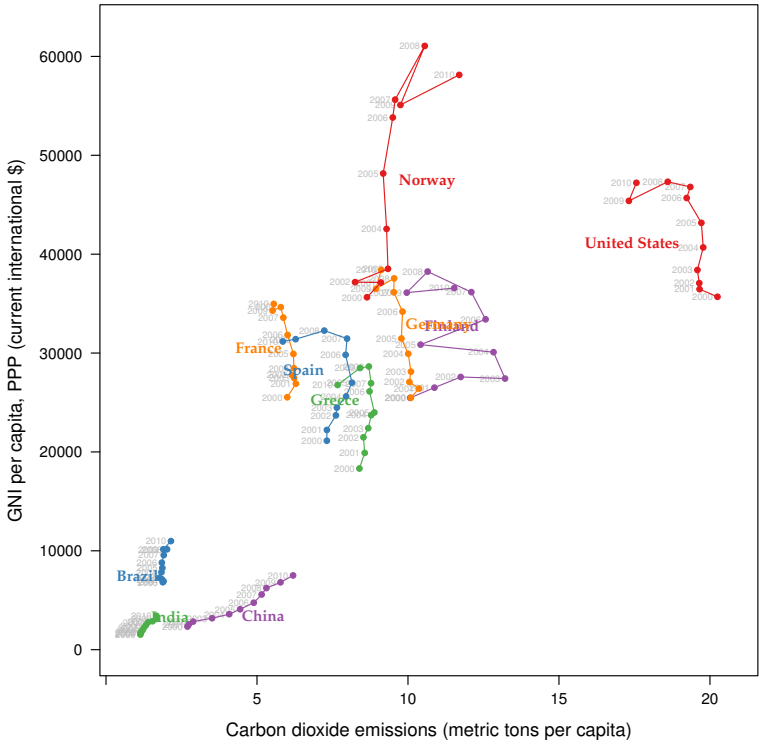

FIGURE 5.3: CO_2 emissions versus GNI per capita. Labels are placed with `panel.pointLabel`.

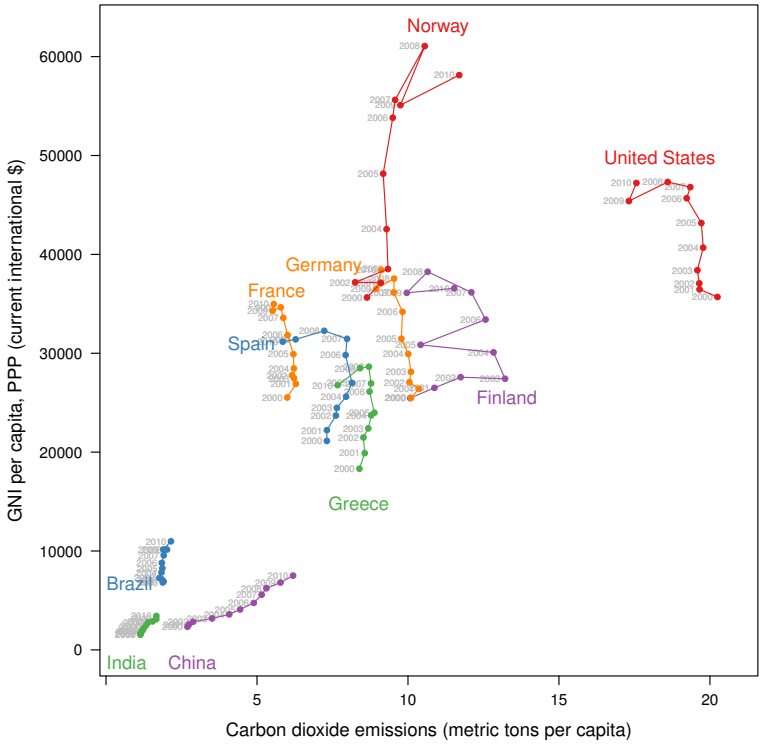

FIGURE 5.4: CO_2 emissions versus GNI per capita. Labels are placed with the `extreme.grid` method of the `directlabels` package.

5.5 A Panel for Each Year

Time can be used as a conditioning variable (as shown in previous sections) to display subsets of the data in different panels. Figure 5.5 is produced with the same code as in Figure 5.1, now including `|factor(Year)` in the lattice version and `facet_wrap(~ Year)` in the `ggplot2` version.

```
xyplot(GNI.capita ~ CO2.capita | factor(Year), data=CO2data,
      xlab="Carbon dioxide emissions (metric tons per capita)",
      ylab="GNI per capita, PPP (current international $)",
      groups=Country.Name, type='b',
      auto.key=list(space='right'))
```

```
ggplot(data=CO2data, aes(x=CO2.capita, y=GNI.capita, colour=Country
    .Name)) +
   facet_wrap(~ Year) + geom_point(pch=19) +
   xlab('CO2 emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

Because the grouping variable, `Country.Name`, has many levels, the legend is not very useful. Once again, point labeling is recommended (Figure 5.6).

```
xyplot(GNI.capita ~ CO2.capita | factor(Year), data=CO2data,
      xlab="Carbon dioxide emissions (metric tons per capita)",
      ylab="GNI per capita, PPP (current international $)",
      groups=Country.Name, type='b',
      par.settings=myTheme) +
   glayer(panel.pointLabel(x, y, labels=group.value,
                      col=palOrdered[group.number], cex=0.7))
```

5.5.1 Using Variable Size to Encode an Additional Variable

Instead of using simple points, we can display circles of different radius to encode a new variable. This new variable is `CO2.PPP`, the ratio of CO_2 emissions to the Gross Domestic Product with purchasing power parity (PPP) estimations.

To use this numeric variable as an additional grouping factor, its range must be divided into different classes. The typical solution is to use `cut` to coerce the numeric variable into a `factor` whose levels correspond to uniform intervals, which could be unrelated to the data distribution. The `classInt` package (R. Bivand 2013) provides several methods to partition data into classes based on natural groups in the data distribution.

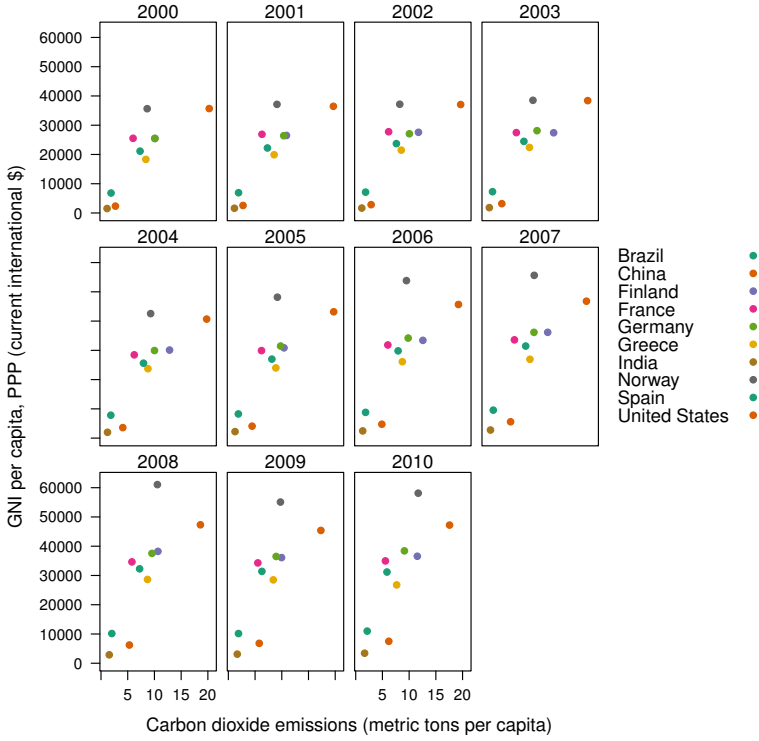

FIGURE 5.5: CO_2 emissions versus GNI per capita with a panel for each year.

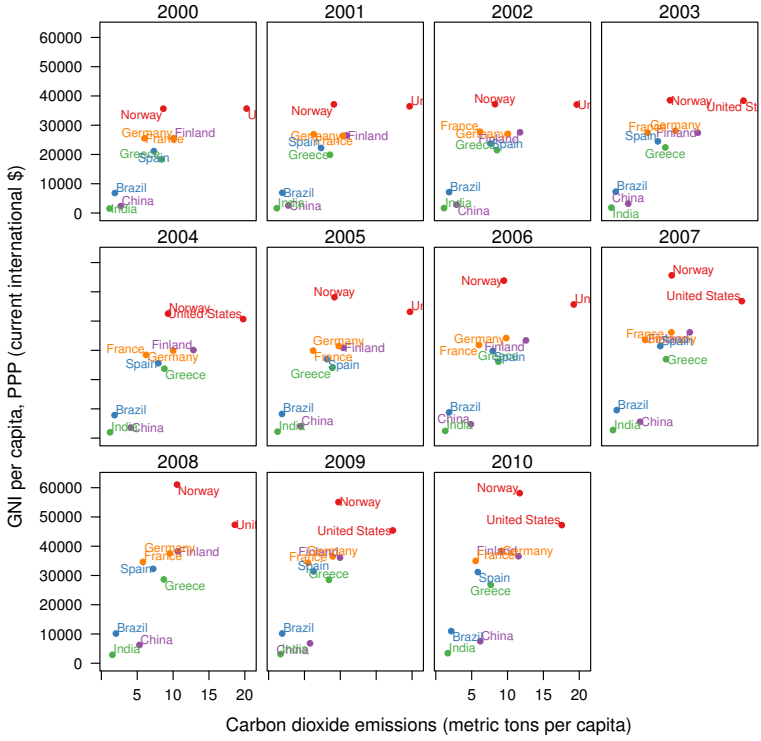

FIGURE 5.6: CO_2 emissions versus GNI per capita with a panel for each year.

```
library(classInt)
z <- CO2data$CO2.PPP
intervals <- classIntervals(z, n=4, style='fisher')
```

Although the functions of this package are mainly intended to create color palettes for maps, the results can also be associated to point sizes. `cex.key` defines the sequence of sizes (to be displayed in the legend) associated with each `CO2.PPP` using the `findCols` function.

```
nInt <- length(intervals$brks) - 1
cex.key <- seq(0.5, 1.8, length=nInt)

idx <- findCols(intervals)
CO2data$cexPoints <- cex.key[idx]
```

The graphic will display information on two variables (`GNI.capita` and `CO2.capita` in the vertical and horizontal axes, respectively) with a conditioning variable (`Year`) and two grouping variables (`Country.Name`, and `CO2.PPP` through `cexPoints`) (Figure 5.7).

```
ggplot(data=CO2data, aes(x=CO2.capita, y=GNI.capita, colour=Country
    .Name)) +
   facet_wrap(~ Year) + geom_point(aes(size=cexPoints), pch=19) +
   xlab('Carbon dioxide emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

The `auto.key` mechanism of the `lattice` version is not able to cope with two grouping variables. Therefore, the legend, whose main componens are the labels (`intervals`) and the point sizes (`cex.key`), should be defined manually (Figure 5.8).

```
op <- options(digits=2)
tab <- print(intervals)
options(op)

key <- list(space='right',
          title=expression(CO[2]/GNI.PPP),
          cex.title=1,
          ## Labels of the key are the intervals strings
          text=list(labels=names(tab), cex=0.85),
          ## Points sizes are defined with cex.key
          points=list(col='black', pch=19,
             cex=cex.key, alpha=0.7))
```

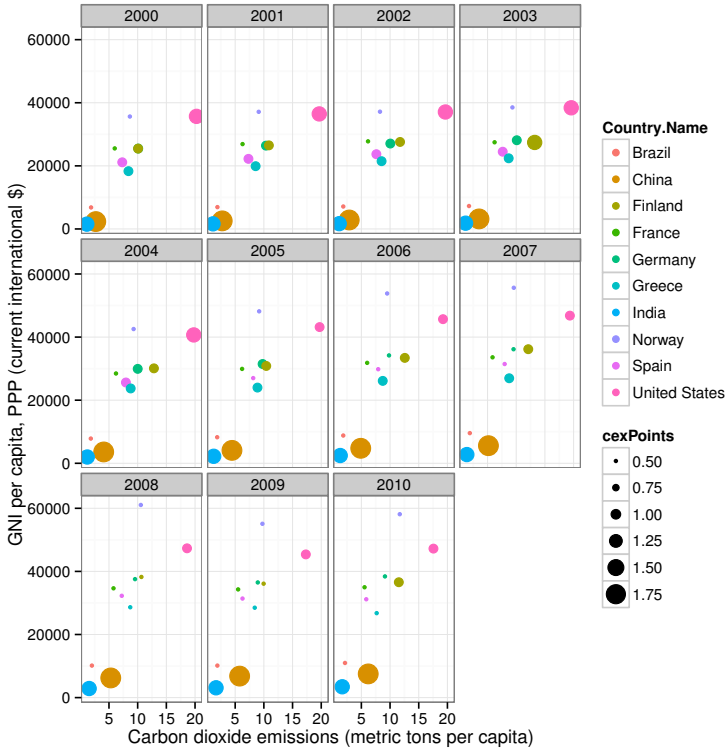

FIGURE 5.7: CO_2 emissions versus GNI per capita for different intervals of the ratio of CO_2 emissions to the GDP PPP estimations.

```
xyplot(GNI.capita ~ CO2.capita|factor(Year), data=CO2data,
     xlab="Carbon dioxide emissions (metric tons per capita)",
     ylab="GNI per capita, PPP (current international $)",
     groups=Country.Name, key=key, alpha=0.7,
     panel = panel.superpose,
     panel.groups = function(x, y,
        subscripts, group.number, group.value, ...){
        panel.xyplot(x, y,
                  col = palOrdered[group.number],
                  cex = CO2data$cexPoints[subscripts])
        panel.pointLabel(x, y, labels=group.value,
                     col=palOrdered[group.number],
                     cex=0.7)
     }
     )
```

5.6 Interactive

5.6.1 googleVis

The first solution is a Motion Chart the googleVis package (Gesmann and Castillo 2011), an interface between R and the Google Visualisation API. With its gvisMotionChart function it is easy to produce a Motion Chart that can be displayed using a browser with Flash enabled (Figure 5.9).

```
library(googleVis)
pgvis <- gvisMotionChart(CO2data, idvar='Country.Name', timevar='
    Year')
```

Although the gvisMotionChart is quite easy to use, the global appearance and behavior are completely determined by Google API[3]. Moreover, you should carefully read their Terms of Use before using it for public distribution.

5.6.2 plotly

```
library(plotly)
```

[3]You should read the Google API Terms of Service before using googleVis: https://developers.google.com/terms/.

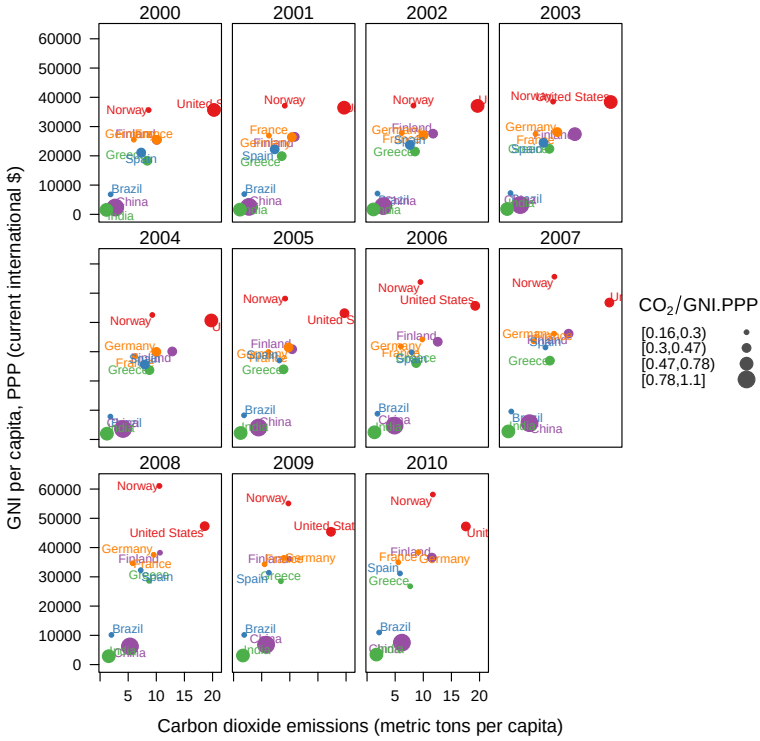

FIGURE 5.8: CO_2 emissions versus GNI per capita for different intervals of the ratio of CO_2 emissions to the GDP PPP estimations.

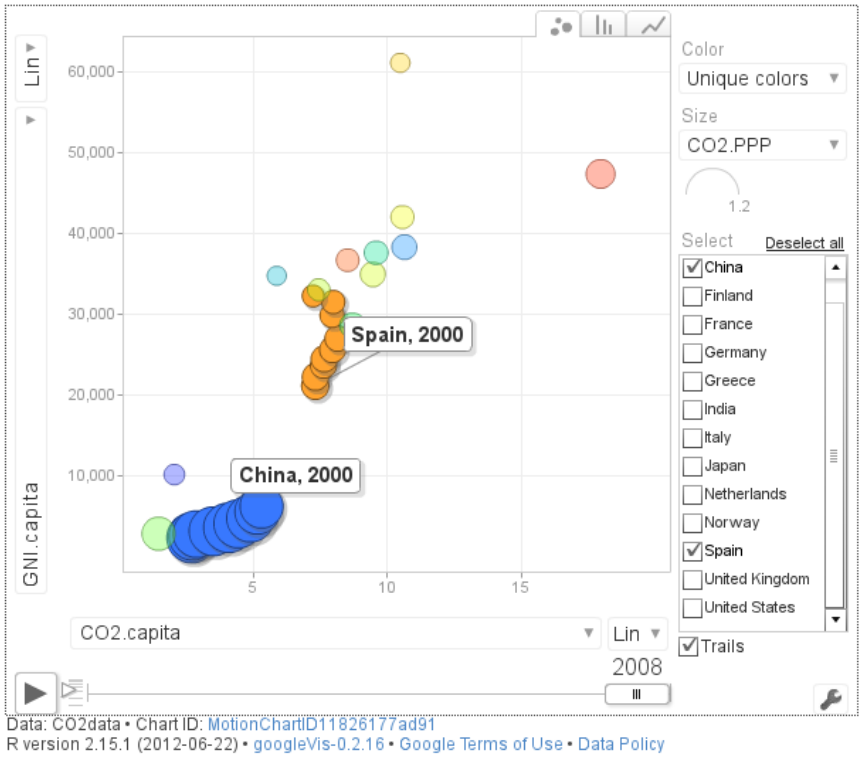

FIGURE 5.9: Snapshot of a Motion Chart produced with googleVis.

```
p <- plot_ly(CO2data,
          x = ~CO2.capita,
          y = ~GNI.capita,
          size = ~CO2.PPP,
          text = ~Country.Name, hoverinfo = "text")

p <- add_markers(p,
              color = ~Country.Name,
              frame = ~Year,
              ids = ~Country.Name,
              showlegend = FALSE)

p <- animation_opts(p,
```

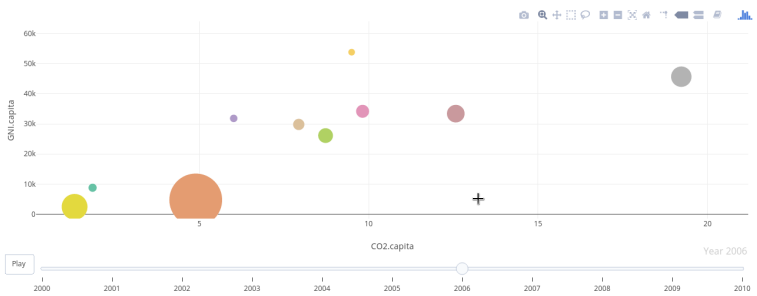

FIGURE 5.10: plotly animation

```
                frame = 1000,
                transition = 800,
                redraw = FALSE) %>%
   animation_slider(
      currentvalue = list(prefix = "Year "))

p
```

5.6.3 gridSVG

The final solution to display this multivariate time series is with animation via the function `grid.animate` of the `gridSVG` package. We will mimic the Trendalyzer/Motion Chart solution, using traveling bubbles of different colors and with radius proportional to `CO2.PPP`.

The first step is to draw the initial state of the bubbles. Their colors are again defined by the `palOrdered` palette, although the `adjustcolor` function is used for a ligther `fill` color. Because there will not be a legend, there is no need to define class intervals, and thus the radius is directly proportional to the value of `CO2data$CO2.PPP`.

```
library(gridSVG)

xyplot(GNI.capita ~ CO2.capita, data=CO2data,
      xlab="Carbon dioxide emissions (metric tons per capita)",
      ylab="GNI per capita, PPP (current international $)",
      subset=Year==2000, groups=Country.Name,
      ## The limits of the graphic are defined
      ## with the entire dataset
      xlim=extendrange(CO2data$CO2.capita),
```

```
      ylim=extendrange(CO2data$GNI.capita),
      panel=function(x, y, ..., subscripts, groups) {
        color <- palOrdered[groups[subscripts]]
        radius <- CO2data$CO2.PPP[subscripts]
        ## Size of labels
        cex <- 1.1*sqrt(radius)
        ## Bubbles
        grid.circle(x, y, default.units="native",
                    r=radius*unit(.25, "inch"),
                    name=trellis.grobname("points", type="panel"),
                    gp=gpar(col=color,
                      ## Fill color ligther than border
                      fill=adjustcolor(color, alpha=.5),
                      lwd=2))
        ## Country labels
        grid.text(label=groups[subscripts],
                  x=unit(x, 'native'),
                  ## Labels above each bubble
                  y=unit(y, 'native') + 1.5 * radius *unit(.25, 'inch'
                      ),
                  name=trellis.grobname('labels', type='panel'),
                  gp=gpar(col=color, cex=cex))
      })
```

From this initial state, `grid.animate` creates a collection of animated graphical objects with the result of `animUnit`. This function produces a set of values that will be interpreted by `grid.animate` as intermediate states of a feature of the graphical object. Thus, the bubbles will travel across the values defined by `x_points` and `y_points`, while their labels will use `x_points` and `x_labels`.

The use of `rep=TRUE` ensures that the animation will be repeated indefinitely.

```
## Duration in seconds of the animation
duration <- 20

nCountries <- nlevels(CO2data$Country.Name)
years <- unique(CO2data$Year)
nYears <- length(years)

## Intermediate positions of the bubbles
x_points <- animUnit(unit(CO2data$CO2.capita, 'native'),
                     id=rep(seq_len(nCountries), each=nYears))
y_points <- animUnit(unit(CO2data$GNI.capita, 'native'),
```

```
                id=rep(seq_len(nCountries), each=nYears))
## Intermediate positions of the labels
y_labels <- animUnit(unit(CO2data$GNI.capita, 'native') +
                1.5 * CO2data$CO2.PPP * unit(.25, 'inch'),
                id=rep(seq_len(nCountries), each=nYears))
## Intermediate sizes of the bubbles
size <- animUnit(CO2data$CO2.PPP * unit(.25, 'inch'),
                id=rep(seq_len(nCountries), each=nYears))

grid.animate(trellis.grobname("points", type="panel", row=1, col=1)
    ,
          duration=duration,
          x=x_points,
          y=y_points,
          r=size,
          rep=TRUE)

grid.animate(trellis.grobname("labels", type="panel", row=1, col=1)
    ,
          duration=duration,
          x=x_points,
          y=y_labels,
          rep=TRUE)
```

A bit of interactivity can be added with the `grid.hyperlink` function. For example, the following code adds the corresponding Wikipedia link to a mouse click on each bubble.

```
countries <- unique(CO2data$Country.Name)
URL <- paste('http://en.wikipedia.org/wiki/', countries, sep='')
grid.hyperlink(trellis.grobname('points', type='panel', row=1, col
    =1),
            URL, group=FALSE)
```

Finally, the time information: The year is printed in the lower right corner, using the `visibility` attribute of an animated `textGrob` object to show and hide the values.

```
visibility <- matrix("hidden", nrow=nYears, ncol=nYears)
diag(visibility) <- "visible"
yearText <- animateGrob(garnishGrob(textGrob(years, .9, .15,
                                    name="year",
                                    gp=gpar(cex=2, col="grey")),
                           visibility="hidden"),
                  duration=20,
```

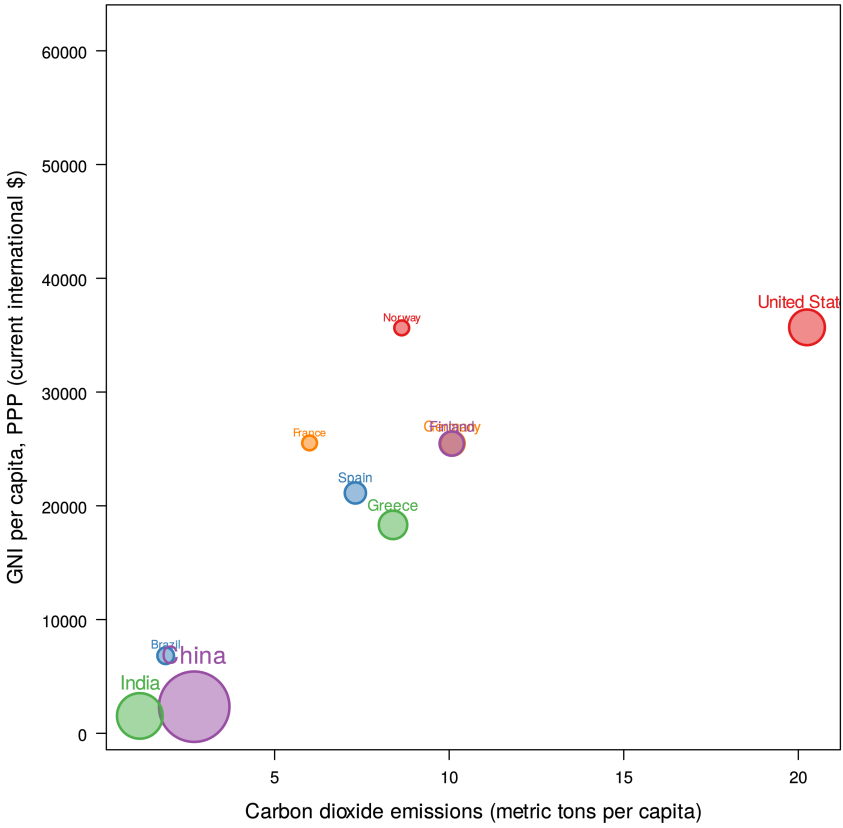

FIGURE 5.11: Animated bubbles produced with `gridSVG`.

```
                    visibility=visibility,
                    rep=TRUE)
grid.draw(yearText)
```

The SVG file produced with `grid.export` is available at the website of the book (Figure 5.11). Because this animation does not trace the paths, Figure 5.4 provides this information as a static complement.

```
grid.export("figs/bubbles.svg")
```

Now, sit down in your favorite easy chair and watch the magistral video "200 Countries, 200 Years, 4 Minutes"[4]. After that, you are ready to open the SVG file of traveling bubbles: It is easier, a short time period with less than twenty countries.

[4] http://www.gapminder.org/videos/200-years-that-changed-the-world-bbc/

Chapter 6

About the Data

6.1 SIAR

The Agroclimatic Information System for Irrigation (SIAR) (MARM 2011) is a free-download database operating since 1999, covering the majority of the irrigated area of Spain. This network belongs to the Ministry of Agriculture, Food and Environment of Spain, as a tool to predict and study meteorological variables for agriculture. SIAR is composed of twelve regional centers and a national center, aiming to centralize and depurate measurements from the stations of the network. Figure 6.1 displays the stations over an altitude map. Some stations from the complete network have been omitted, due to difficulties accessing their coordinates or to incomplete or spurious data series[1].

6.1.1 Daily Data of Different Meteorological Variables

As an example of multiple time series with different scales, we will use 8 years (from January 2004 to December 2011) of daily data corresponding to several meteorological variables measured at the SIAR station located at Aranjuez (Madrid, Spain) available on the SIAR webpage[2]. The `aranjuez.gz` file, available in the `data` folder of the book repository, contains

[1] The name and location data of these stations are available at the GitHub repository of the paper (Antonanzas-Torres, Cañizares, and O. Perpiñán 2013).

[2] `http://eportal.magrama.gob.es/websiar`

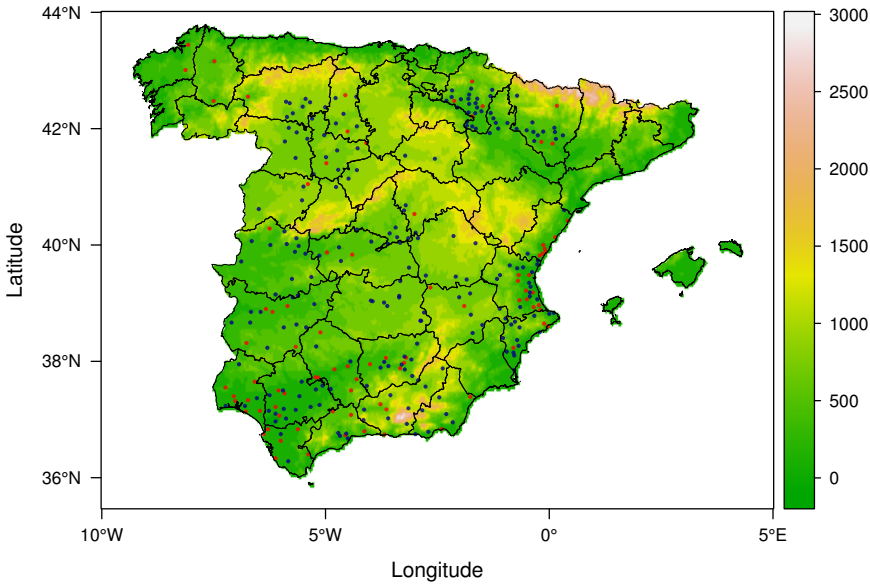

FIGURE 6.1: Meteorological stations of the SIAR network. The color key indicates the altitude (meters).

this information with several meteorological variables: average, maximum, and minimum ambient temperature; average and maximum humidity; average and maximum wind speed; rainfall; solar radiation on the horizontal plane; and evotranspiration.

The `read.zoo` from the `zoo` package accepts this string and downloads the data to construct a `zoo` object. Several arguments are passed directly to `read.table` (`header`, `skip`, etc.) and are detailed conveniently on the help page of this function. The `index.column` is the number of the column with the time index, and `format` defines the date format of this index.

```
library(zoo)

aranjuez <- read.zoo("data/aranjuez.gz",
                 index.column = 3, format = "%d/%m/%Y",
                 fileEncoding = 'UTF-16LE',
                 header = TRUE, fill = TRUE,
                 sep = ';', dec = ",", as.is = TRUE)
aranjuez <- aranjuez[, -c(1:4)]
```

```
names(aranjuez) <- c('TempAvg', 'TempMax', 'TempMin',
                     'HumidAvg', 'HumidMax',
                     'WindAvg', 'WindMax',
                     'Radiation', 'Rain', 'ET')


summary(aranjuez)
```

From the summary it is clear that parts of these time series include erroneous outliers that can be safely removed:

```
aranjuezClean <- within(as.data.frame(aranjuez),{
  TempMin[TempMin>40] <- NA
  HumidMax[HumidMax>100] <- NA
  WindAvg[WindAvg>10] <- NA
  WindMax[WindMax>10] <- NA
})

aranjuez <- zoo(aranjuezClean, index(aranjuez))
```

6.1.2 Solar Radiation Measurements from Different Locations

As an example of multiple time series with the same scale, we will use data of daily solar radiation measurements from different locations.

Daily solar radiation incident on the horizontal plane is registered by meterological stations and estimated from satellite images. This meteorological variable is important for a wide variety of scientific disciplines and engineering applications. Its variations and trends, dependent on the location (mainly latitude, and also longitude and altitude) and on time (day of the year), have been analyzed and modeled in a huge collection of papers and reports. In this section we will focus our attention on the time evolution of the solar radiation. The spatial distribution and the spatio-time behavior will be the subject of later sections.

The stations of the SIAR network include first-class pyranometers according to the World Meteorological Organization (WMO), whose absolute accuracy is within $\pm 5\%$ and is typically lower than $\pm 3\%$. Solar irradiance is recorded every 15 minutes and then collated through a datalogger within the station to generate the daily irradiation, which is later sent to the regional and national centers.

The file `navarra.RData` contains daily solar radiation data of 2011 from the meteorological stations of Navarra, Spain. The names of the dataset are the abbreviations of each station name.

6.2 Unemployment in the United States

As an example of time series that can be displayed both in individual and in aggregate, we will use the unemployment data in the United States. The information on unemployed persons by industry and class of worker is available in Table A-14 published by the Bureau of Labor Statistics[3].

The dataset arranges the information with a row for each category (`Series.ID`) and a column for each monthly value. In addition, there are columns with the annual summaries (`annualCols`). We rearrange this `data.frame`, dropping the `Series.ID` and the annual columns, and transpose the data.

```
unemployUSA <- read.csv('data/unemployUSA.csv')
nms <- unemployUSA$Series.ID
##columns of annual summaries
annualCols <- 14 + 13*(0:12)
## Transpose. Remove annual summaries
unemployUSA <- as.data.frame(t(unemployUSA[,-c(1, annualCols)]))
## First 7 characters can be suppressed
names(unemployUSA) <- substring(nms, 7)
head(unemployUSA)
```

With the transpose, the column names of the original data set are now the row names of the `data.frame`. The `as.yearmon` function of the `zoo` package converts the character vector of names into a `yearmon` vector, a class for representing monthly data. With `Sys.setlocale("LC_TIME", 'C')` we ensure that month abbreviations (`%b`) are correctly interpreted in a non-English locale. This vector is the time index of a new `zoo` object.

```
library(zoo)

Sys.setlocale("LC_TIME", 'C')
idx <- as.yearmon(row.names(unemployUSA), format='%b.%Y')
unemployUSA <- zoo(unemployUSA, idx)
```

Finally, those rows with `NA` values are removed.

```
isNA <- apply(is.na(unemployUSA), 1, any)
unemployUSA <- unemployUSA[!isNA,]
```

[3] http://www.bls.gov/webapps/legacy/cpsatab14.htm

6.3 Gross National Income and CO_2 Emissions

The catalog data of the World Bank Open Data initiative includes the World Development Indicators (WDI)[4]. Among them we will analyze the evolution of the relationship between Gross National Income (GNI) and CO_2 emissions for a set of countries. The package `WDI` is able to search and download these data series.

```
library(WDI)

CO2data <- WDI(indicator=c('EN.ATM.CO2E.PC', 'EN.ATM.CO2E.PP.GD',
           'NY.GNP.MKTP.PP.CD', 'NY.GNP.PCAP.PP.CD'),
        start=2000, end=2011,
        country=c('BR', 'CN', 'DE', 'ES',
           'FI', 'FR', 'GR', 'IN', 'NO', 'US'))

names(CO2data) <- c('iso2c', 'Country.Name', 'Year',
                'CO2.capita', 'CO2.PPP',
                'GNI.PPP', 'GNI.capita')
```

Only two minor modifications are needed: Remove the missing values and convert the `Country.Name` column into a `factor`. This first modification will save problems when displaying the time series, and the `factor` conversion will be useful for grouping.

```
isNA <- apply(is.na(CO2data), 1, any)
CO2data <- CO2data[!isNA, ]

CO2data$Country.Name <- factor(CO2data$Country.Name)
```

[4] http://databank.worldbank.org/data/reports.aspx?source=world-development-indicators