HOUR OF CODE:
AN
Introduction to R

Dr. Michel d. S. Mesquita
Dr. Vidyunmala Veldore
Dr. Morgan Yarker

m2lab.org and Yarker Consulting

Contents

Welcome

Welcome to 'HOUR OF CODE - An Introduction to R'! This course has been prepared by Dr. Michel Mesquita[1], Dr. Vidyunmala Veldore[2] and Dr. Morgan Yarker[3].

The materials in this booklet are designed to be used in conjunction with our free online course (with the same title) on m2lab.org. You will benefit more fully by joining this online course. For information on downloading and installing R, please refer to Tutorial 0 in the online course.

Also, feel free to ask us any questions along the way. You will find a link on the course website, where you can post your questions.

We hope you enjoy our course!

[1]Uni Research Climate and Bjerknes Centre for Climate Research, Norway - mmeclimate@gmail.com
[2]Det Norske Veritas, Norway - vidyainteri@gmail.com
[3]Yarker Consulting, USA - morgan@yarkerconsulting.com

Chapter 1

Arithmetics in R

Welcome to the first tutorial of R. In this section we will try to use the R as a programing language and learn the simple operations that could be done be easily with few commands in R- programming environment. R is a interactive tool and has a vast helpline resources. In order to work in R we can use a terminal application (e.g.: Terminal in Linux or Mac), or using the R-console graphical interface or RStudio (Linux, Windows or Mac computers).

1.1 Getting started

This tutorial includes the following R commands (taken from the R Cheat Sheet[1]):

- Various basic arithmetic functions (+, -, etc.)
- log(x, base) computes the logarithm of x with base base
- Re(x) real part of a complex number
- Im(x) imaginary part of a complex number
- c(...) generic function to combine arguments with the default forming a vector; with $recursive = TRUE$ descends through lists combining all elements into one vector
- length(x) number of elements in x
- mean(x) mean of the elements of x
- var(x) or cov(x) variance of the elements of x; if x is a matrix or a data frame, the variance-covariance matrix is calculated
- matrix(x,nrow=,ncol=) matrix; elements of x recycle

For more information about commands in R, please refer to the ”Cheat sheet” and ”Introduction to R”[2] provided for you in Tutorial 0 online.

[1] `https://cran.r-project.org/doc/contrib/Short-refcard.pdf`
[2]`https://www.stat.berkeley.edu/~spector/R.pdf`

1.2 First encounter with R

If you have not installed R yet, please refer to our Tutorial 0, where you will get more information on how to do that. If you have installed it, then let's get started by opening R (R-console or RStudio). If you prefer to use a Terminal window (in Linux or Mac), then just type **R**.

When you open R, you might have the following information showing on the R console:

Terminal application

```
vidyunmala:Mcbook: /Bayesian-Course $ R
    R version 2.13.1 (2011-07-08)
Copyright (C) 2011 The R Foundation for Statistical Computing ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

    R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions. Type 'license()' or 'licence()'
for distribution details.

    Natural language support but running in an English locale

    R is a collaborative project with many contributors. Type 'contributors()' for more
information and 'citation()' on how to cite R or R packages in publications.

    Type 'demo()' for some demos, 'help()' for on-line help, or 'help.start()' for an HTML
browser interface to help. Type 'q()' to quit R.

    >
```

Note here that the > sign is called the **R-prompt**. It indicates that we are ready to type any commands we want to. So, let us start with a few (simple) summation, multiplication, division and power operations, so that you can see how R can be used as a calculator.

Let's type the following command (note that > is the R-prompt, so you should not type it):

```
> 8+20
```

You will find that R will give you the result 28, just as you might find in a calculator. We can also try to calculate different powers, for instance, the square of a number (e.g. 344^2). You would type this in R as follows:

```
> 344^2
```

Division is very simple to perform in R (e.g.: $\frac{289}{4}$),

```
> 289/4
```

Multiplication is done through the asterisk symbol (e.g.: 3×456),

```
> 3*456
```

You can also combine the above operations in more complex ones. In this case you would use parentheses to separate them. For instance, to calculate the following expression $\left(\frac{(8\times2987)\times4}{34\sqrt{678}}\right)^3$, you would type,

```
> ((8*2)*4)/(34*sqrt(678))^3
```

Here are examples of other operations you can make in R:

```
Natural log of base e
> log(10)

Base 10 logs; log(100,base=10) is the same
> log10(100)

Square root of a value
> sqrt(100)

Factorial
> factorial(5)

Absolute value
> abs(19/-5)

Real part of a complex number
> Re(2+3i)

Imaginary part of a complex number
> Im(2+3i)
```

R has several other functions and operations you can use (e.g.: trigonometric functions, solving differential equations, etc.). It is indeed a very powerful software. Also, R provides some useful help for its commands and functions. Fo example, if you want to find out how the standard deviation function works (**sd** function in R), you can type:

```
help(sd)
```

We highly recommend you to go through the Cheat Sheet, provided in the course, so that you can familiarise yourself with the amount of operations you can perform in R.

1.3 Matrix and Array Manipulation

R can also be used to create sequences and matrices and perform operations on these matrices. This is shown in the box below.

The first and foremost part in creating an array or a matrix is to assign the array to a variable name. For example, if we are working with a variable called temperature at 2m over a given region for 12 months. In R console, we can assign few values to a variable called **temp**,

```
> temp <- c(29.5,29.8,30.0,31.0,31.2,31.5,30.0,29.8,29.2,28.5,27.0,25.0)
```

We can get more information about this variable. For instance, to see the length of this time series, to calculate its mean and variance, type the following commands (and pay attention to the output):

```
> length(temp)
> mean(temp)
> var(temp)
```

Similarly you can apply many built-in functions (like max(), min(), etc) to this time series. The Cheat Sheet will give you several options [3].

We can also create a sequence of numbers using the command,

```
> seq(from = 1, to=10, by=1)
```

which will create the numbers 1 to 10 with an increment of 1. We can also transform the **temp** data into a 4×3 matrix in R, as follows,

```
> tempm <- matrix(temp,4,3)
```

Then, type **tempm** to see the outcome.

[3] If your work involves time series, we might consider using an R package called **zoo**. Here's an Intro to this package: `https://cran.r-project.org/web/packages/zoo/vignettes/zoo-quickref.pdf` Also, this R Time Series reference page is a wonderful resource for you to learn in details how to use several time series methods in R: `http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/src/timeseries.html`

We can then perform operations, such as addition, to each element of the matrix. For instance, if we wanted to convert the units of each row and column from Celsius to Kelvin, we could type the following,

```
> tempm <- tempm+273.16
```

Then, type **tempm** to see the outcome. Also, subtraction, multiplication, and transpose (as well as other operations) can be performed on the matrix. The transpose of a matrix can be obtained by typing,

```
> t(tempm)
```

If you want to extract a single column of a matrix, just type,

```
> tempm[,1]
```

Similarly if we would like to extract the first two columns of a matrix

```
> tempm[,1:2]
```

We can also extract the first two rows by typing,

```
> tempm[1:2,]
```

Chapter 2

Reading data in R

2.1 Getting started

This tutorial includes the following new R commands (taken from the R Cheat Sheet[1]):

- library(x) load add-on packages
- read.table(file) reads a file in table format and creates a data frame from it; the default separator sep = ”” is any whitespace; use $header = TRUE$ to read the first line as a header of column names; use $as.is = TRUE$ to prevent character vectors from being converted to factors; use $comment.char$ = ”” to prevent ”#” from being interpreted as a comment; use $skip = n$ to skip n lines before reading data; see the help for options on row naming, NA treatment, and others
- colMeans(x) fast version of column means
- max(x) maximum of the elements of x
- min(x) minimum of the elements of x
- sd(x) standard deviation of x
- print(a, ...) prints its arguments; generic, meaning it can have different methods for different objects

For more information about commands in R, please refer to the cheat sheet and Introduction to R provided for you in tutorial 0.

[1] `https://cran.r-project.org/doc/contrib/Short-refcard.pdf`

2.2 Introduction

In this chapter, we will introduce you to opening and reading data files into R. Also, you will practice opening and generating various file types (in this and other chapters), so that you are comfortable using as many data formats as possible.

This chapter will introduce you to a very common text (.txt) file of global temperature data. In chapter 3, you will use this data to generate plots in order to visualize the data. In chapter 4 you will generate arrays of randomized data for 2 cities in order to practice calculating statistical quantities. At the end of chapter 4, we will challenge you with reading in a new dataset for 2 cities and performing your own analysis on the data using R. Additionally, our optional chapter, chapter 5, will have a section on reading NetCDF datasets into R, which are very common file types for climatological data.

For now, lets start this chapter by installing necessary libraries to prepare you for the data input process.

2.3 How to install, load and unload packages in R

R comes with a few pre-installed packages. But if we would like to use some other functions (e.g.: a package to create Google Maps in R), we have to install the desired package. The R community has grown quite a lot in the past few years, and many new packages, to tackle a wide range of tasks, have been developed. So, it is good to know how to install these packages. Let's now start with some basics of package installation and learn how to load and unload them in R.

Before starting the installation of packages it is always good to find out which packages are already installed in R. So, if you want to see the packages that are installed on your computer, just type,

```
> library()
```

Now, let's say you want to install a package that makes it possible for you to read NetCDF data. This package is called **ncdf**[2]. So, to install this package, just type,

```
 > install.packages("ncdf")
```

After installation is complete, you can load the package by typing,

```
 > library(ncdf)
```

[2]There is also a newer version called **ncdf4**.

If you want to unload a package, just type,

```
 > detach("package:ncdf",unload=TRUE)
```

2.4 Opening text datasets in R

Now, let us start by opening a global mean temperature dataset in R. This is an example dataset from National Climate Data Center and could be downloaded from our online course page on `m2lab.org`

After you download the data file, you can read it by typing,

```
> inp <- read.table("globalmean.txt",header=TRUE)
```

Note that we have created a variable called **inp** to hold the temperature data. Also, remember that if your file is not located in the R working directory, you need to tell R where the file is located by typing the appropriate path. For instance,

```
> inp <- read.table("C:\MyDirectoryPath\globalmean.txt",header=TRUE)
```

Or, if you are using a Mac or Linux computer,

```
> inp <- read.table("/MyDirectoryPath/globalmean.txt",header=TRUE)
```

Now, if you want to see what **inp** looks like, you can type,

```
head(inp)
```

This will show you the first few lines of this data set.

We can now create a seasonal time series of the temperature data you have just read in R. We can do this by converting the third column of **inp** into a 12×128 matrix, that is, 12 months by 128 years. This will make it easier for us to pick specific months to create seasonal averages. For example, if we want data for June, July and August (JJA), we would then specify the columns containing these months.

Let us try to do this exercise ourselves and find out the mean, standard deviation, max and min value of the JJA time series.

Note that the dataset might contain undefined (or spurious) values and these values should not be included in the analysis. We will also tell R that such values should not be included in our analysis. We call these numbers as NA.

So, let's type the following commands,

```
Retrieve the temperature values from the file, 3rd column
> temp <- inp[,3]

Matrix creation of yearly data for 128 years
> tempm <- matrix(temp,12,128)

Defining undefined values as NA
> tempm[tempm==-999.0000] <- NA

Creating JJA time series
> tempjja <- tempm[6:8,]

Creating a seasonal mean of JJA series
> temp_jjamn <- colMeans(tempjja,na.rm=TRUE)
```

These commands will give us a seasonal time series of global surface temperature. We can now apply some operations to this time series to find out, for example, the maximum, minimum, and standard deviation for the given 128 years starting from 1880 till 2007. So, type the following now,

```
Maximum value of the seasonal means
> max(temp_jjamn)

Minimum value of the seasonal means
> min(temp_jjamn)

Standard deviation for the JJA season over 128 years
> sd(temp_jjamn)
```

Chapter 3

Plotting in R

3.1 Getting started

This tutorial includes the following new R commands (taken from the "R Cheat Sheet"[1]):

- summary(a) gives a summary of a, usually a statistical summary but it is generic meaning it has different operations for different classes of a
- seq(from,to) generates a sequence by = specifies increment; $length$ = specifies desired length
- plot(x, y) bivariate plot of x (on the x-axis) and y (on the y-axis)
- points(x, y) adds points (the option $type$ = can be used)
- lines(x, y) adds lines (the option $type$ = can be used)
- legend(x, y, legend) adds the legend at the point (x, y) with the symbols given by legend
- title() adds a title and optionally a sub-title
- hist(x) histogram of the frequencies of x

The following parameters are common to many plotting functions:

- $add = FALSE$ if TRUE superposes the plot on the previous one (if it exists)
- $axes = TRUE$ if FALSE does not draw the axes and the box
- $type$ = "p" specifies the type of plot, "p": points, "l": lines, "b": points connected by lines, "o": id. but the lines are over the points, "h": vertical lines, "s": steps, the data are represented by the top of the vertical lines, "S": id. but the data are represented by the bottom of the vertical lines
- $xlim$ =, $ylim$ = specifies the lower and upper limits of the axes, for example with $xlim = c(1, 10)$ or $xlim = range(x)$
- $xlab$ =, $ylab$ = annotates the axes, must be variables of mode character
- $main$ = main title, must be a variable of mode character
- sub = sub-title (written in a smaller font)

For more information about commands in R, please refer to the "cheat sheet" and "Introduction to R" provided for you in Tutorial 0.

[1] `https://cran.r-project.org/doc/contrib/Short-refcard.pdf`

3.2 Introduction

This chapter will use the same global temperature data file you read into R in the previous chapter. It is important to generate several different plots in order to visualize the data in a meaningful way. In this chapter, you will generate line plots, bar plots, histograms and scatter plots. You will also practice adding details to your plots including legends, axis titles, and color.

3.3 Line, Scatter, and Histogram in R

Before starting this section, an important information for plotting in R. When we create a new plot the previous plot will be overwritten, however one would like to save all the previous graphs to do a comparison analysis. The useful command for this is **x11()** for Linux, **windows()** in Windows systems and **quartz()** on Mac computers. In any case if you want to clear your workspace from all the cluttering then you can use, **rm(list=ls())**, this command removes all the variables in the workspace.

Let us again use the same globalmean.txt and plot some temperature analysis. Please, type the following commands to read in the data and calculate seasonal means (as we did in the previous Chapter),

```
> rm(list=ls())
> quartz()
> inp <- read.table("globalmean.txt",header=TRUE)
> temp <- inp[,3]
> tempm <- matrix(temp,12,128)
> tempm[tempm==-999.0000] <- NA
> tempjja <- tempm[6:8,]
> tempjja <-colMeans(tempjja,na.rm=TRUE)
```

Similarly, let's create the March-April-May (MAM) series and compare it with the JJA season. We will also make line plots for the two seasons and do some preliminary assessment. First, let's create the MMA series,

```
> tempmam <- tempm[3:5,]
> tempmam <- colMeans(tempmam,na.rm=TRUE)
```

Then, we will create a vector containing a sequence of years from 1880 to 2007, which will be used in our plot. So, let's type the following,

```
> year <- seq(1880,2007,by=1)
```

Now, we are ready to make our first plot,

```
> plot(year,tempmam,type="l",col="blue",ylab="",xlab="")
```

The above command plots the first seasonal time series of temperature for MAM as a line plot in blue. Then, to overlay the JJA time series (in red), we type,

```
> points(year,tempjja,type="l",col="red",ylab="",xlab"")
```

If you want a scatterplot, instead of a line plot, then just use $type = "p"$ in the plotting commands above. Let us now add a title, labels to the $x-$ and $y-$axes, and a legend. We can do this by typing the following commands,

```
> title(main="Global Mean Temperature in NCDC for JJA and DJF seasons",
    xlab="year",ylab="Temperature in deg C",cex=1.5,pch=20)
> legend('topleft','groups',c("MAM","JJA"),lty=c(1,1),col=c('blue','red'),
    ncol=3,bty="n")
```

Also, if you want a statistical summary of the two time series, you can use the following commands,

```
> summary(tempmam)
> summary(tempjja)
```

Next, let us now make some histogram plots. This is simply done as follows,

```
 > c1=hist(tempmam)
 > c2=hist(tempjja)
```

Chapter 4

Statistics in R

4.1 Getting started

This tutorial includes the following new R commands (taken from the "R Cheat Sheet"[1]):

- cor(x) correlation matrix of x if it is a matrix or a data frame (1 if x is a vector)
- lm(formula) fit linear models; formula is typically of the form response $termA+termB+...$; use $I(x * y) + I(x2)$ for terms made of nonlinear components

For more information about commands in R, please refer to the "Cheat sheet" and "Introduction to R" provided for you in Tutorial 0.

4.2 Introduction

In this section, we will be learning how to calculate a few statistical quantities using R. First, we will construct some fictitious datasets for you to work with throughout this section. Later on, you will have the chance to work with some real data. Let's start by constructing our fictitious datasets. We will create some temperature data for two cities, which we will call CityA and CityB, both of which will be normally distributed. So, go ahead and open the R software that you installed and type the following commands:

```
1 > set.seed(100)
  > day = 1:200
3 > CityA = rnorm(200, 24.0, 10.0)
  > CityB = rnorm(200, 15.0, 5.0)
```

In line 1, you noticed that we used the command *set.seed*(100). This was used so that we could generate random numbers following a specific seed, which means that if you repeat the *rnorm* commands once again, you will get the same result. This also means that the results we show you in this Tutorial will be the same you will obtain on your computer - because we are using the same seed. If you do not use a seed, then you will generate different results. Seeds are useful to use, especially when you want to compare results with others. You can also try and play with different seed numbers.

In lines 2-4, we then created a time vector from 1 to 200. You can think of it as a vector of days with length 200 (from day 1 to 200). Then, we generated some normally distributed data for CityA with size 200, mean temperature of 24.0 degrees Celsius (you can use Fahrenheit if you prefer) and standard deviation of 10.0 degrees Celsius. For CityB, we used a similar approach, but with a lower temperature and lower standard deviation.

[1] `https://cran.r-project.org/doc/contrib/Short-refcard.pdf`

4.3 First look at our data

Now that you have generated some fictitious temperature data for two cities, let's try to get some information about our datasets. The following activity will give you the chance to do that:

Activity 1
When you are analysing some data, it is always good practise to plot your data in order to get a feeling for what you will be working with. It is also recommended that you get some general statistics out of it, such as the mean, the maximum, the minimum and other values. These will help you to better understand your data. First, let's take a look at the temperature time series that we created for CityA:

```
> plot(day, CityA, 'o')
> summary(CityA)
```

What is the maximum and minimum temperatures that you obtained for CityA? Do you think this dataset is realistic? Now, try to do the same for CityB. How are the two datasets different in terms of their variability?

We hope you enjoyed this short activity. It helped us understand a little bit more about the data we will be working with. Below, you will find some of our comments on this activity.

Comments on Activity 1
The dataset for CityA has a lot of variability in it. You may have noticed that the maximum temperature reaches almost about 50 degrees Celsius in a couple of days. This is because we created these data to have high standard deviation. The reason why we did this, it is because we wanted to illustrate the idea of outliers in the data. For example, suppose you received data from a weather station of a certain city. Some of the measurements may contain errors and it is always a good idea to check these. These errors may be unrealistically too high or too low - and these are called outliers.

R has a great way to check for outliers and this is through the boxplot tool. When you make a boxplot in R, if your data contain possible outliers, R will plot these outliers as a circle in your boxplot. Try the following command and see if you can spot some outliers:

```
> boxplot(CityA)
```

We found at least four outliers in the higher end of our boxplot and two in the lower end. If this was a real research case, you might have to consider whether you should keep or exclude these points from your data. Sometimes, these points may indicate that there were some weather extremes for that specific city. If this was the case, then you might consider keeping these points, since they will provide you with information on extremes. By the way, did you find any outliers for CityB?

4.4 Correlation

Another aspect you might want to investigate is whether there are correlations between the two cities. You can check that in R by using the command *cor* as follows:

```
> cor(CityA, CityB)
> plot(CityA, CityB)
```

After you type these commands, you will see that there is no correlation between them. The value obtained was around 0.09, which is very low. When you plotted both datasets, you may have also noticed that there was a poor relationship between them. If you need to determine the uncertainty in your correlation estimation, you can use the following command:

```
> cor.test(CityA, CityB)
```

This will tell you that the p-value obtained is 0.189. When the p-value is higher than 0.005, it means that your correlation is not significant at the 95% confidence interval. That command will also show you a confidence interval for your correlation estimation, that is, your correlation is 0.09 and your confidence interval, which is not significant, goes from -0.05 to 0.23 (to 2 decimal places).

4.5 Linear trends

Let's now take a look at these data to see if there are any linear trends in them. We do this by fitting a linear regression model to our dataset. This is very straightforward to do in R:

```
> CityA.lm = lm(CityA ~ day)
> summary.lm(CityA.lm)
```

What we have done here is to regress the temperature data against time: $y = ax + b$, where y is our dependent variable (temperature) and x is our explanatory variable (time). If there is a trend, then the temperature will be dependent on time. a and b represent the slope (trend) and the intercept (where the trend line crosses the y-axis).

When we summarised our linear model, the p-value given was 0.9519, which was too high. Since this value was higher than 0.005, this meant that our trend was not significant at the 95% confidence interval. So, there were no trends for CityA.

Activity 3
Are there any temperature trends in CityB?

Comments on Activity 3
Which p-value did you obtain for your linear model for CityB? Was it higher than 0.05? Well, we tested CityB and we could not find any trends that were significant at the 95% confidence level.

Well, let's create a fictitious trend so that you understand how that works. Try the following command, please:

```
> seq(0, 1.99, 0.01)
> CityB2 = CityB + 4* seq(0, 1.99, 0.01)
> plot(day, CityB2, 'o')
```

So, what we have done is to add a linear trend to our time series: a trend that is 4 times 0.01 Celsius increase per day. Let's check what our linear model tells us about this:

```
> CityB2.lm = lm(CityB2~day)
> summary.lm(CityB2.lm)
```

Well, now our linear model shows a p-value of 0.000000009! This means that this value is lower than 0.05, and therefore our trend is statistically significant at the 95% confidence interval! But how much is this trend? Well, we can find out from that summary we got in R. The summary says that our intercept is 14.88 (to 2 decimal places) and that our slope is 0.04 (to 2 decimal places). Our regression equation is thus: $y = 0.04x + 14.88$. This means that our trend (slope) is of 0.04 degrees Celsius per day. This makes sense, since we added a fictitious trend of 4*0.01 to our dataset! Let's plot this trend line:

```
> plot(day, CityB2, 'o')
> lines(0.04*day+14.88)
```

Note that we used our regression equation (shown above) to make the plot. We also used the command *lines*, so that the trend line could be added to our existing plot.

Activity 4

We have provided you with observed daily temperature data for 2 cities in the United States, provided by NOAA. You can download the file on the course website. As your final assignment for the course, we challenge you to read the data file into R and perform a variety of analysis on the data until you feel you have learned something interesting about the dataset. Consider trying to generate one plot and one statistical quantity for the data at minimum. Report what you learned in our final assessment exercise in the online version of the course.

Please note: The USA frequently reports their temperature values in fahrenheit rather than Celsius. You may find it beneficial to convert the temperature values to Celsius using the formula $(°F - 32) \times 5/9 = °C$

Chapter 5

(Optional) More activities to try

5.1 Introduction

Congratulations on completing the course! Since our goal was to introduce you to R in a short period of time, we could only cover the basics in Chapters 1 through 4. However, R is a very powerful tool and there is a lot more you can do! In this chapter, we have provided some optional "going further" activities for you to try. If you work with climatological data, this section introduces you to reading and analyzing NetCDF files, which can be very useful for you. We have also provided a few links to external websites that show you a couple different options for creating map plots in R. We hope you find these additional activities useful!

5.2 Working with NetCDF datasets

NetCDF files (network common data form[1]) are most commonly used in climate research applications. Most of our present day observations and models are available in this format. Since, our temperature, rainfall, sea level pressure etc datasets are often array-oriented and have four dimensions say *(x,y,z,t)*, the NetCDF format makes it easier to handle such data. So, let's read a few NetCDF files in this section.

To start with we need to load the netcdf package in R and we also need a netcdf data file to work with. We can download a sample NetCDF dataset from,

`ftp://ftp.cdc.noaa.gov/Datasets/cpc_us_hour_precip/precip.hour.2002.nc`

[1]For more information, visit `https://en.wikipedia.org/wiki/NetCDF`

The dataset is hourly precipitation gridded and smoothed over the United States from 20N-60N and 220E-297.5E, similarly we can download other datasets from NOAA. To open these datasets in R, we need to first load the ncdf package. So, let's do this now,

```
Load the NetCDF package
> library(ncdf)\\

Open the file and assigning it to ifile
> ifile <- open.ncdf("precip.hour.2002.nc")

Get information about the input file you've just opened
> print(ifile)
```

Note here that the variable has three dimensions: latitude, longitude and time, and all these three dimensions can also be viewed as variables in R, similar to other climate visualization applications. Our next step is to get the variable **precip** (rainfall) in our working environment and view the data summary. So, let's extract the **precip** variable,

```
> precip <- get.var.ncdf(ifile,"precip")
```

Note here the dataset has 33 x 21 (longitude x latitude) points and 6552 time points (i.e., from 1 January 2002 to 30 September 2002). The information about the dimension and the details of the dataset could be obtained by *dim(precip).* Let us do some simple calculations by averaging it over the whole US region and analysing the timeseries data.

```
> prmean <- colMeans(colMeans(precip[,,1:6552],na.rm=TRUE),na.rm=TRUE)
```

This will calculate the mean over the latitude and longitude dimensions. Note that if we would like to find the number of undefined values present in the dataset we can check this with,

```
> any(is.na(precip))
```

We can check the **length(prmean)** of the dataset, which should be equal to 6552 time points. Other operations like **mean**, **max**, **sd** can also provide further information.

5.3 Autocorrelation

Note: this section is based on the examples and data given in Chapter 4.

Autocorrelation tells you some information about whether there are any correlations within your dataset. For example, let's suppose that the temperature today is 20.0 degrees Celsius. Would you expect the temperature tomorrow to be around the same value? If the weather conditions do not change much, that would definitely be the case. Now, do you think that the temperature today could influence the temperature in the next 5 days? How can you measure that? You can check this through the autocorrelation function.

Activity 2
The autocorrelation function in R is called "acf" and we can check whether there are correlations within the data for CityA, by typing the following command:

```
> acf(CityA)
```

If you get a spike that its higher than the blue lines, it means that your data is autocorrelated (and statistically significant) for that specific day, given in the x-axis. Do you see any autocorrelations in these data?

The autocorrelation function is also a good way to see whether your data is independent or not. If there are correlations, it means that your data is dependent. We will come back to this point later on in our course when we start using climate time series data in Bayesian models.

Comments on Activity 2
You may notice that there is a spike for day 0. This is because if you correlate your CityA time series with itself for the same days: this will give you a perfect correlation of 1.0. But when you start correlating it with a lag between the days, then this is what the remaining of the ACF plot will be telling you.

When we made our ACF plot, we noticed at least three statistically significant spikes: one in day 1, another in day 9 and another one in day 19.

What about CityB? Do you find any autocorrelations in that dataset?

5.4 Creating maps in R

In this section, we will see how we can plot maps using R. We will be working with an excellent website where you will learn how to make plots for different parts of the world. This also includes an interesting R package for you to make Google-like plots!

Before we get started, we need to show you how you can install new packages in R. During the map Tutorial, you will find commands like this one:

```
> library(maps)
```

If you do not have the "maps" package in your R installation, this means that you will need to install that specific package. This is not difficult to do in R. For example, to install the package "maps," you would type the following in R:

```
> install.packages("maps")
```

You will notice that R will then show you a window with options from where you would like to download that package from. For example, if I am in Norway, I would choose a location that is close to Norway. This means that the package download will be faster for me. So, choose a location that is closer to you. After you choose the location, R will do the installation for you! You can then test to see if the package was installed by typing:

```
library(maps)
```

If the package was installed properly, then you will get no error messages.

Now, visit this website: http://www.molecularecologist.com/2012/09/making-maps-with-r/, which teaches you how to make plots in R. After you finish studying the website, can you try to make a map plot of your country or of the region where you are living at the moment?

5.5 Creating maps with Google Maps in R

Now that you have had experience creating maps in R, you can go even further and create plots using Google Maps. Click here for a link to the tutorial:

http://yarkerconsulting.com/index.php/blog/15-google-maps-and-r