

R practice using data from the ENSEMBLES Project

Joaquín Bedia

November 6, 2012



Contents

1	Introduction	3
1.1	Getting help	3
1.2	Sample Datasets	3
1.3	Before starting	3
2	Accessing netCDF data	4
3	Handling dates	10
4	Spatial overlay and interpolation	14
4.1	Point sampling	15
4.2	Spatial Overlay with vector maps	19
4.3	Nearest neighbour interpolation	21
4.4	Bilinear interpolation	23
5	Re-Projecting model grids	24
5.1	Introduction	24
5.2	Rotated coordinates	26
5.3	Re-projecting data	29
6	VALUE-THREDDS data access	37
6.1	Introduction to THREDDS	37
6.2	THREDDS data download	38
6.2.1	netCDF subsetting	39
7	A practical example	41
7.1	Analyzing model results	45
7.1.1	Taylor diagram	45
7.1.2	Correlation	47
8	Summary of R packages used	49

1 Introduction

During this practice we will undertake some operations related with the handling, analysis and visualization of climate data in the R environment. To this aim, we will use several packages that can be useful to this aim (see Section 8 for a quick overview), although the methods and steps followed in this practice are just examples that can in many cases be accomplished following other alternative approaches.

1.1 Getting help

Once R is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

```
help.start() general help
```

```
help(foo) help about function foo
```

```
?foo same thing
```

```
apropos("foo") list all functions containing string foo
```

```
example(foo) show an example of function foo
```

```
RSiteSearch("foo") searches for help manuals and archived mailing lists
```

```
vignette() show available vignettes. Vignettes are optional supplementary  
documentation available in some packages
```

```
vignette("foo") show specific vignette
```

1.2 Sample Datasets

R comes with a number of sample datasets that you can experiment with. Type `data()` to see the available datasets. The results will depend on which packages are loaded. Type `help(datasetname)` for details on a sample dataset. For instance, the world map that we will use during this practice is a built-in dataset of package `maptools`. We load it first of all:

```
> library(maptools)
> data(wrld_simpl) # loads the world map dataset
> as(wrld_simpl,"SpatialLines") -> wrl
```

1.3 Before starting

In addition, we will use some example files of the ENSEMBLES database. For convenience, these files will be provided and stored locally into your local machine before starting this practical session. Ideally, these files could be directly downloaded from the ENSEMBLES site using the appropriate R tools. However, because of the size of the files and to avoid network problems, it will be faster to have this files locally stored. We will later illustrate remote data download in Section 6 using the VALUE-THREDDS data server capabilities.

During the next sections, the example data will be stored at:

```
"C:/VALUE/files/"
```

First of all, we will define our working directory using the `setwd` command. If you want to specify another different working directory, now it is the moment to do it. However, please conserve a sub-directory into the working directory named "files" with the data provided inside, for full reproducibility of the example scripts presented.

```
> setwd("C:/VALUE/")
> getwd()

[1] "C:/VALUE"
```

2 Accessing netCDF data

During this practice we will use the utilities of the `ncdf` package for netCDF data access. However, it is important to note that the `ncdf` package is designed to work with the netCDF library version 3¹. Newer package `ncdf4` is designed to work with the netCDF library version 4, and supports features such as compression and chunking. Unfortunately, for various reasons the `ncdf4` package must have a different API than the `ncdf` package.

In this practice, we will use the `ncdf` interface, partly because data stored in the ENSEMBLES database correspond to the netCDF version 3, and also because the installation of `ncdf4` is not as straightforward as `ncdf`, because it is not available at the R-CRAN repository at the moment of writing this tutorial (you can find further details on `ncdf4` installation under different platforms at this link: <http://cirrus.ucsd.edu/~pierce/ncdf/>).

In order to access data stored in a netCDF, first of all we need to open a connection with the file. This is done with the `open.ncdf` command:

```
> library(ncdf)
> open.ncdf("./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc") -> nc
```

Once the connection is open, we can obtain some preliminary information about a netCDF file, including the variables and dimensions it contains:

```
> print.ncdf(nc)

[1] "file ./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc has 3 dimensions:"
[1] "x   Size: 93"
[1] "y   Size: 101"
[1] "time Size: 3653"
[1] "-----"
[1] "file ./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc has 4 variables:"
[1] "float lon[x,y] Longname:longitude Missval:1e+30"
[1] "float lat[x,y] Longname:latitude Missval:1e+30"
[1] "char Lambert_conformal[] Longname:Lambert_conformal Missval:NA"
[1] "float pr[x,y,time] Longname:Precipitation Missval:1e+30"
```

¹see here for more information on the R interfaces to netCDF: <http://cirrus.ucsd.edu/~pierce/ncdf/>

In order to read the data from the netCDF, we will use the `get.var.ncdf` function. You can type `?get.var.ncdf` to see all details about the function. In this example we will illustrate a simple example using an ENSEMBLES file of precipitation from the CNRM-RM4.5 RCM coupled to ERA-40. The three most important arguments of the `get.var.ncdf` function, apart from specifying the file to read from, are the `varid` argument, which specifies the variable to be read, and the `start` and `count` arguments, which are vectors of indices whose length is equal to the number of dimensions the variable has, indicating where start to reading and the count of values to read along each dimension respectively.

In order to know the `varid` value (which can be either a number or a string), we can for instance look into the `nc` object:

```
> names(nc$var)

[1] "lon"          "lat"
[3] "Lambert_conformal" "pr"

> str(nc$var$pr)

List of 16
 $ id      : int 7
 $ name    : chr "pr"
 $ ndims   : int 3
 $ natts   : int 6
 $ size    : int [1:3] 93 101 3653
 $ prec    : chr "float"
 $ dimids  : num [1:3] 1 2 3
 $ units   : chr "kg m-2 s-1"
 $ longname : chr "Precipitation"
 $ dims    : list()
 $ dim     :List of 3
 ..$ :List of 8
 .. ..$ name      : chr "x"
 .. ..$ len       : int 93
 .. ..$ unlim     : logi FALSE
 .. ..$ id        : int 1
 .. ..$ dimvarid  : num 1
 .. ..$ units     : chr "km"
 .. ..$ vals      : num [1:93(1d)] 1 2 3 4 5 6 7 8 9 10 ...
 .. ..$ create_dimvar: logi TRUE
 .. ..- attr(*, "class")= chr "dim.ncdf"
 ..$ :List of 8
 .. ..$ name      : chr "y"
 .. ..$ len       : int 101
 .. ..$ unlim     : logi FALSE
 .. ..$ id        : int 2
 .. ..$ dimvarid  : num 2
 .. ..$ units     : chr "km"
 .. ..$ vals      : num [1:101(1d)] 1 2 3 4 5 6 7 8 9 10 ...
 .. ..$ create_dimvar: logi TRUE
 .. ..- attr(*, "class")= chr "dim.ncdf"
```

```

..$ :List of 8
.. ..$ name      : chr "time"
.. ..$ len       : int 3653
.. ..$ unlim     : logi TRUE
.. ..$ id        : int 3
.. ..$ dimvarid  : num 5
.. ..$ units     : chr "days since 1950-01-01"
.. ..$ vals      : num [1:3653(1d)] 14975 14976 14977 14978 14979 ...
.. ..$ create_dimvar: logi TRUE
.. ..- attr(*, "class")= chr "dim.ncdf"
$ varsize       : int [1:3] 93 101 3653
$ unlim         : logi TRUE
$ missval       : num 1e+30
$ hasAddOffset : logi FALSE
$ hasScaleFact : logi FALSE
- attr(*, "class")= chr "var.ncdf"

> nc$var$pr$id

[1] 7

```

From the above lines we know that the variable name is “pr” and that it is identified with number 7. The same can be done in order to retrieve information about the dimensions:

```

> str(nc$dim)

List of 3
 $ x   :List of 8
  ..$ name      : chr "x"
  ..$ len       : int 93
  ..$ unlim     : logi FALSE
  ..$ id        : int 1
  ..$ dimvarid  : num 1
  ..$ units     : chr "km"
  ..$ vals      : num [1:93(1d)] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ create_dimvar: logi TRUE
  ..- attr(*, "class")= chr "dim.ncdf"
 $ y   :List of 8
  ..$ name      : chr "y"
  ..$ len       : int 101
  ..$ unlim     : logi FALSE
  ..$ id        : int 2
  ..$ dimvarid  : num 2
  ..$ units     : chr "km"
  ..$ vals      : num [1:101(1d)] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ create_dimvar: logi TRUE
  ..- attr(*, "class")= chr "dim.ncdf"
 $ time:List of 8
  ..$ name      : chr "time"
  ..$ len       : int 3653

```

```

..$ unlim      : logi TRUE
..$ id         : int 3
..$ dimvarid   : num 5
..$ units      : chr "days since 1950-01-01"
..$ vals       : num [1:3653(1d)] 14975 14976 14977 14978 14979 ...
..$ create_dimvar: logi TRUE
..- attr(*, "class")= chr "dim.ncdf"

> names(nc$dim)

[1] "x"    "y"    "time"

```

So precipitation is coded as “pr” in the file (this nomenclature is consistent among all the ENSEMBLES datasets), as is coded as variable 7. In order to access all precipitation data stored, we could just type:

```
> get.var.ncdf(nc=nc, varid="pr") -> pr.cnrnm.1
```

Alternatively, the index of the variable can be indicated:

```
> get.var.ncdf(nc=nc, varid=7) -> pr.cnrnm.2
> identical(pr.cnrnm.1, pr.cnrnm.2)

[1] TRUE

```

If we look at the structure of the data, we will find that data are arranged in a 3-dimensional array:

```
> str(pr.cnrnm.1)

num [1:93, 1:101, 1:3653] 7.15e-06 1.10e-06 4.65e-07 4.26e-07 2.10e-07 ...

> class(pr.cnrnm.1)

[1] "array"

> dim(pr.cnrnm.1)

[1] 93 101 3653

```

Basically, the information is arranged in a “cube”, in which the first two dimensions indicate the position of the observations in the space (x and y coordinates), and the third dimension corresponds to time, which in this particular case it has a length of 3653 days (10 years, 1991-2000). Remember that you can retrieve all the information about this variable by typing:

```
> str(nc$var$pr)

List of 16
 $ id      : int 7
 $ name    : chr "pr"
 $ ndims   : int 3
 $ natts   : int 6
 $ size    : int [1:3] 93 101 3653

```

```

$ prec      : chr "float"
$ dimids    : num [1:3] 1 2 3
$ units     : chr "kg m-2 s-1"
$ longname  : chr "Precipitation"
$ dims      : list()
$ dim       :List of 3
..$ :List of 8
.. ..$ name      : chr "x"
.. ..$ len       : int 93
.. ..$ unlim     : logi FALSE
.. ..$ id        : int 1
.. ..$ dimvarid  : num 1
.. ..$ units     : chr "km"
.. ..$ vals      : num [1:93(1d)] 1 2 3 4 5 6 7 8 9 10 ...
.. ..$ create_dimvar: logi TRUE
.. ..- attr(*, "class")= chr "dim.ncdf"
..$ :List of 8
.. ..$ name      : chr "y"
.. ..$ len       : int 101
.. ..$ unlim     : logi FALSE
.. ..$ id        : int 2
.. ..$ dimvarid  : num 2
.. ..$ units     : chr "km"
.. ..$ vals      : num [1:101(1d)] 1 2 3 4 5 6 7 8 9 10 ...
.. ..$ create_dimvar: logi TRUE
.. ..- attr(*, "class")= chr "dim.ncdf"
..$ :List of 8
.. ..$ name      : chr "time"
.. ..$ len       : int 3653
.. ..$ unlim     : logi TRUE
.. ..$ id        : int 3
.. ..$ dimvarid  : num 5
.. ..$ units     : chr "days since 1950-01-01"
.. ..$ vals      : num [1:3653(1d)] 14975 14976 14977 14978 14979 ...
.. ..$ create_dimvar: logi TRUE
.. ..- attr(*, "class")= chr "dim.ncdf"
$ varsize   : int [1:3] 93 101 3653
$ unlim     : logi TRUE
$ missval   : num 1e+30
$ hasAddOffset: logi FALSE
$ hasScaleFact: logi FALSE
- attr(*, "class")= chr "var.ncdf"

```

In order to use the `start` and `count` arguments, first of all we need to know the length of the vector defining the dimensions of the variable, that in this case is equal to 3, arranged in the form `[x,y,time]` (note that the time dimension is always the last):

```

> length(nc$var$pr$dim)
[1] 3

```



```

> str(nc$var$pr$dim)

List of 3
 $ :List of 8
  ..$ name      : chr "x"
  ..$ len       : int 93
  ..$ unlim     : logi FALSE
  ..$ id        : int 1
  ..$ dimvarid  : num 1
  ..$ units     : chr "km"
  ..$ vals      : num [1:93(1d)] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ create_dimvar: logi TRUE
  ..- attr(*, "class")= chr "dim.ncdf"
 $ :List of 8
  ..$ name      : chr "y"
  ..$ len       : int 101
  ..$ unlim     : logi FALSE
  ..$ id        : int 2
  ..$ dimvarid  : num 2
  ..$ units     : chr "km"
  ..$ vals      : num [1:101(1d)] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ create_dimvar: logi TRUE
  ..- attr(*, "class")= chr "dim.ncdf"
 $ :List of 8
  ..$ name      : chr "time"
  ..$ len       : int 3653
  ..$ unlim     : logi TRUE
  ..$ id        : int 3
  ..$ dimvarid  : num 5
  ..$ units     : chr "days since 1950-01-01"
  ..$ vals      : num [1:3653(1d)] 14975 14976 14977 14978 14979 ...
  ..$ create_dimvar: logi TRUE
  ..- attr(*, "class")= chr "dim.ncdf"

```

For instance, suppose we are interested only in the first year of data (i.e. the first 365 days). This would be indicated as follows:

```

> get.var.ncdf(nc, varid="pr", start=c(1,1,1),
+             count=c(-1,-1,365)) -> pr.slice1
> str(pr.slice1)

num [1:93, 1:101, 1:365] 7.15e-06 1.10e-06 4.65e-07 4.26e-07 2.10e-07 ...

```

Note that we have the “-1” value the count means “all entries along this dimension”. Similarly, we can access the first year of one single point:

```

> get.var.ncdf(nc, varid="pr", start=c(15,51,1),
+             count=c(1,1,365)) -> pr.slice2
> str(pr.slice2)

num [1:365(1d)] 1.55e-04 2.43e-05 1.21e-04 1.09e-04 7.99e-05 ...

```

Or a particular slice or region of interest (e.g. a 10×10 grid cells region):

```
> get.var.ncdf(nc, varid="pr", start=c(39,35,1),
+             count=c(10,10,-1)) -> pr.slice3
> str(pr.slice3)

num [1:10, 1:10, 1:3653] 0.00 3.09e-08 1.92e-07 1.42e-06 1.11e-06 ...
```

These are the very basics of netCDF reading. During this practice we will use these commands frequently, although for the sake of simplicity we will not focus of slicing. By default, the `get.var.ncdf(nc, varid="foo")`, will retrieve all data, without slicing.

Next, some plotting examples are presented, that will be explained in more depth during the next steps of the practical session. The spatial slices taken in the previous lines are displayed. Note that we convert the units from the original ($\text{kg} \times \text{m}^{-2} \times \text{s}^{-1}$) to mm.

```
> get.var.ncdf(nc, varid="x") -> x
> get.var.ncdf(nc, varid="y") -> y
> library(RColorBrewer)
> brewer.pal(9, "Blues") -> colors
> apply(pr.slice1, MARGIN=c(1,2),
+       FUN=sum) -> pr.accum
> pr.accum*3600*24 -> pr.accum.mm

> image(x,y,pr.accum.mm,col=colors)
> contour(x,y,pr.accum.mm, nlevels=12,
+         add=TRUE)
> title(main="Total precipitation 1991 (mm)")
> rect(xleft=x[39], xright=x[59], ybottom=y[35],
+      ytop=y[55], border="red", lwd=2)
> text(49,62,"Slice 3",col="red", cex=1.5)
> points(15,51, pch=10, col="red", cex=1.5)
> text(15,60,"Point Selection",col="red",cex=.9)
```

One final important remark: always a netCDF file is opened via the `open.ncdf` command, the connection remains open until it is explicitly closed. Forgetting this may give raise to problems because there is a maximum number of connections that R can maintain open. So after extracting all the necessary data, remember to close the file by typing:

```
> close.ncdf(nc)

[[1]]
[1] 6
```

3 Handling dates

There are many tutorials and information regarding the use of chronological objects in R, to which the interested practitioners are referred². We next show

²see e.g. <http://statistics.berkeley.edu/classes/s133/dates.html> for a quick overview

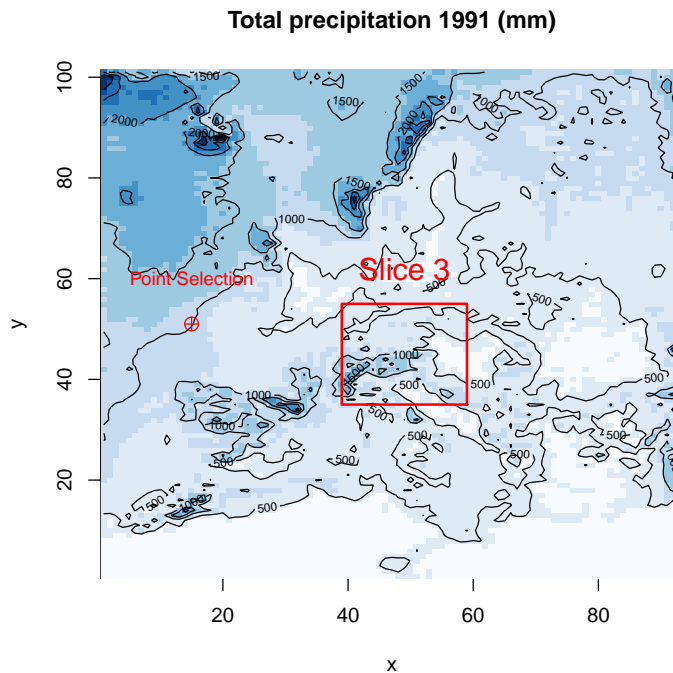


Figure 1: Example of netCDF reading and slice selection.

just a few examples that may result useful for handling time series data from RCM simulations.

Time is a dimension that can be easily extracted from a netCDF file, using the `get.var.ncdf` command previously shown. In this example, we will retrieve the time dimension of the KNMNI-RACMO2.

```
> list.files("./files", pattern="KNMI", full=TRUE) -> target.file
> library(ncdf)
> open.ncdf(target.file) -> nc.knmi
```

When we extract the time dimension, we get a numeric vector that does not contain any chronological information. However, we might be interested in analysing our data using some time aggregation, for instance monthly or seasonal analyses. There are different approaches to convert this numeric vector to a chronological vector of dates. One possibility is to convert the numeric vector to a "Date" class object representing calendar dates using `as.Date`. First of all we need to know the units:

```
> nc.knmi$dim$time$units
[1] "days since 1950-01-01 00:00:00"

> get.var.ncdf(nc.knmi, varid="time") -> ti
> str(ti)
```

```

num [1:3653(1d)] 14976 14976 14978 14978 14980 ...

> as.Date(ti, origin=c("1950-01-01")) -> dates
> class(dates)

[1] "Date"

> str(dates)

Date[1:3653], format: "1991-01-01" "1991-01-02" "1991-01-03" ...

> range(dates)

[1] "1991-01-01" "2000-12-31"

> # Lon/Lat data
> get.var.ncdf(nc.knmi, varid="lon") -> lon
> get.var.ncdf(nc.knmi, varid="lat") -> lat

```

Now the maximum temperature data can be represented as a time series:

```

> get.var.ncdf(nc.knmi, varid="tasmax") -> tx

> # Generates Fig.2
> plot(dates, tx[54,21,], ty='l', col=4, ylab="degree K")
> title(main=paste("Relative max temp from",
+   dates[1],"to",dates[length(dates)]))
> mtext(paste("Point",round(lon[54,21],2),
+   "deg E - ",round(lat[54,21],2),"deg N"),line=.3)

```

We can directly compute months, weeks, years, quarters, julian days and many other chronological objects. In addition, package `chron` provides further utilities. Note that `months`, `quarters` and `weekdays` are generics defined in package `base` which also provides methods for objects of class `"Date"`. These methods return character rather than factor variables as the default methods in `chron` do. To take advantage of the latter, `Date` objects can be converted to `dates` objects using `as.chron`.

```

> library(chron)
> months(dates) -> month.default
> class(month.default)

[1] "character"

> months(as.chron(dates)) -> month.chron
> class(month.chron)

[1] "ordered" "factor"

> levels(month.chron)

[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[10] "Oct" "Nov" "Dec"

```

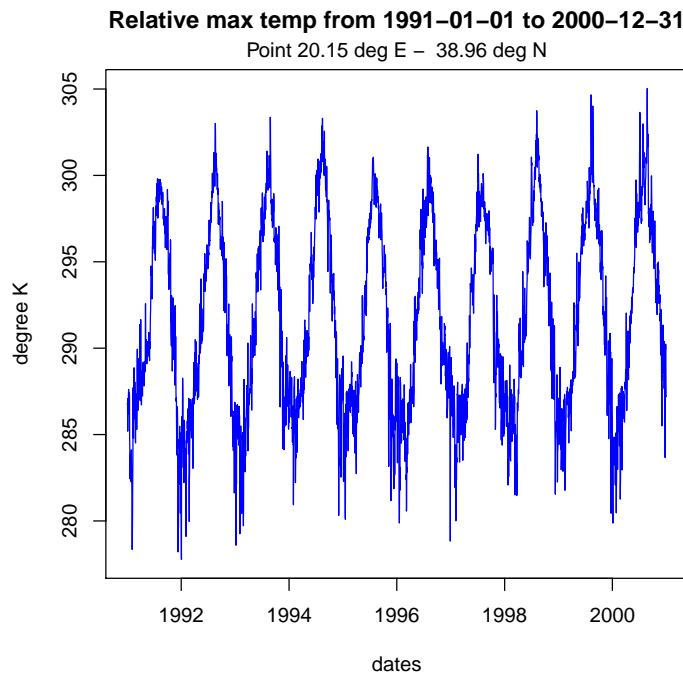


Figure 2: Daily time series of maximum surface air temperature taken at a random point of the RCM grid.

Seasonal representation can be also easily achieved. We create a vector of seasons and then convert it to an ordered factor, so the levels are always represented in chronological order beginning in winter (DJF), rather than alphabetical order.

```
> rep(NA, length(month.chron)) -> seasons
> seasons[grep("Dec|Jan|Feb", month.chron)] <- "DJF"
> seasons[grep("Mar|Apr|May", month.chron)] <- "MAM"
> seasons[grep("Jun|Jul|Aug", month.chron)] <- "JJA"
> seasons[grep("Sep|Oct|Nov", month.chron)] <- "SON"
> factor(seasons, levels=c("DJF","MAM","JJA","SON")
+       , ordered=TRUE) -> seasons
```

In the following example, mean maximum temperature is mapped:

```
> # Generates Fig.3
> library(RColorBrewer)
> get.var.ncdf(nc.knmi, varid="rlon") -> x
> get.var.ncdf(nc.knmi, varid="rlat") -> y
> colorRampPalette(rev(brewer.pal(9,"Spectral")))-> color.palette
> par(mfrow=c(2,2))
> for (i in 1:length(levels(seasons))) {
```

```

+       which(seasons==levels(seasons)[i]) -> season.index
+       apply(tx[ , ,season.index], FUN=mean, MAR=c(1,2)) -> z
+       image(x,y,z, col=color.palette(21), asp=1,
+           main=paste(levels(seasons)[i]))
+       contour(x,y,z,nlevels=15,add=TRUE)
+   }

```

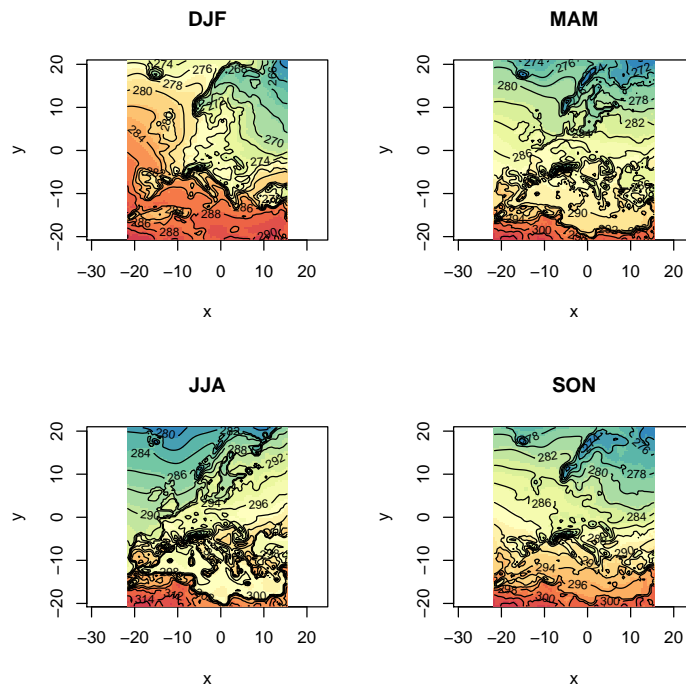


Figure 3: Mean maximum temperature of the period 1991-2000, aggregated by seasons

```

> close.ncdf(nc.knmi)

[[1]]
[1] 6

```

4 Spatial overlay and interpolation

In the following examples we will use the SMHI-RCA RCM grid, a set of data points corresponding to weather stations in Spain and the regular grid of EOBS of 0.5 degree resolution, in order to perform a number of frequent spatial operations like finding the closest grid point to a given location and interpolate the irregular lon-lat RCM grid onto the EOBS regular grid using either nearest neighbours or bilinear interpolation. First of all the required datasets are loaded:

```

> # These are the files in our working directory
> list.files("./files", full.names=TRUE)

[1] "./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc"
[2] "./files/elev_0.50deg_reg_v4.0.nc"
[3] "./files/KNMI-RACMO2_CTL_ERA40_DM_50km_1991-2000_tasmax.nc"
[4] "./files/SMHIRCA_CTR_ERA40_DM_50km_1991-2000_tasmax.nc"
[5] "./files/Stations_Spain.txt"
[6] "./files/TX_EOBS_0.50deg_IP.nc"

> # Read coordinates of weather stations
> read.table("./files/Stations_Spain.txt",
+           col.names=c("lon_st","lat_st")) -> st.coords
> # Open SMHI data
> (list.files("./files", full.names=TRUE,
+           pattern="SMHI") -> target.file)

[1] "./files/SMHIRCA_CTR_ERA40_DM_50km_1991-2000_tasmax.nc"

> open.ncdf(target.file) -> nc.smhi
> # Open EOBS grid
> (list.files("./files", full.names=TRUE,
+           pattern="0.50deg") -> target.file)

[1] "./files/elev_0.50deg_reg_v4.0.nc"
[2] "./files/TX_EOBS_0.50deg_IP.nc"

> open.ncdf(target.file) -> nc.eobs
> get.var.ncdf(nc.eobs, "longitude") -> lon.eobs
> get.var.ncdf(nc.eobs, "latitude") -> lat.eobs
> get.var.ncdf(nc.smhi, "lon") -> lon.rcm
> get.var.ncdf(nc.smhi, "lat") -> lat.rcm
> get.var.ncdf(nc.smhi, "tasmax") -> tx

```

Next, the mean maximum temperature is calculated and the arrays of geographical coordinates are vectorized:

```

> apply(tx, MAR=c(1,2), FUN=mean) -> tx.mean
> as.vector(lon.rcm) -> x
> as.vector(lat.rcm) -> y
> as.vector(tx.mean) -> z

```

4.1 Point sampling

Usually, we need to find where on the RCM grid are certain points or specific locations, such as meteorological weather stations. To this aim, knowing the coordinates of the reference location the aim is finding the closest RCM grid point to that location. This can be achieved in many different ways in R, for instance by the calculation of distance matrices from which the smallest value represents the nearest neighbour. In this example we will use the `spDistsN1` function in package `sp`, for which we can specify both euclidean and great circle distances. The latter option is more appropriate in this case, given that we

are measuring distances along the surface of the Earth's geoid (the WGS84 ellipsoid, in this case), although in practice the results are similar because the differences between the input and the output grid are usually not that large to have any measurable effect. Note that the function `spDistsN1` works faster when euclidean distances are computed. To use the function, we need the coordinates as a matrix of 2D points.

```
> cbind(x, y) -> grid2D.rcm
> as.matrix(st.coords) -> st.coords

> # Generates Fig4
> plot(grid2D.rcm, cex=.4, pch=3,col="grey",
+       asp=1,xlim=c(-25,20), ylim=c(30,55))
> lines(wr1)
> points(st.coords, pch=22, col="red", cex=.6)
> title(main="Weather Stations in Spain")
> legend("bottomleft", c("RCM grid","Stations"),
+       pch=c(3,22), col=c("grey","red"))
```

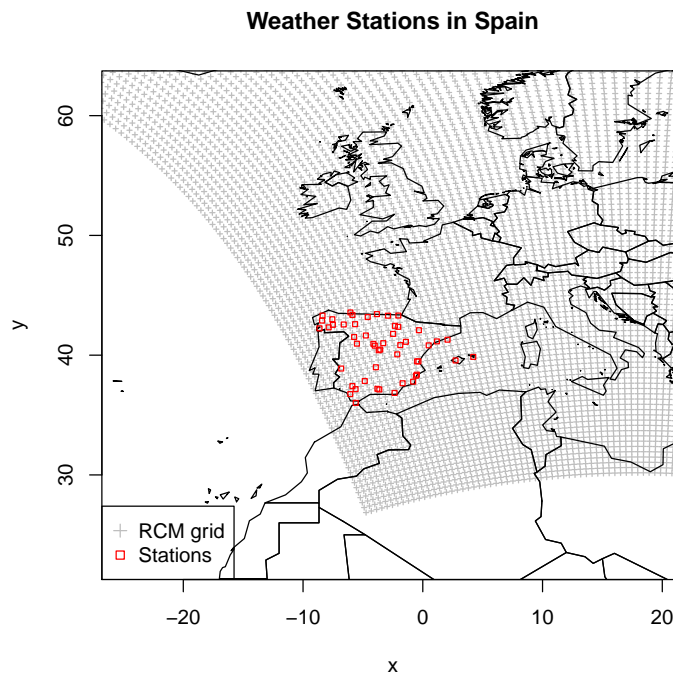


Figure 4: RCM grid and weather station points

For instance, in order to find the closest RCM grid point to a randomly chosen station (for instance station 17 of the 2D matrix of coordinates), we can proceed as follows:

```
> st.coords[17, ] # Coordinates of the reference point
```



```

lon_st lat_st
-4.7544 41.6408

> spDistsN1(pts=grid2D.rcm, pt=st.coords[17, ],
+          longlat=TRUE) -> f
> str(f)

num [1:8075] 1638 1624 1612 1600 1590 ...

```

Object `f` is a long vector, with a length equal to the number of points in the RCM grid. It contains the distance of the reference location to each RCM grid point (in degrees or km depending on the value of the argument `longlat`). The position in which the minimum distance is found corresponds to the index of the RCM coordinate:

```

> which.min(f) -> index
> grid2D.rcm[index, ] # This is the nearest neighbour

      x      y
-4.577028 41.697369

```

`find.nn` is an example wrapper for the `spDistsN1` to perform this task, requiring as argument the 2D matrices of the input and output coordinates. The argument `verbose` is a logical flag indicating whether the function should print on screen the progress of the computation or not. The function returns a list with the grid coordinates corresponding to the nearest location and an index of positions in order to retrieve data from the RCM.

```

> find.nn <- function(points, grid, verbose = FALSE) {
+   pt <- points
+   gr <- grid
+   rep(NA, nrow(pt)) -> index
+   for (i in 1:nrow(pt)) {
+     if(isTRUE(verbose)) {
+       print(paste("Processing point", i,
+                   "out of", nrow(pt)))
+     }
+     which.min(spDistsN1(pts=gr,
+                         pt=pt[i, ])) -> index[i]
+   }
+   return(list("coords"=gr[index, ], "index"=index))
+ }

```

We apply this function to the set of coordinates of stations:

```

> find.nn(st.coords, grid2D.rcm) -> st.grid.coords
> str(st.grid.coords)

```

```

List of 2
 $ coords: num [1:50, 1:2] 1.09 2.22 -2.07 -2.82 -3.97 ...
 ..- attr(*, "dimnames")=List of 2
 .. ..$ : NULL
 .. ..$ : chr [1:2] "x" "y"
 $ index : int [1:50] 2401 2403 2907 2991 2989 2988 3156 3071 3152 3152 ...

```

For instance, the coordinates of the corresponding RCM points are retrieved as follows:

```
> st.grid.coords$coords
      x      y
[1,]  1.0898490 41.03489
[2,]  2.2177310 41.22669
[3,] -2.0718780 43.16106
[4,] -2.8155770 43.46550
[5,] -3.9711630 43.21692
[6,] -4.5451889 43.08785
[7,] -6.0654202 43.65566
[8,] -5.8741679 43.23814
[9,] -8.3438358 43.07521
[10,] -8.3438358 43.07521
[11,] -8.4894915 42.09788
[12,] -7.5759468 42.81058
[13,] -6.8152518 42.54042
[14,] -7.3760829 42.39590
[15,] -7.9341698 42.24837
[16,] -2.7265680 41.65311
[17,] -4.5770278 41.69737
[18,] -4.2289052 40.85715
[19,] -4.2289052 40.85715
[20,] -5.6894889 41.42963
[21,] -5.6855011 42.82031
[22,] -5.5068932 41.01162
[23,] -1.8504280 40.92412
[24,] -3.1219220 41.10886
[25,] -3.5094891 40.56312
[26,] -3.5094891 40.56312
[27,] -4.0938892 39.04705
[28,] -6.9241538 38.78541
[29,] -4.8143101 37.94728
[30,] -3.4407020 37.36354
[31,] -3.9602890 37.23755
[32,] -5.6808510 37.26005
[33,] -5.6808510 37.26005
[34,] -6.0184102 36.70522
[35,] -5.6698651 35.87127
[36,] -2.0938799 36.76035
[37,] -0.8141651 37.94984
[38,] -1.8703920 37.72413
[39,] -0.4224014 38.48448
[40,] -0.4224014 38.48448
[41,] -2.0927370 39.95975
[42,] -0.7055645 39.33645
[43,] -0.1617279 39.44419
[44,] -2.4851489 42.61928
```

```
[45,] -1.7541070 42.31257
[46,] -1.2931440 41.03826
[47,] -0.4615669 42.11021
[48,]  0.5286503 40.93419
[49,]  2.7031560 39.50598
[50,]  4.3581529 39.76140
```

And these are the corresponding mean temperature RCM values previously calculated:

```
> z[st.grid.coords$index]

 [1] 291.7770 290.7250 289.3575 288.8829 288.2719 286.5146
 [7] 288.0437 286.3255 289.0548 289.0548 289.4470 287.7225
[13] 285.9164 286.9739 288.2026 286.1185 290.0869 288.5888
[19] 288.5888 289.6334 284.5639 288.5835 286.6655 287.4188
[25] 290.2802 290.2802 291.7511 293.9595 294.9286 290.5398
[31] 292.3596 296.2580 296.2580 296.1519 293.6458 292.4078
[37] 294.3693 292.6831 293.4700 293.4700 288.8154 293.3367
[43] 291.5201 288.3678 289.4001 287.7087 289.9821 293.1815
[49] 293.3735 291.5365
```

4.2 Spatial Overlay with vector maps

A typical operation is to select a certain region, that can be defined either by a rectangular window or using a vector layer defining any other shape, for instance the country boundaries or a land mask. In the next step we illustrate how to perform simple overlay operations in order to extract domains of interest using vectorial masks. To this aim, we will explore the capabilities of the `over` methods in package `sp`.

Note that one requirement of `over` is that both spatial objects bear identical PROJ.4 definition (see Section 5.3). To this aim, the 2D matrix of coordinates is converted to a spatial object, in this case a `SpatialPoints` object (type `?sp::over` for further details). The definition of a projections will be addressed in more detail in Section 5.3, where the following steps are further explained.

```
> proj4string(wrld_simpl)

[1] "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +towgs84=0,0,0"

> SpatialPoints(grid2D.rcm) -> spPoints.rcm
> proj4string(wrld_simpl) -> proj4string(spPoints.rcm)
```

The following overlay operation returns a `data.frame` containing the attributes of the world map for each of the 32300 RCM points:

```
> over(spPoints.rcm, wrld_simpl) -> ov

> str(ov)
'data.frame': 32300 obs. of  11 variables:
 $ FIPS      : Factor w/ 244 levels "", "AC", "AE", "AF", ...: 5 5 5 ...
 $ ISO2     : Factor w/ 246 levels "AD", "AE", "AF", ...: 61 61 61 ...
```

```

$ ISO3      : Factor w/ 246 levels "ABW","AFG","AGO",...: 64 64 ...
$ UN        : int   12 12 12 12 12 12 12 12 12 12 ...
$ NAME      : Factor w/ 246 levels "Aaland Islands",...: 4 4 4 4 ...
$ AREA      : int  238174 238174 238174 238174 238174 238174 ...
$ POP2005   : int  32854159 32854159 32854159 32854159 ...
$ REGION    : int   2 2 2 2 2 2 2 2 2 2 ...
$ SUBREGION : int  15 15 15 15 15 15 15 15 15 15 ...
$ LON       : num   2.63 2.63 2.63 2.63 2.63 ...
$ LAT       : num  28.2 28.2 28.2 28.2 28.2 ...

```

For instance, these are the RCM points falling in Spain:

```

> which(ov$NAME=="Spain") -> sp.index

> # Generates Fig.5
> plot(grid2D.rcm, asp=1, pch="+", col="grey",
+       xlim=c(-25,20), ylim=c(30,55), cex=.4)
> points(grid2D.rcm[sp.index, ], asp=1, pch="+",
+        col="red", cex=.4)
> lines(wr1)

```

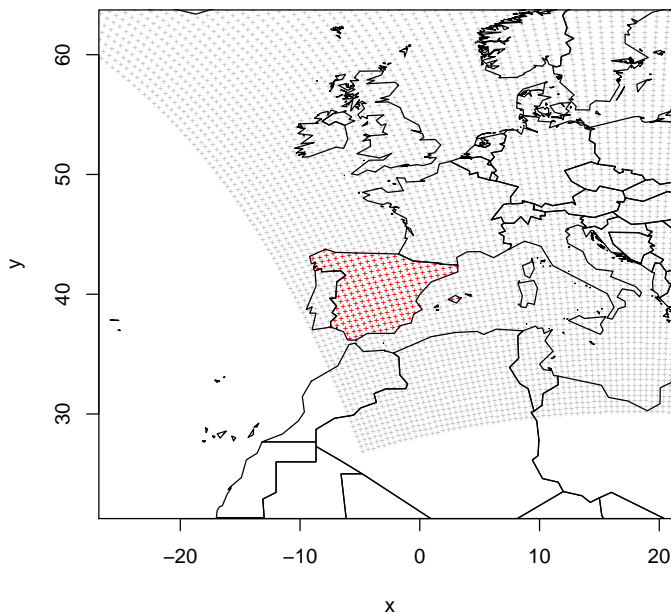


Figure 5: Example of Spatial overlay: SMHI-RCA lon-lat grid of 0.5 degrees clipped to Spain

4.3 Nearest neighbour interpolation

The nearest neighbor algorithm selects the value of the nearest point and does not consider the values of neighboring points, yielding a piecewise-constant interpolant. This is often needed in climate research in order to preserve unaltered the original values of the climate model simulations when passing from one grid to another. It is conceptually the same as the point sampling done before, although in practice its implementation is slightly different, because not always retrieving the nearest points of the output grid produces a regular grid (for instance, two adjacent points in an irregularly distributed grid may have as their nearest neighbour the same point of the output regular grid, thus leading to “gaps” in the output grid).

One possible approach is retaining only those output grid points that fall inside the domain of the input grid, and compute for each output grid point the closest input grid point. To this aim, we first define a convex hull (i.e. the smallest convex polygon that contains all input grid points), in order to reduce the number of points to compute. This can be easily computed using the function `gConvexHull` of package `rgeos`. Then, we perform an overlay operation of the output grid and the convex hull, to retain those points inside the domain:

```
> library(rgeos)
> gConvexHull(SpatialPoints(grid2D.rcm)) -> ch
> class(ch)

[1] "SpatialPolygons"
attr(,"package")
[1] "sp"

> expand.grid(lon.eobs, lat.eobs) -> grid2D.eobs
> as.matrix(grid2D.eobs) -> grid2D.eobs
> over(SpatialPoints(grid2D.eobs), ch) -> ovr
> grid2D.eobs[which(is.na(ovr)==FALSE), ] -> eobs.clip

> # Generates Fig.6
> plot(grid2D.eobs, pch=3, col="grey", cex=.1, asp=1)
> as(ch, "SpatialLines") -> hull
> points(eobs.clip, pch=3, col="blue", cex=.1)
> lines(hull, col="red", lwd=2)
> lines(wrl)
> legend("bottomright",c("EOBS grid","Clipped Area",
+ "convex hull"), pch=c(3,3,0),
+ col=c("grey","blue","red"), bg="white")
```

In order to implement this procedure, We will write a function called `interp.nn`, taking as arguments the 2D matrices of coordinates of the input and output grids and the values to represent (z). If z is not provided, the function will return the interpolated grid.

```
> interp.nn <- function(input.grid, output.grid, z = NULL, verbose = TRUE) {
+   ig <- as.matrix(input.grid)
+   og <- as.matrix(output.grid)
+   z <- z
```

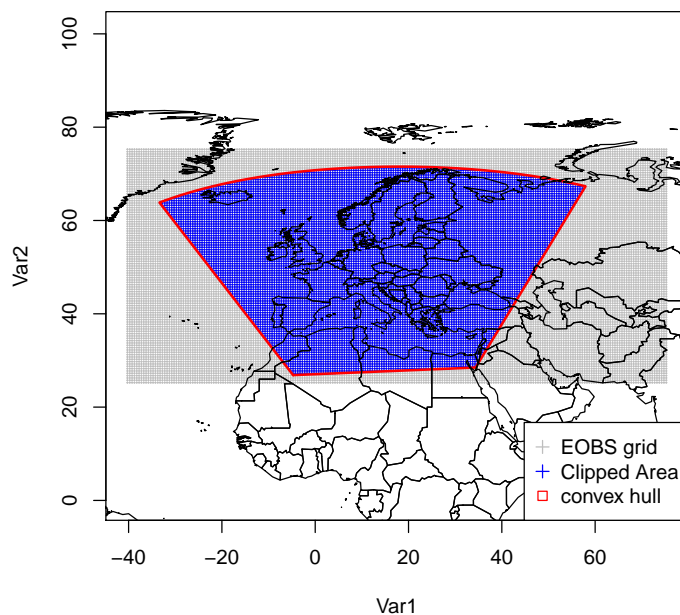


Figure 6: The EOBS grid of 0.5 degree resolution clipped to the convex hull defining the extent of the RCM grid

```

+       v <- verbose
+       if(is.null(z) == FALSE) {
+         if(length(z) != nrow(ig)) {
+           stop(paste("Length of z",length(z),
+             "does not match input grid size",nrow(ig)))
+         }
+       }
+       gConvexHull(SpatialPoints(ig)) -> ch
+       over(SpatialPoints(og), ch) -> ovr
+       og[which(is.na(ovr) == FALSE), ] -> og.clip
+       rep(NA,nrow(og.clip)) -> ind
+       for (i in 1:nrow(og.clip)) {
+         if(isTRUE(v)) {
+           print(paste("Processing point", i, "out of",
+             nrow(og.clip)))
+         }
+         which.min(spDistsN1(pts=ig, pt=og.clip[i, ])) -> ind[i]
+       }
+       if(is.null(z)) {
+         z <- "Not provided"
+       }
+     }

```

```

+         else {
+             z[ind] -> z
+         }
+         return(list("Index"=ind, "Grid"=og.clip, "z"=z))
+     }

```

In the following lines of code we will apply the function `interp.nn` to interpolate the SMHI-RCA RCM lon-lat grid onto the regular grid of EOBS.

```

> interp.nn(grid2D.rcm, grid2D.eobs, z,
+           verbose=FALSE) -> nn.grid

```

The function `nn.grid` returns a list with three elements:

Index The index of the positions (rows) of the input grid in the output grid

Grid The 2D matrix of the coordinates of the output grid

z The values of the variable `z` at each output grid location

```

> str(nn.grid)
List of 3
 $ Index: int [1:11402] 86 87 3 4 5 6 8 9 10 11 ...
 $ Grid : num [1:11402, 1:2] -4.75 -4.25 -3.75 -3.25 ...
 ..- attr(*, "dimnames")=List of 2
 .. ..$ : NULL
 .. ..$ : chr [1:2] "Var1" "Var2"
 $ z : num [1:11402] 305 305 305 305 306 ...

```

We next map the resulting nearest neighbour interpolation:

```

> cbind.data.frame(nn.grid$Grid, "TX"=nn.grid$z) -> df
> coordinates(df) <- c(1,2)
> gridded(df) <- TRUE
> colorRampPalette(rev(brewer.pal(11,"Spectral"))) -> col
> list("sp.lines", wrl, col="blue") -> l1

> # Generates Fig.7
> spplot(df, col.regions=col(21), sp.layout=list(l1),
+       scales=list(draw=TRUE), main="NN interpolation")

```

4.4 Bilinear interpolation

There are different R packages to perform bilinear and bicubic interpolation in R, like `akima` or `fields`, the latter also accompanied by didactic commented source files³. In this example we will use the `interp` function of package `akima`, intended for the interpolation of irregularly spaced data, in order to perform bilinear interpolation of the lon-lat geographical grid of the SMHI-RCA RCM onto the EOBS 0.5 degree regular grid.

³<http://www.image.ucar.edu/~nychka/Fields/Source/R/>

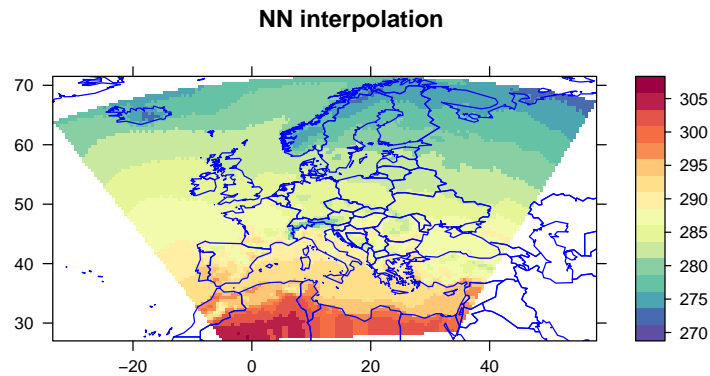


Figure 7: SMHI-RCA model output interpolated onto the EOBS regular grid of 0.5 degrees using nearest neighbor interpolation.

```

> library(akima)
> interp.old(x=x, y=y, z=z, xo=lon.eobs,
+           yo=lat.eobs) -> interp.eobs.grid

> # Generates Fig.8
> image(interp.eobs.grid, asp=1, ylim=c(20,75),
+       xlim=c(-40,60), col=col(21),
+       main="Bilinear interpolation")
> lines(wrl, col="blue")

> close.ncdf(nc.smhi)

[[1]]
[1] 6

```

5 Re-Projecting model grids

5.1 Introduction

Ideally in dynamical climate modelling, the map projection and its accompanying parameters should be chosen to minimize the maximum distortion within

Bilinear interpolation

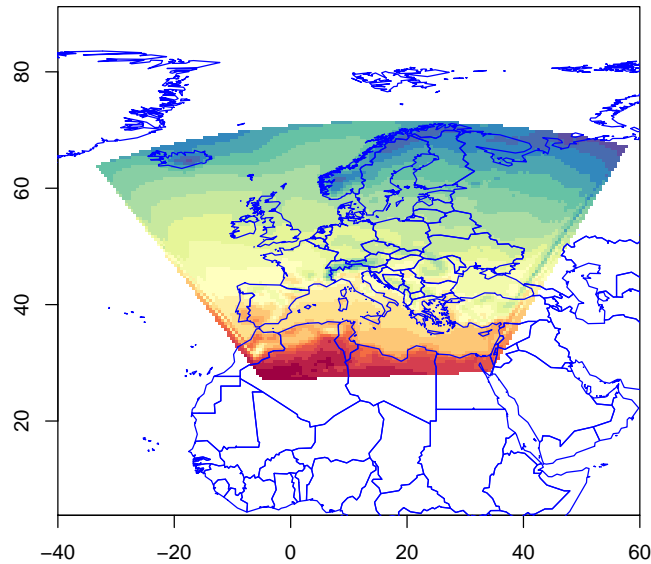


Figure 8: SMHI-RCA model output interpolated onto the EOBS regular grid of 0.5 degrees using bilinear interpolation.

the area covered by the model grids, since a high amount of distortion, evidenced by map scale factors significantly different from unity, can restrict the model time step more than necessary⁴. This is the reason the native RCM grids are referenced to different types of projections. As a general guideline, the polar stereographic projection is best suited for high-latitude domains, the Lambert conformal projection is well-suited for mid-latitude domains, and the Mercator projection is good for low-latitude domains or domains with predominantly west-east extent. The cylindrical equidistant projection is required for global simulations, although in its rotated aspect (i.e., when pole coordinates are shifted) it can also be well-suited for regional domains anywhere on the earth's surface. Among the ENSEMBLES RCM's, both the Lambert Conformal and the rotated pole (Mercator oblique) can be found.

In this practice we illustrate how to proceed in R when data need to be re-projected. We will use the resources of the PROJ.4 library, constructed for performing conversions between cartographic projections. The library is based on the work of Gerald Evenden at the USGS, but is now an OSGeo project maintained by Frank Warmerdam⁵. The library also ships with executables for performing these transformations from the command line, that we will use from

⁴WPS WRF-ARW V3: User's Guide 3-10, http://www.mmm.ucar.edu/wrf/users/docs/user_guide_V3/ARWUsersGuideV3.pdf

⁵<https://trac.osgeo.org/proj/>

the R environment.

5.2 Rotated coordinates

Usually, the rotated coordinates are provided in the variable definition of the files, so it is easy to project the simulations either in their rotated or non-rotated form. All the netCDF files stored in the ENSEMBLES database contain both the geographic lat/lon and the projected coordinates, so the representation of data is quite straightforward. In the following example we will illustrate this point using the KNMI-RACMO2 RCM data stored in the ENSEMBLES database, from which we have an example in our working directory.

```
> library(ncdf)
> list.files("./files", pattern="KNMI",
+           full.names=TRUE) -> file
> file

[1] "./files/KNMI-RACMO2_CTL_ERA40_DM_50km_1991-2000_tasmax.nc"

> open.ncdf(file) -> nc.knmi
> names(nc.knmi$dim) # dimensions

[1] "bnds"   "rlon"   "rlat"   "height" "time"

> names(nc.knmi$var) # variables

[1] "rotated_pole" "lon"      "lat"
[4] "time_bnds"    "tasmax"
```

In this step we extract the geographical coordinates:

```
> get.var.ncdf(nc.knmi, "lat") -> lat
> get.var.ncdf(nc.knmi, "lon") -> lon
```

The variables `lon` and `lat` are each one arranged in 2-dimensional matrix, because the grid is not regular, given that the native projection of the RCM is rotated.

```
> dim(lat)

[1] 85 95

> str(lat)

 num [1:85, 1:95] 26.9 27 27.1 27.2 27.4 ...
```

The spatial representation of the information contained in the file can be easily performed in R, and the tuning options are too wide to introduce all of them here. The packages `maptools` and `sp` provide many utilities for reading and handling spatial objects. In the next lines we will plot the RCM grid in geographical coordinates. We will also include the delimiting lines of world countries for visual reference. Gridded data can also be easily represented with the command `image` and contours can be overlaid using `contour`, for instance, belonging to the package `graphics`. During the next examples we will use these different approaches for data representation.

```

> as.vector(lon) -> vlon
> as.vector(lat) -> vlat
> cbind(vlon, vlat) -> coords

> # Generates Fig.9
> plot(coords, asp=1, cex=.4, col="grey",
+       pch="+", main="KNMI-RACMO2 lon-lat grid")
> lines(wrl)

```

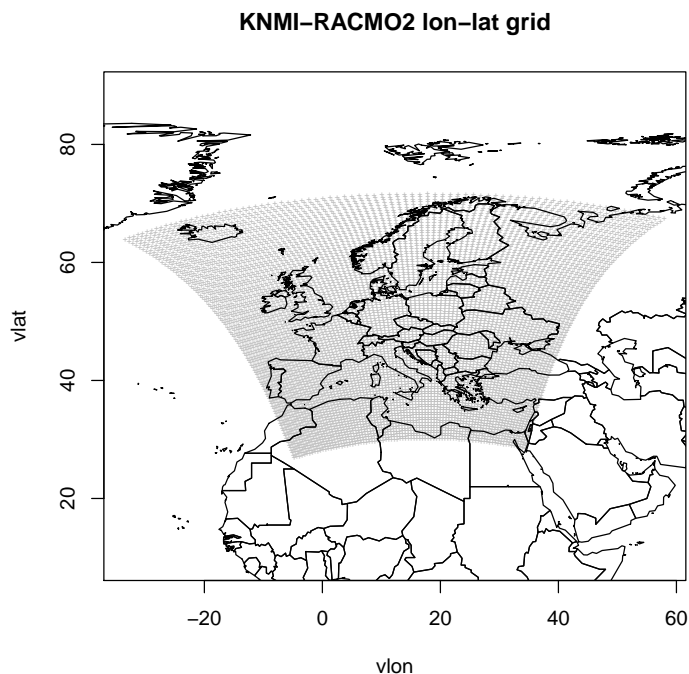


Figure 9: Geographical coordinates of the KNMI-RACMO2 RCM

And this is the representation in rotated coordinates, as provided by the netCDF file:

```

> get.var.ncdf(nc.knmi, "rlat") -> rlat
> get.var.ncdf(nc.knmi, "rlon") -> rlon
> expand.grid(rlon,rlat) -> rot.coords

> # Generates Fig.10
> plot(rot.coords, asp=1, cex=.4, col="grey",
+       pch="+", main="KNMI-RACMO2 Rotated Grid")

```

These examples are better visualized by representing the climatic variable, in this case maximum daily air surface temperature. We now calculate the mean max. temperature of the period encompassed by the file (1991-2000):

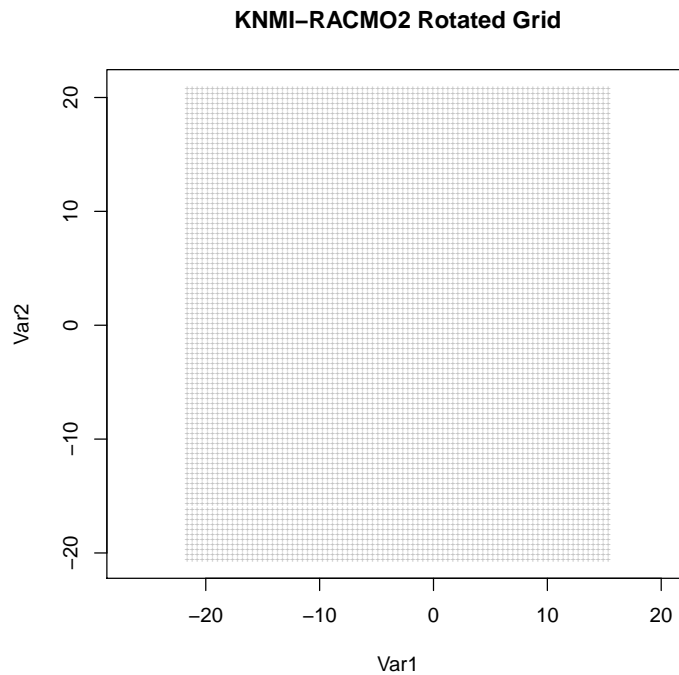


Figure 10: Rotated coordinates of the KNMI-RACMO2 RCM

```

> get.var.ncdf(nc.knmi,varid="tasmax") -> tx
> str(tx) # a 3-d array (lon,lat,time)

num [1:85, 1:95, 1:3653] 290 290 290 289 290 ...

> # applies function mean to margins 1 and 2 of the array
> apply(tx, MARGIN=c(1,2), FUN=mean) -> tx.mean
> str(tx.mean)

num [1:85, 1:95] 304 303 304 304 305 ...

> list("sp.lines",wrl) -> l1
> as.vector(tx.mean) -> t
> cbind.data.frame(coords, t) -> t.lonlat
> coordinates(t.lonlat) <- c(1,2)
> library(scales)
> library(RColorBrewer)
> rev(brewer.pal(11,"Spectral")) -> color.palette

> # Generates Fig.11
> spplot(t.lonlat, scales=list(draw=TRUE), sp.layout=list(l1),
+   col.regions=alpha(color.palette,.2), cuts=10,
+   main="Mean Max Surface Temp 1991-2000, lon/lat projection")

```

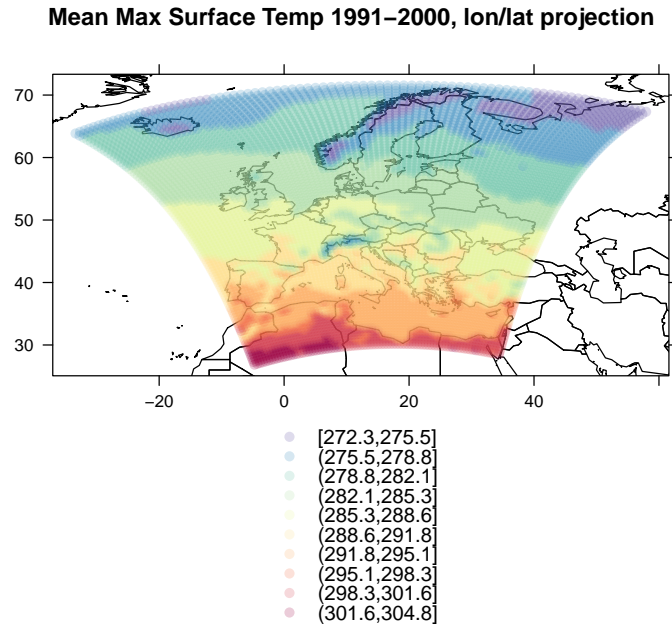


Figure 11: Mean maximum temperature (1991-2000) represented using the geographical lon-lat coordinates KNMI-RACMO2 RCM

Alternatively, the same data can be easily mapped into the rotated RCM grid, because the rotated coordinates are provided by the netCDF file, and they have been already loaded.

```
> # Generates Fig.12
> image(rlon, rlat, as.matrix(tx.mean), asp=1,
+       main="Mean Max Surface Temp 1991-2000, rot. coords",
+       col=color.palette)
> contour(rlon, rlat, as.matrix(tx.mean),
+         add=TRUE, nlevels=20)

> close.ncdf(nc.knmi)

[[1]]
[1] 6
```

5.3 Re-projecting data

However, not in all cases the rotated coordinates are contained in the netCDF files (although they are always in the ENSEMBLES models). In other cases, we might be interested in combining the climatic information with other spatially

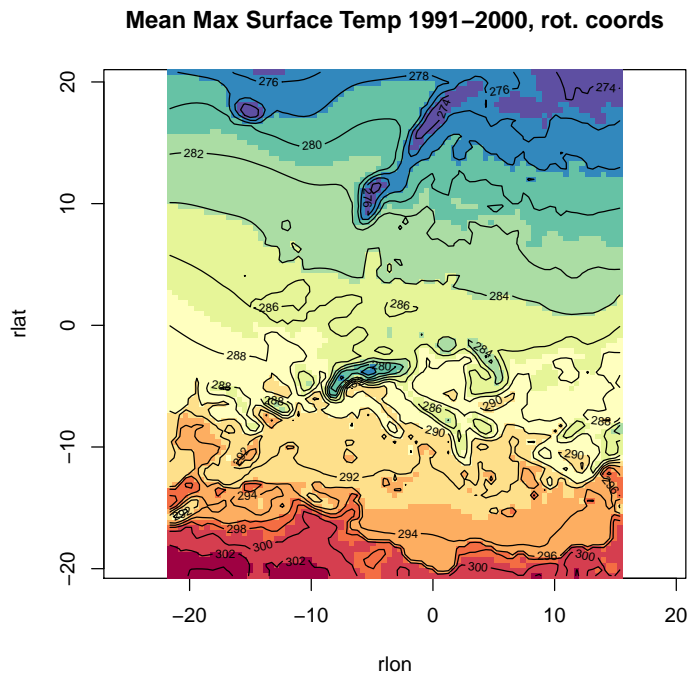


Figure 12: Mean maximum temperature (1991-2000, same as Fig. 11) represented using the rotated coordinates KNMI-RACMO2 RCM

explicit information that has different projection, for instance according to the national grids of each country. In these cases, we will need to undertake a re-projection of the data. Following with the PROJ.4 library presented in the Section 5.1, PROJ.4 strings are character vectors used to identify a spatial reference system. Using the PROJ.4 syntax, it is possible to specify the complete set of parameters that define a particular spatial reference system. In R, we can use the packages `rgdal` or `proj4` to obtain information about these parameters, for instance all the projections available, the ellipsoids and the datums, etc. In this example, we will use the `rgdal` package:

```
> library(rgdal)
> str(projInfo("proj"))

'data.frame': 129 obs. of 2 variables:
 $ name      : Factor w/ 129 levels "aea","aeqd",...: 1 2 ...
 $ description: Factor w/ 127 levels "Airy",...: 3 6 1 2 ...

> str(projInfo("ellps"))

'data.frame': 42 obs. of 4 variables:
 $ name      : Factor w/ 42 levels "airy","andreae",...: 29 36 ...
 $ major     : Factor w/ 39 levels "a=6370997.0",...: 19 17 ...
```

```

$ ell          : Factor w/ 25 levels "b=6355834.8467",...: 16 ...
$ description: Factor w/ 42 levels "Airy 1830",...: 29 37 ...

>str(projInfo("datum"))

'data.frame': 10 obs. of  4 variables:
 $ name          : Factor w/ 10 levels "carthage",...: 10 2 ...
 $ ellipse       : Factor w/  8 levels "airy",...: 8 5 5 4 2 ...
 $ definition    : Factor w/  9 levels "nadgrids=@conus",...: 4 ...
 $ description   : Factor w/ 10 levels "", "Airy 1830",...: 1 4 9 ...

```

Sometimes additional parameters (and their values) are required to define a specific projection. More generally, the PROJ.4 string will be built from a small set of parameters common to most projections. These parameters can be found here: http://www.remotesensing.org/geotiff/proj_list/. The meaning and nature of these parameters is documented online at this link: <http://trac.osgeo.org/proj/wiki/GenParms>.

Because finding the correct projection specification is often a hard task, lists still known as EPSG lists are maintained, and can be used to find the appropriate PROJ.4 definition. EPSG refers to the now-defunct European Petroleum Survey Group (EPSG), which developed a large geodetic parameter dataset as distributed with PROJ.4 software. Today, the EPSG codes are still in use, and the EPSG dataset is included in the `rgdal` package.

In the next example we will load the grid of the CNRM-RM4.5 RCM of the ENSEMBLES dataset in order to illustrate the steps (and difficulties) of re-projecting data. We will also use the EPSG codes in order to define the appropriate PROJ.4 projections.

```

> list.files("./files", full.names=TRUE,
+           pattern="CNRM") -> target.file
> open.ncdf(target.file) -> nc.cnrm

> print(nc.cnrm)
[1] "file ./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc
has 3 dimensions:"
[1] "x   Size: 93"
[1] "y   Size: 101"
[1] "time Size: 3653"
[1] "-----"
[1] "file ./files/CNRM-RM4.5_CTL_ERA40_DM_50km_1991-2000_pr.nc
has 4 variables:"
[1] "float lon[x,y] Longname:longitude Missval:1e+30"
[1] "float lat[x,y] Longname:latitude Missval:1e+30"
[1] "char Lambert_conformal[] Longname:Lambert_conformal Missval:NA"
[1] "float pr[x,y,time] Longname:Precipitation Missval:1e+30"

> names(nc.cnrm$dim) # dimensions

[1] "x"   "y"   "time"

> names(nc.cnrm$var) # variables

```

```
[1] "lon"          "lat"
[3] "Lambert_conformal" "pr"
```

We have information about the native grid of the RCM, Lambert Conformal, and hopefully also about the necessary parameters to undertake data projection.

```
> get.var.ncdf(nc.cnrm, "lon") -> lon
> get.var.ncdf(nc.cnrm, "lat") -> lat
> cbind(as.vector(lon), as.vector(lat)) -> cnrm.coords

> # Generates Fig.13
> plot(cnrm.coords, col="grey", pch=3, cex=.3, asp=1,
+      main="CNRM-RM4.5 lon-lat grid")
> lines(wr1)
```

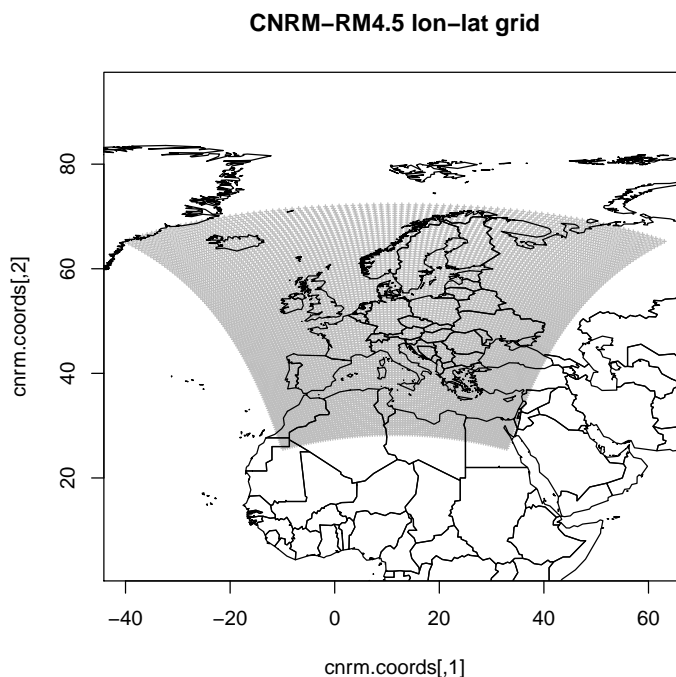


Figure 13: Geographical Lon-Lat grid of the CNRM-RM5.1 RCM of the ENSEMBLES database

We go again to the projections in http://www.remotesensing.org/geotiff/proj_list and click on the Lamber Conic Conformal. We get the following info of the required parameters for the PROJ.4 definition:

```
+proj=lcc
+lat_1=Latitude of natural origin
+lon_0=Longitude of natural origin
```



```
+k_0=Scale factor at natural origin
+x_0=False Origin Easting
+y_0=False Origin Northing
```

We next go to the projection parameters available in the netCDF file meta-data:

```
> att.get.ncdf(nc.cnrn, varid="Lambert_conformal",
+             "longitude_of_central_meridian")$value

[1] "11.5"

> att.get.ncdf(nc.cnrn, varid="Lambert_conformal",
+             "latitude_of_projection_origin")$value

[1] "50.4"

> att.get.ncdf(nc.cnrn, varid="Lambert_conformal",
+             "standard_parallel")$value

[1] "11.5"
```

It seems that we have part of the required information (longitude of the central meridian, the latitude of the origin and the standard parallel). However, the values of other required parameters (e.g., the ellipsoid, the datum, the `+towgs` parameter...) to properly undertake the transformation remain unknown. Note that the `+towgs84` tag should be used where needed to make sure that datum transformation does take place. Parameters for `+towgs84` will be taken from the bundled EPSG database if they are known unequivocally, but may be entered manually from known authorities. Not providing the appropriate `+datum` and `+towgs84` tags may lead to coordinates being out by hundreds of metres. Unfortunately, there is no easy way to provide this information: the user has to know the correct metadata for the data being used, even if this can be hard to discover. In this particular case, it may result useful to use as reference the standard lambert projections used in France. To this aim, we will use the PROJ.4 parameter definitions included in the package `gdal`, previously commented. Note that our transformation in this case might be not absolutely precise, because we have to make some assumptions, so it is advisable to have some references. In this case, we will perform the transformation also on the countries map (which is projected in the lon/lat WGS84 coordinate reference system), to check the spatial consistency of the transform.

First of all we go to the EPSG lists, using the function `make_EPSG` to retrieve the table with the EPSG codes, names and PROJ.4 definitions

```
> library(rgdal)
> make_EPSG() -> epsg
> epsg[grep("Paris.*Lambert", epsg[,2]), 1:2]

      code      note
2980 27561 # NTF (Paris) / Lambert Nord France
2981 27562 # NTF (Paris) / Lambert Centre France
2982 27563 # NTF (Paris) / Lambert Sud France
```

```

2983 27564          # NTF (Paris) / Lambert Corse
2984 27571          # NTF (Paris) / Lambert zone I
2985 27572          # NTF (Paris) / Lambert zone II
2986 27573          # NTF (Paris) / Lambert zone III
2987 27574          # NTF (Paris) / Lambert zone IV

>epsg[2980:2987, 3]
>[1] "+proj=lcc +lat_1=49.50000000000001 +lat_0=49.50000000000001
      +lon_0=0 +k_0=0.999877341 +x_0=600000 +y_0=200000
      +a=6378249.2 +b=6356515 +towgs84=-168,-60,320,0,0,0,0
      +pm=paris +units=m +no_defs"
[2] "+proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742
      +x_0=600000 +y_0=200000 +a=6378249.2 +b=6356515
      +towgs84=-168,-60,320,0,0,0,0 +pm=paris +units=m +no_defs"
[3] "+proj=lcc +lat_1=44.10000000000001 +lat_0=44.10000000000001
      +lon_0=0 +k_0=0.999877499 +x_0=600000 +y_0=200000
      +a=6378249.2 +b=6356515 +towgs84=-168,-60,320,0,0,0,0
      +pm=paris +units=m +no_defs"
[4] ...

```

It seems reasonable to use the metadata provided by the netCDF in combination with the other parameters required derived from these standard projections used in France (+towgs etc.). As we can see, the parameter "k_0", referring to the scale factor at the origin of the coordinate reference system is not equal to 1 (as one would expect from the Lambert Conformal projection definition), but slightly lower in all cases (≈ 0.999877). This is a regular feature of the mapping of some former French territories and has the effect of making the scale factor unity on two other parallels either side of the standard parallel, an information that we probably ignore, but that can be obtained at the list of projection definitions at http://www.remotesensing.org/geotiff/proj_list/, that is always advisable to check.

We specify the PROJ.4 string using the CRS (coordinate reference system) command.

```

>CRS("+proj=lcc +lat_1=50.4 +lat_0=50.4 +lon_0=11.5
      +k_0=0.999877 +x_0=600000 +y_0=200000
      +a=6378249.2 +b=6356515
      +towgs84=-168,-60,320,0,0,0,0 +pm=paris
      +units=m +no_defs")-> rcm.lambert.proj4

```

The `spTransform` methods in library `rgdal` provide transformation between datums and conversion between projections, from one unambiguously specified CRS to another, using PROJ.4 projection arguments. This is similar to using the `ptransform` function of library `proj4`. We next use the CNRM model lon-lat grid and re-project into the native Lambert Conformal Grid using this utility and the previous PROJ.4 specification. We need to assign also a specific coordinate reference system to the lon-lat grid (Fig. 13), in this case the WGS84 reference system. In case of doubts, remember that we can retrieve the corresponding PROJ.4 string from the lists included in the `rgdal` package via the `make_EPSG` command. In addition, we need to convert the coordinates to an adequate spatial class, in this case the `SpatialPoints` class of package `sp` for (irregularly spaced) points.

```

> epsg[grep("WGS 84$", epsg[,2]), 1:2]

      code      note
249 4326 # WGS 84

> CRS(epsg[249, 3]) -> wgs.crs
> SpatialPoints(cnrm.coords) -> rcm.lonlat.grid
> proj4string(rcm.lonlat.grid) <- wgs.crs
> proj4string(rcm.lonlat.grid)

[1] "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +towgs84=0,0,0"

> rcm.lambert.proj4

CRS arguments:
+proj=lcc +lat_1=50.4 +lat_0=50.4 +lon_0=11.5
+k_0=0.999877 +x_0=600000 +y_0=200000 +a=6378249.2
+b=6356515 +towgs84=-168,-60,320,0,0,0,0 +pm=paris
+units=m +no_defs

> spTransform(rcm.lonlat.grid,
+             rcm.lambert.proj4) -> rcm.lambert.grid
> summary(rcm.lambert.grid)

Object of class SpatialPoints
Coordinates:
           min      max
coords.x1 -1946129 2819935
coords.x2 -2362328 2779359
Is projected: TRUE
proj4string :
[+proj=lcc +lat_1=50.4 +lat_0=50.4 +lon_0=11.5
+k_0=0.999877 +x_0=600000 +y_0=200000 +a=6378249.2
+b=6356515 +towgs84=-168,-60,320,0,0,0,0 +pm=paris
+units=m +no_defs]
Number of points: 9393

> spTransform(wrl, rcm.lambert.proj4) -> world.trans

```

Next we plot the re-projected data in order to check the consistency of the results:

```

> # Generates Fig.14
> plot(rcm.lambert.grid@coords, cex=.2, pch=3, asp=1, col="grey",
+      main="Projected RCM Grid - Lambert Conical Conformal")
> mtext(paste("Projection",epsg[2980, 2],
+            "(EPSG code",epsg[2980,1],")"),line=.3)
> lines(world.trans, col="red")

```

As we can see from the comparison of figures 13 and 14, both the RCM grid and the map of the world countries have been adequately transformed. We will now display the precipitation field. First of all we check the units

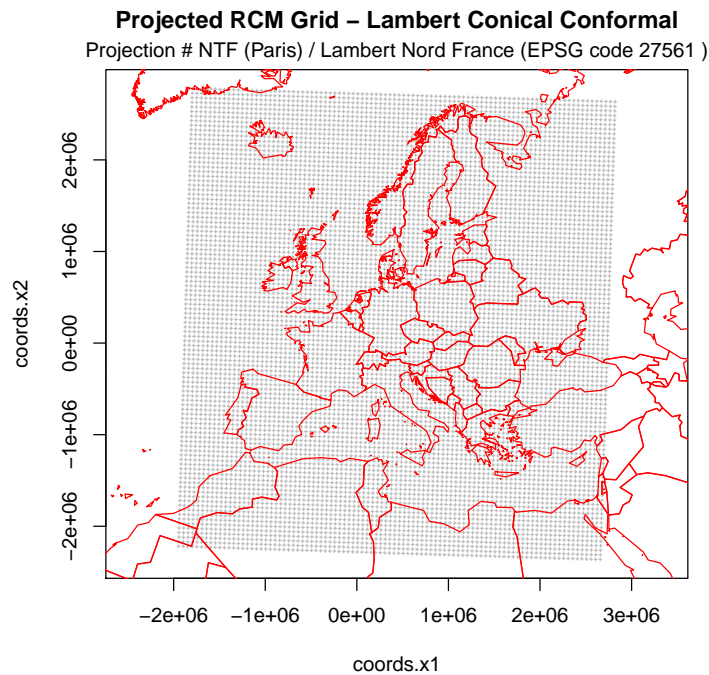


Figure 14: Projected grid of the CNRM-RM4.5 RCM

```

> get.var.ncdf(nc.cnrn, varid="pr") -> precip
> str(precip)

num [1:93, 1:101, 1:3653] 7.15e-06 1.10e-06 4.65e-07 4.26e-07 2.10e-07 ...

> att.get.ncdf(nc.cnrn, varid="pr", attname="units")

$hasatt
[1] TRUE

$value
[1] "kg m-2 s-1"

```

As we can see, precipitation is a flux. In order to convert to mm/day, we need to apply the corresponding correction factor, in this case to pass from seconds to days: $60 \text{ seconds} \times 60 \text{ minutes} \times 24 \text{ hours} = 86400$. Then, we calculate the average annual precipitation.

```
> precip*86400 -> precip.mm
```

In the next lines we will extract the vector of dates in order to aggregate annually the precipitation (as shown in Section 3). Due to the cumulative nature of this variable, we have to proceed in two steps: first we calculate the annual accumulated precipitation, and then we calculate the inter-annual mean of the period:

```

> get.var.ncdf(nc.cnrm, varid="time") -> time
> att.get.ncdf(nc.cnrm, varid="time", attname="units")

$hasatt
[1] TRUE

$value
[1] "days since 1950-01-01"

> as.Date(time, origin="1950-01-01") -> dates
> library(chron)
> years(dates) -> yrs
> levels(yrs)

[1] "1991" "1992" "1993" "1994" "1995" "1996" "1997" "1998"
[9] "1999" "2000"

> array(NA, dim=c(93,101,length(levels(yrs)))) -> annual.pr
> for (i in 1:length(levels(yrs))) {
+   precip.mm[ , ,which(yrs==levels(yrs)[i])] -> pr.year
+   apply(pr.year, MAR=c(1,2), FUN=sum) -> annual.pr[ , ,i]
+   rm(pr.year)
+ }
> apply(annual.pr, MAR=c(1,2), FUN=mean) -> annual.mean.pr

```

Now the matrix is converted to a spatial object, by binding the coordinates and the data in a single dataframe and specifying the corresponding coordinates:

```

> cbind.data.frame(coordinates(rcm.lambert.grid),
+   "Precip"=as.vector(annual.mean.pr)) -> pr.df
> coordinates(pr.df) <- c(1,2)
> list("sp.lines", world.trans) -> l1
> colorRampPalette(c("yellow","cyan",
+   "blue","purple")) -> color.palette

> # Generates Fig.15
> spplot(pr.df, sp.layout=list(l1), cuts=7, cex=1.5,
+   col.regions=alpha(color.palette(7),.15), pch=rep(15,7),
+   main="Mean annual Precipitation 1991-2000 (mm/yr)")

> # Close netCDF connection
> close.ncdf(nc.cnrm)

[[1]]
[1] 6

```

6 VALUE-THREDDS data access

6.1 Introduction to THREDDS

The THREDDS Data Server (Thematic Realtime Environmental Distributed Data Services) is a web server that provides metadata and data access for scientific datasets, using OPeNDAP, OGC WMS and WCS, HTTP, and other remote

Mean annual Precipitation 1991–2000 (mm/yr)

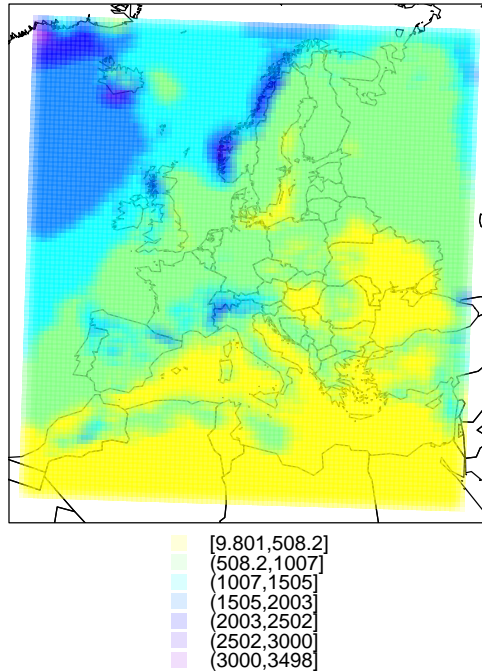


Figure 15: Representation of the mean annual precipitation (1991-2000) using the re-projected lon-lat grid of the CNRM-RM4.5 RCM onto the Lambert Conical Conformal projection


data access protocols. The mission of THREDDS is for students, educators and researchers to publish, contribute, find, and interact with data relating to the Earth system in a convenient, effective, and integrated fashion. In the context of the Cost Action VALUE, UC has established the VALUE-THREDDS service for data access. In this section we will illustrate some of the capabilities of the VALUE-THREDDS data server and will download some example files using R in order to perform some basic operations of data representation and analysis, building upon the experienced gained from the previous examples.

The link to the VALUE-THREDDS is: <http://www.meteo.unican.es/thredds/catalog/VALUE/>. There is also a User Manual of this server available to VALUE partners.

6.2 THREDDS data download

The `download.file` function of the R package `utils` (by default included in the basic R install), is a useful tool for downloading files from the internet. The first thing we need to know is the exact internet link to the file. This is explained in more detail in the VALUE-THREDDS User Manual, so we won't extend much on this issue.

1. The whole file can be directly downloaded into the local machine. This is


THREDDS4.2
THREDDS Data Server

Catalog http://www.meteo.unican.es/thredds/catalog/VALUE/Observations/Spain02_v2.1/catalog.html

Dataset: [Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)

- *Data size:* 231.6 Mbytes
- *ID:* VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc

Access:

1. **OPENDAP:** [/thredds/dodsC/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)
2. **HTTPServer:** [/thredds/fileServer/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)
3. **WCS:** [/thredds/wcs/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)
4. **WMS:** [/thredds/wms/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)
5. **NetcdfSubset:** [/thredds/ncss/grid/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)
6. **QueryCapability:** [/thredds/dqc/VALUE/Observations/Spain02_v2.1/Spain02D_v2.1_tasmax.nc](#)

Dates:

- 2010-12-09 11:22:48Z (modified)

Viewers:

- [NetCDF-Java ToolsUI \(webstart\)](#)
- [Godiva2 \(browser-based\)](#)

Figure 16: The data access window of the VALUE-THREDDS data server.

possible using the links (1) and (2) of the **Access** section of the data access window, which correspond to OpenDAP and HTTP protocols respectively (Fig. 16).

2. Links (3) and (4) give access to map coverages via Map Servers, that allow for unfiltered access to raster data.
3. Links (5) and (6) allow for querying and sub-setting the data, instead of downloading the whole file. This is illustrated in the example below.
4. In addition, files can be interactively viewed using either ToolsUI or Godiva2).

In this practice, we will concentrate of the netCDF Subset Service (Link 5 on Figure 16), wich allows downloading subsets of netCDF files, for instance by selecting areas of interest or certain time periods instead of the whole files, which are often large and slow to download.

6.2.1 netCDF subsetting

In the next example we will select a sub-domain of the EOBS gridded dataset centered on the Iberian Peninsula for the period 1970-2000 (same period as before). In this case, a number of specifications detailing the bounding box and time span of the subset must be specified in the character string used to define the URL path. The bounding box has west as its west edge, includes all points going east until the east edge. Units must be degrees east, may be

positive or negative, and will be taken modulo 360. Therefore, when crossing the dateline, the west edge may be greater than the east edge. If the grid is on a projection, the 4 corners of the lat/lon bounding box are converted into projection coordinates, then the smallest rectangle including those 4 points is used. Regarding the time interval definition, the requested time point must lie within the dataset time range. The time slice closest to the requested time will be returned. For example, a bounding box centered on the Iberian Peninsula would be defined as follows:

```
> bbox = "&north=44&south=36&east=5&west=-10"
```

Similarly, in order to define specific time ranges, both start and end times must be adequately specified as W3C strings ⁶. For instance, if we want to select the period 1991-2000, this must be specified as follows:

```
> time.start = "time_start=1991-01-01T00:00:00Z"
> time.end = "time_end=2000-12-31T00:00:00Z"
```

Further details of netCDF subsetting can be found at <http://www.unidata.ucar.edu/projects/THREDDS/tech/interfaceSpec/GridDataSubsetService.html>

```
> url = "http://www.meteo.unican.es/thredds/ncss/grid/VALUE/
+ Observations/E-OBS_v7/Grid_050deg_reg/tx_0.50deg_reg_v7.0.nc"
```

and this is the complete route to the NCSS file, including the var specification, which is always required:

```
> paste(url, "?var=tx", bbox, "&", time.start, "&", time.end, sep="") -> f

> f
[1] "http://www.meteo.unican.es/thredds/ncss/grid/VALUE
/Observations/E-OBS_v7/Grid_050deg_reg/tx_0.50deg_reg_v7.0.nc
?var=tx&north=44&south=36&east=5&west=-10&
time_start=1991-01-01T00:00:00Z&
time_end=2000-12-31T00:00:00Z"

> download.file(f, destfile="C:/VALUE/files/TX_EOBS_0.50deg_IP.nc",
+ mode="wb")
probando la URL 'http://www.meteo.unican.es/thredds/ncss/grid
/VALUE/Observations/E-OBS_v7/Grid_050deg_reg/tx_0.50deg_reg_v7
.0.nc?var=tx&north=44&south=36&east=5&west=-10&time_start=1991
-01-01T00:00:00Z&time_end=2000-12-31T00:00:00Z'
Content type 'application/x-netcdf' length 7716652 bytes (7.4 Mb)
URL abierta
downloaded 7.4 Mb
```

When the download finishes, the EOBS file is stored in our machine ready to be used.

⁶<http://www.unidata.ucar.edu/projects/THREDDS/tech/interfaceSpec/NetcdfSubsetService.html#W3Cdate>

7 A practical example

In the following example we will compare the performance of two ENSEMBLES RCMs run on the control scenario (i.e. coupled to ERA-40), using as reference the EOBS data, previously downloaded netCDF subset (Section 6.2.1). This example will focus on maximum 2 m temperature, and for the sake of simplicity we use just the 10-year period 1991-2000 and the 0.5 degree resolution scenarios, although the steps presented here can be applied to any other dataset.

We first read the file downloaded via THREDDDS in the previous section, and check that the time range and the bounding box correspond to the parameters defined in the subset query:

```
> open.ncdf("C:/VALUE/files/TX_EOBS_0.50deg_IP.nc") -> nc.eobs
> print(nc.eobs)

[1] "file C:/VALUE/files/TX_EOBS_0.50deg_IP.nc has 3 dimensions:"
[1] "time      Size: 3653"
[1] "latitude  Size: 17"
[1] "longitude  Size: 31"
[1] "-----"
[1] "file C:/VALUE/files/TX_EOBS_0.50deg_IP.nc has 1 variables:"
[1] "float tx[longitude,latitude,time] Longname:maximum temperature Missval:1e+30"

> get.var.ncdf(nc.eobs, varid="time") -> ti.eobs
> range(ti.eobs)

[1] 14975 18627

> names(nc.eobs$dim)

[1] "time"      "latitude"  "longitude"

> nc.eobs$dim$time$units

[1] "days since 1950-01-01 00:00"

> as.Date(ti.eobs, origin="1950-01-01") -> dates.eobs
> range(dates.eobs)

[1] "1991-01-01" "2000-12-31"

> get.var.ncdf(nc.eobs, varid="longitude") -> lon.eobs
> get.var.ncdf(nc.eobs, varid="latitude") -> lat.eobs
> range(lon.eobs)

[1] -9.75  5.25

> range(lat.eobs)

[1] 36.25 44.25
```

We next proceed to reading the RCM files, previously applied to other examples: the KNMI-RACMO2 and the SMHI-RCA models:

```

> open.ncdf("./files/KNMI-RACMO2_CTL_ERA40_DM_50km_1991-2000_tasmax.nc") -> nc.knmi
> print(nc.knmi)

[1] "file ./files/KNMI-RACMO2_CTL_ERA40_DM_50km_1991-2000_tasmax.nc has 5 dimensions:"
[1] "bnds    Size: 2"
[1] "rlon   Size: 85"
[1] "rlat   Size: 95"
[1] "height Size: 1"
[1] "time   Size: 3653"
[1] "-----"
[1] "file ./files/KNMI-RACMO2_CTL_ERA40_DM_50km_1991-2000_tasmax.nc has 5 variables:"
[1] "char rotated_pole[] Longname:rotated_pole Missval:NA"
[1] "float lon[rlon,rlat] Longname:longitude Missval:1e+30"
[1] "float lat[rlon,rlat] Longname:latitude Missval:1e+30"
[1] "double time_bnds[bnds,time] Longname:time bounds Missval:1e+30"
[1] "float tasmax[rlon,rlat,height,time] Longname:Daily maximum 2-m temperature Missval"

> open.ncdf("./files/SMHIRCA_CTR_ERA40_DM_50km_1991-2000_tasmax.nc") -> nc.smhi
> print(nc.smhi)

[1] "file ./files/SMHIRCA_CTR_ERA40_DM_50km_1991-2000_tasmax.nc has 5 dimensions:"
[1] "bnds    Size: 2"
[1] "rlon   Size: 85"
[1] "rlat   Size: 95"
[1] "height Size: 1"
[1] "time   Size: 3653"
[1] "-----"
[1] "file ./files/SMHIRCA_CTR_ERA40_DM_50km_1991-2000_tasmax.nc has 5 variables:"
[1] "char rotated_pole[] Longname:rotated_pole Missval:NA"
[1] "float lon[rlon,rlat] Longname:longitude Missval:1e+30"
[1] "float lat[rlon,rlat] Longname:latitude Missval:1e+30"
[1] "double time_bnds[bnds,time] Longname:time bounds Missval:1e+30"
[1] "float tasmax[rlon,rlat,height,time] Longname:Daily maximum 2-m temperature Missval"

```

In principle, the time range encompassed by these two RCM files matches the time range of the EOBS dataset, but we will check it:

```

> get.var.ncdf(nc.knmi, varid="time") -> ti.knmi
> get.var.ncdf(nc.smhi, varid="time") -> ti.smhi
> identical(ti.smhi, ti.knmi)

[1] TRUE

> identical(ti.eobs, ti.knmi)

[1] FALSE

```

The numeric time vectors are not identical, because time is defined at 00:00Z in the EOBS files, but at 12:00Z in the RCM files. However, the days are the same:

```

> as.Date(ti.knmi, origin="1950-01-01") -> dates.knmi
> as.Date(ti.smhi, origin="1950-01-01") -> dates.smhi
> range(dates.eobs)

```

```

[1] "1991-01-01" "2000-12-31"
> range(dates.smhi)
[1] "1991-01-01" "2000-12-31"
> range(dates.knmi)
[1] "1991-01-01" "2000-12-31"

```

So now we are sure the time period encompassed by the files is the same. Now we should have all datasets interpolated to the same grid for perfect comparison.

```

> get.var.ncdf(nc.knmi, varid="lon") -> lon.knmi
> get.var.ncdf(nc.knmi, varid="lat") -> lat.knmi
> get.var.ncdf(nc.smhi, varid="lon") -> lon.smhi
> get.var.ncdf(nc.smhi, varid="lat") -> lat.smhi
> cbind(as.vector(lon.knmi), as.vector(lat.knmi)) -> grid.knmi
> cbind(as.vector(lon.smhi), as.vector(lat.smhi)) -> grid.smhi
> expand.grid(lon.eobs, lat.eobs) -> grid.eobs

> # Generates Fig.17
> plot(grid.knmi, pch=3, cex=.2, col="grey", asp=1)
> points(grid.smhi, pch=3, cex=.2, col="blue")
> points(grid.eobs, col="red")
> lines(wr1)
> legend("bottomright", c("KNMI-RACMO2", "SMHI-RCA", "EOBS_0.5"),
+       pch=3, col=c("grey", "blue", "red"), bg="white")

```

It seems that SMHI-RCA and KNMI-RACMO2 share the same grid. Next we will apply the `interp.nn` function (see Section 4.3) in order to interpolate mean maximum temperatures of the RCMs onto the EOBS 0.5 degree regular grid centered on the Iberian Peninsula. To this aim, first the mean temperature values are calculated.

```

> names(nc.knmi$var)
[1] "rotated_pole" "lon"          "lat"
[4] "time_bnds"    "tasmax"

> names(nc.smhi$var)
[1] "rotated_pole" "lon"          "lat"
[4] "time_bnds"    "tasmax"

> names(nc.eobs$var)
[1] "tx"

> get.var.ncdf(nc.knmi, varid="tasmax") -> tx.knmi
> get.var.ncdf(nc.smhi, varid="tasmax") -> tx.smhi
> get.var.ncdf(nc.eobs, varid="tx") -> tx.eobs
> apply(tx.knmi, MAR=c(1,2), FUN=mean) -> tx.mean.knmi
> apply(tx.smhi, MAR=c(1,2), FUN=mean) -> tx.mean.smhi
> apply(tx.eobs, MAR=c(1,2), FUN=mean) -> tx.mean.eobs
> close.ncdf(nc.knmi)

```

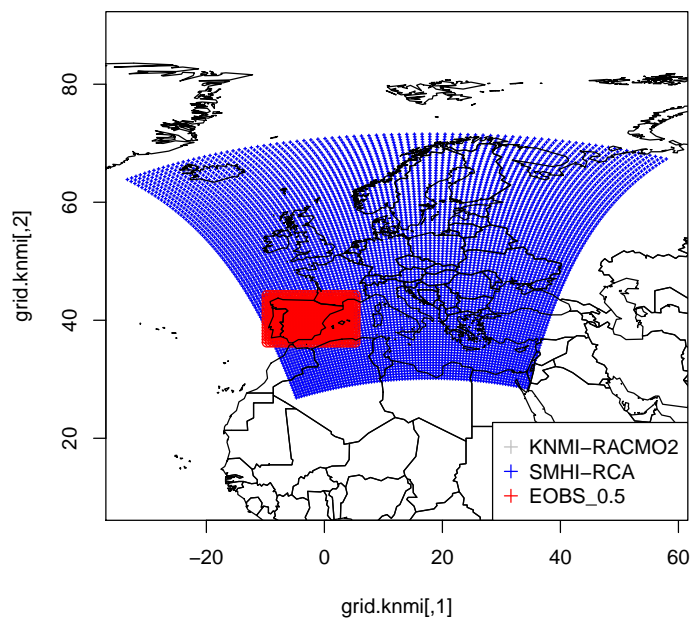


Figure 17: The RCMs grid in geographical coordinates and the regular 0.5 degree EOBS grid centered on the Iberian Peninsula

```

[[1]]
[1] 10

> close.ncdf(nc.smhi)

[[1]]
[1] 11

> close.ncdf(nc.eobs)

[[1]]
[1] 6

```

And next the mean maximum temperature fields are interpolated:

```

> interp.nn(input.grid=grid.knmi,
+ output.grid=grid.eobs, z=as.vector(tx.mean.knmi)
+ , verbose=FALSE) -> tx.knmi.interp
> interp.nn(input.grid=grid.smhi,
+ output.grid=grid.eobs, z=as.vector(tx.mean.smhi)
+ , verbose=FALSE) -> tx.smhi.interp

```

Now we prepare the data in a `data.frame`, with the first two columns corresponding to the coordinates of the EOBS grid and the remaining three columns

with the data from EOBS and the two RCMs. In addition, we make the conversion of the RCM temperatures in order to have them in degrees Celsius instead of Kelvin to match the EOBS data.

```
> cbind.data.frame(grid.eobs,
+                 "TX_EOBS"=as.vector(tx.mean.eobs),
+                 "TX_KNMI"=tx.knmi.interp$z-273.15,
+                 "TX_SMHI"=tx.smhi.interp$z-273.15) -> df
```

Once the `data.frame` is created, we create an object of the class `Spatial` by indicating the coordinates.

```
> coordinates(df) <- c(1,2)
> gridded(df) <- TRUE
> list("sp.lines", wrl) -> l1
> colorRampPalette(rev(brewer.pal(9,
+                 "Spectral")))) -> color.pal

> # Generates Fig.18
> spplot(df, sp.layout=list(l1), scales=list(draw=TRUE),
+ col.regions=color.pal(21), main="Tasmax 1991-2000, CTL scenario")
```

As we can see, EOBS is restricted to land observations. In order to have full comparability and same number of observations, we can clip the RCMs data easily by retaining only the data points which are not missing data in the EOBS dataset:

```
> which(is.na(as.vector(tx.mean.eobs))) -> missvals
> str(missvals)

int [1:216] 1 2 3 4 5 6 7 8 10 11 ...

> cbind.data.frame(grid.eobs[-missvals, ],
+                 "TX_EOBS"=as.vector(tx.mean.eobs)[-missvals],
+                 "TX_KNMI"=(tx.knmi.interp$z-273.15)[-missvals],
+                 "TX_SMHI"=(tx.smhi.interp$z-273.15)[-missvals]) -> landmask
> coordinates(landmask) <- c(1,2)
> gridded(landmask) <- TRUE

> # Generates Fig.19
> spplot(landmask, sp.layout=list(l1), scales=list(draw=TRUE),
+ col.regions=color.pal(21), main="Tasmax 1991-2000, CTL scenario")
```

7.1 Analyzing model results

7.1.1 Taylor diagram

Karl Taylor⁷ has devised a very useful diagrammatic form (termed “Taylor diagram”) for conveying information about the pattern similarity between a model

⁷Taylor, K.E. 2001. Summarizing multiple aspects of model performance in single diagram, *J. Geophys. Res.*, 106, D7, 7183–7192, 2001.

Tasmax 1991–2000, CTL scenario

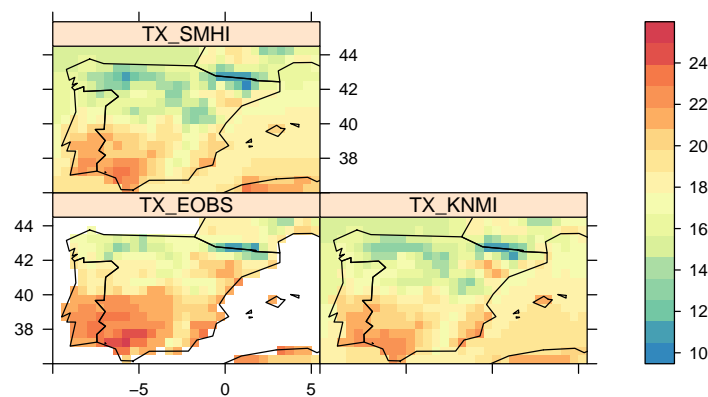


Figure 18: Mean maximum temperature (1990-1991) as simulated by the KNMI-RACMO2 and SMHI-RCA RCMs coupled to ERA-40, and as observed by the EOBS dataset.

and observations (see example below). This same type of diagram can be used to illustrate the relative accuracy amongst a number of model variables or different observational data sets. One additional advantage of the Taylor diagram is that there is no restriction placed on the time or space domain considered⁸.

The R package `plotrix` has a function to create Taylor diagrams. Further customization can be done by modifying the code of the function, but for illustrative purposes the original function `taylor.diagram` will suffice. In the following example, we compare the performance of KNMI-RACMO2 and SMHI-RCA RCMs using as reference the EOBS dataset, previously displayed in Fig. 19:

```
> library(plotrix)

> # Generates Fig.20
> taylor.diagram(ref=as.vector(tx.mean.eobs)[-missvals],
+ model=(tx.knmi.interp$z-273.15)[-missvals], normalize=TRUE)
> taylor.diagram(ref=as.vector(tx.mean.eobs)[-missvals],
+ model=(tx.smhi.interp$z-273.15)[-missvals], add=TRUE,
```

⁸http://www.ipsl.jussieu.fr/~jmesce/Taylor_diagram/taylor_diagram_definition.html#reference

Tasmax 1991–2000, CTL scenario

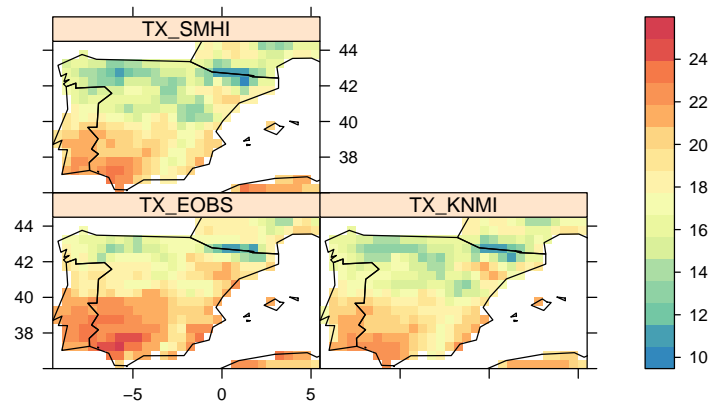


Figure 19: Same as Fig. 18, but applying a landmask to keep the same number of observations for all datasets

```
+ normalize=TRUE, col="blue")  
> legend("topright", c("KNMI", "SMHI"), pch=19, col=c(2,4))
```

7.1.2 Correlation

In this example we will compute cross-correlations between the daily temperature values of the previous RCMs. We will represent the data in the EOBS regular grid of degree resolution. The function `cor` computes the Pearson's correlation coefficient by default, although Kendall and Spearman are also implemented. More information on the correlation test can be obtained via the `cor.test` function, which also provides p-values and other statistics. First of all, we interpolate the geographical grid of the RCMs onto the regular EOBS grid using the nearest neighbor function `interp.nn` presented in Section 4.3

```
> open.ncdf("./files/elev_0.50deg_reg_v4.0.nc") -> nc.eobs.eu  
> get.var.ncdf(nc.eobs.eu, varid="longitude") -> lon.eobs.eu  
> get.var.ncdf(nc.eobs.eu, varid="latitude") -> lat.eobs.eu  
> close.ncdf(nc.eobs.eu)
```

```
[[1]]  
[1] 6
```

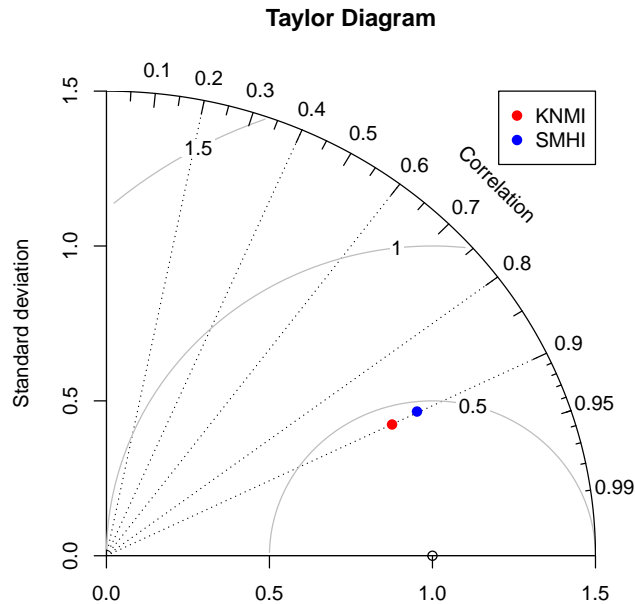


Figure 20: A Taylor diagram displaying the similarity of SMHI-RCA and KNMI-RACMO2 control runs against the EOBS observational dataset, all at 0.5 degree resolution for the period 1991-2000. Variable analysed is mean maximum surface temperature

```
> expand.grid(lon.eobs.eu, lat.eobs.eu) -> grid.eobs.eu
> interp.nn(input.grid=grid.knmi,
+           output.grid=grid.eobs.eu, verbose=FALSE) -> grid05.rcm
```

In the next step the cross-correlation is computed:

```
> dim(tx.smhi)
[1] 85 95 3653

> c() -> cor.vals
> matrix(rep(NA,85*95), ncol=2) -> coords
> colnames(coords) <- c("x","y")
> for (j in 1:95) {
+   for (i in 1:85) {
+     cor(tx.smhi[i,j,], tx.knmi[i,j,]) -> a
+     c(cor.vals,a) -> cor.vals
+   }
+ }
```

And finally the results are mapped, creating a `data.frame` that is afterwards converted to a `SpatialPixelsDataFrame` object by indicating the coordinates:


```

> cbind.data.frame(grid05.rcm$Grid,
+                 "Cor"=cor.vals[grid05.rcm$Index]) -> df.cor
> coordinates(df.cor) <- c(1,2)
> gridded(df.cor) <- TRUE
> colorRampPalette(brewer.pal(9,"Oranges")[1:8]) -> color.pal
> list("sp.lines", wrl) -> l1

> # Generates Fig.21
> spplot(df.cor, zcol="Cor", col.regions=color.pal(21),
+       sp.layout=list(l1), scales=list(draw=TRUE))

```

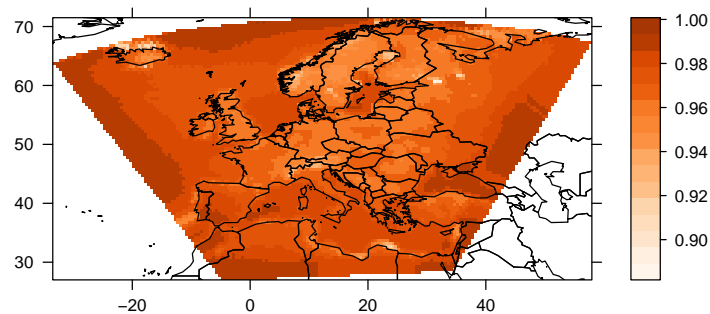


Figure 21: Cross-correlation (Pearson correlation coefficient) of air surface daily maximum temperature between KNMI-RACMO2 and SMHI-RCA RCMs coupled to ERA-40 (1991-2000)

8 Summary of R packages used

akima Interpolation of irregularly spaced data

chron Chronological objects which can handle dates and times

maptools Tools for reading and handling spatial objects. Includes the world map dataset used in the figures of this manual.

- ncdf** The R interface to the Unidata netCDF data files. It only supports netCDF until version 3. For a newer version see the next library **ncdf4**.
- ncdf4** This package provides a high-level R interface to data files written using Unidata's netCDF library (version 4 or earlier), which are binary data files that are portable across platforms and include metadata information in addition to the data sets. Using this package, netCDF files (either version 4 or "classic" version 3) can be opened and data sets read in easily. It is also easy to create new netCDF dimensions, variables, and files, in either version 3 or 4 format, and manipulate existing netCDF files. This package replaces the former **ncdf** package, which only worked with **netcdf** version 3 files. For various reasons the names of the functions have had to be changed from the names in the **ncdf** package. The old **ncdf** package is still available at the URL given below, if you need to have backward compatibility. It should be possible to have both the **ncdf** and **ncdf4** packages installed simultaneously without a problem. However, the **ncdf** package does not provide an interface for **netcdf** version 4 files.
- plotrix** Lots of plots, various labeling, axis and color scaling functions.
- proj4** A simple interface to lat/long projection and datum transformation of the PROJ.4 cartographic projections library. It allows transformation of geographic coordinates from one projection and/or datum to another.
- RColorBrewer** Provides palettes for drawing nice maps shaded according to a variable
- rgdal** Provides bindings to Frank Warmerdam's Geospatial Data Abstraction Library (GDAL) and access to projection/transformation operations from the PROJ.4 library. The GDAL and PROJ.4 libraries are external to the package, and, when installing the package from source, must be correctly installed first. Both GDAL raster and OGR vector map data can be imported into R, and GDAL raster data and OGR vector data exported. Use is made of classes defined in the **sp** package.
- rgeos** Interface to Geometry Engine - Open Source (GEOS) using the C API for topology operations on geometries. The GEOS library is external to the package, and, when installing the package from source, must be correctly installed first. Windows and Mac Intel OS X binaries are provided on CRAN.
- scales** Scales map data to aesthetics, and provide methods for automatically determining breaks and labels for axes and legends. Used in our examples to add an alpha channel (transparency) to the color legends.
- sp** Classes and methods for spatial data