Learning to interact with R

Dr. Kyle Dexter
Lecturer, School of GeoSciences
University of Edinburgh
kyle.dexter@ed.ac.uk

(Many thanks to Luke Smallman and Olivier Missa for letting me pirate their material!)

Working with R is rewarding in the long run, but takes a little getting used to. In this practical, we are going to spend some time learning the basics of R. It will be a little tedious, but getting this step right will save you a lot of troubles and headaches later. If you are already familiar with R, read on anyways, as you may learn a few tricks and useful tips along the way.

Table of Contents

1.1 Obtaining R

R is free and if you wish to install it on your personal computer, it is very easy to do so.

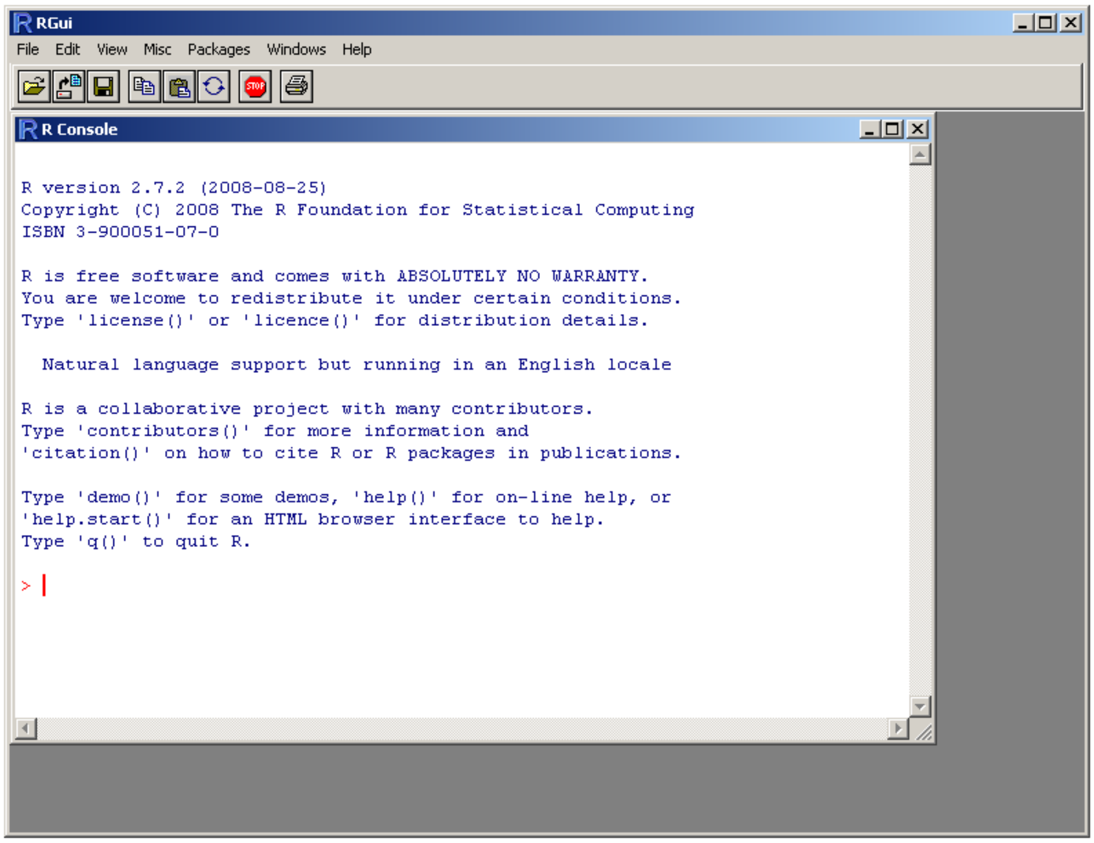
Just follow the link http://cran.r-project.org/

What you need will be in *Download and Install R*. The current up-to-date version of R will then be available for download (stay clear of the development version as it still needs to go through rigorous testing). Select your computer environment (PC, Mac or Linux) and choose the *base* install. The easiest form of install is then to download and run the setup program, which either ends with *.exe* (for PC) or *.dmg* (for Mac). Then sit back and relax (occasionally answering a few questions from the setup program). After a few minutes R will be up and running on your computer.

1.2 Starting & Quitting R

If R is already installed on your computer, then all you need to do to start R is to click on the R shortcut (once from the *Start/Programs* menu or the *Quick Launch* bar, and twice from a desktop icon). On the University Computers in the MRes suite, the R icon can be found on the ***Application Launcher***, in the ***Math. Applications*** folder.

After a few seconds, the main R window will appear (called **RGui**) and within it another R window called the **R Console** (Figure 1.1). Typically the **R Console** starts with some information in blue `Courier` font, for instance at the top: the particular R version being used and its release date. Next comes a few commands in single quotation marks about the details of the licence agreement (**`licence()`**) and how to cite R in your publications (**`citation()`**) to name only two.



```
R version 2.7.2 (2008-08-25)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

> |
```

Figure 1.1 The main **RGui** window and the **R Console** window within it at start-up.

At the bottom of the **R console** you will find in red the command prompt, "**>**", which indicates that R has completed its previous tasks (in this case start-up and initialisation) and is waiting for further instructions.

Out of curiosity, type in the commands that appeared on the **R Console** at start-up and press *Enter*. Most instructions will produce output in the **R Console** straight after the instructions themselves, *e.g.* `licence()` and `citation()`. By convention, their output will always be in blue font while the commands you type (or send to R via a script file) will always be in red. This provides a convenient way to separate your commands from the results returned, but if you would prefer other colours or fonts you can change them in the **RGui** preferences (menu ***Edit*** / ***GUI preferences …***).

Some instructions, however, will produce output in a separate window.

- `contributors()` opens an **R Information** window with a list of contributors and a brief history of the R project.
- `demo()` opens an **R demos** window with a list of available demonstrations (showcasing some of R capabilities), which you can then access from within the **R Console**, typing `demo` again but entering the name of the demo in-between brackets, *e.g.* `demo(graphics)`.
- `help()` opens an **R help** window, which is actually an independent programme allowing you to navigate through the main R commands, learn their syntax and how they are used. You can leave this R help program open and return to R by using the ***Alt-Tab*** combination.
- `help.start()` opens your preferred web browser and displays an R search engine, working offline (*i.e.* from files stored on your computer) where you can search for information (on your version of R and your installed packages). See Section 1.4. Asking for Help.

Quitting R is very straightforward. You can either:

- Type `q()` in the **R console** and press *Enter*.
- Click on the cross (**x**) in the upper right corner of the main **RGui** window or **R Console** window.
- Select the ***Exit*** option in the ***File*** menu.

Any of these operations will trigger the quit routine. A pop-up dialogue will then appear asking you if you want to save the Workspace image. This image holds a copy of all the objects (data, results) that you have been working on and that are still active at the end of your session. If you don't want to repeat all your analyses and may need to work on them at a later (not too distant) date, saving the workspace image can be very useful indeed.

1.3 Two ways to use R

R is a very rich environment for working with data, but it hides this fact under a stern and austere appearance (the blank **R console**). It is actually very flexible and allows you to get to the same result in many different ways (whichever feels more natural to you). As long as you use the right sequence of instructions, there is actually no end to what you can achieve in R.

The simplest way to use R (not doing it truly justice though) is ***as a calculator***. All the usual mathematical operations (+, -, /, *) work. For instance:

```
> 4 + 12
[1] 16
```

```
> pi * 342.71
[1] 1076.655
```

`pi` being here a pre-defined constant in R (equal to 3.141592653589793)

```
> ( (100-3) / (95-2) ) * (1.5^2)
[1] 2.346774
```

"^" being used to raise a number to a certain power, here 1.5 to the power of 2.

```
> log(100, 10)
[1] 2
```

The base-10 logarithm of 100. You can conversely use `log10(100)`

```
> log(100)
[1] 4.60517
```

The natural logarithm (or base *e*) of 100.

```
> log(100,2)
[1] 6.643856
```

The base-2 logarithm of 100. You can conversely use `log2(100)`. Beware though: other bases (besides 2 and 10) will need to be specified as the second argument of the log command, there are no shortcuts for them.

```
> exp(1)
[1] 2.718282
```

Equivalent to, e^1, *Euler's* constant (the base of natural logarithms) raised to the power of 1. In other words, *Euler's* constant itself.

Box 1.1 Common Mathematical Functions

`pi`	the value of ***pi***, 3.141593
`log(x)`	log to base *e* of ***x***
`log(x,n)`	log to base *n* of ***x***
`log2(x)`	log to base 2 of ***x***
`log10(x)`	log to base 10 of ***x***
`exp(x)`	exponential function of ***x***, *i.e.* e^x
`sqrt(x)`	square root of ***x***
`abs(x)`	absolute (unsigned) value of ***x***
`ceiling(x)`	smallest integer equal to or greater than ***x***
`floor(x)`	largest integer equal to or smaller than ***x***
`round(x, digits=0)`	rounds ***x*** to the nearest integer (since `digits=0`)
`trunc(x)`	integer part of ***x***
`signif(x, digits=6)`	returns ***x*** with 6 significant digits as per the scientific notation.
`sin(x)`	sine of angle ***x*** expressed in radians (i.e. 2*pi = 360 degrees)
`cos(x)`	cosine of angle ***x*** (expressed in radians)
`tan(x)`	tangent of angle ***x*** (expressed in radians), = sin(*x*)/cos(*x*)
`asin(x), acos(x)`	arc-sine, arc-cosine of ***x*** (expressed as a real or complex number)
`atan(x)`	arc-tangent of ***x*** (expressed as a real or complex number), useful to convert a slope (from a straight line equation) into an angle.
`factorial(x)`	*x*! (for integers only), equivalent to *x*.(*x*-1).(*x*-2) . … .1

USING R FOR CALCULATIONS

The second way to use R is ***as a modular system of data entry and analysis***. The important word here is modular. You can choose which data you want to work on (selecting a subset if necessary) and what analysis you want to perform, but more importantly you can save the results of your analyses (into "objects"), which then become available for further analysis, either immediately or later down the line. This is a very powerful approach as you can build your analyses progressively as a set of "Lego" pieces and assemble them in whichever order you see fit. This may sound a bit abstract, so let's follow a simple example.

```
> seq(1,10)
[1]  1  2  3  4  5  6  7  8  9 10
```

The function **seq** is used to create sequences of numbers, here the first 10 integers.

```
> x <- seq(1,10)
```

The vector (of numbers) returned by the **seq** function is assigned (with the combination of the "*smaller than*" and "*hyphen*" symbols without space!, *i.e.* **<-**) to a new object, called here ***x*** (I could have chosen any other name). On purpose, R doesn't return any output when an assignment is made. If you wanted to see what ***x*** contained you just need to call ***x*** itself after having created it:

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

Or you can enclose your assignment within brackets in order to display it while creating it.

```
> ( x <- seq(1,12) )
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Here note that a second assignment to the same object ***x*** will replace the earlier version.

After creating this object, you can process it further, for instance:

```
> sum(x)
[1] 78
```

Returns the sum of all elements of ***x***.

```
> nx <- length(x); nx
[1] 12
```

Returns the number of elements contained in ***x*** (*i.e.* the vector's length), and assigns the result to a new object named ***nx***. Note the call to ***nx*** (separated by "`;`") on the same line to display the value returned.

```
> mean(x); sd(x)
[1] 6.5
[1] 3.605551
```

The mean and standard deviation of ***x***.

```
> median(x)
[1] 6.5
```

Here the median is the same as the mean (see above), but it usually won't be.

```
> min(x); max(x)
[1] 1
[1] 12
```

```
> range(x)
[1]  1 12
```

Returns two values (as a single vector): the minimum and maximum values contained in ***x***.

```
> (log10x <- log10(x))
[1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
[8] 0.9030900 0.9542425 1.0000000 1.0413927 1.0791812
```

Returns the log10 of each element of ***x*** and assigns the result to a new object: ***log10x***.

```
> mean(log10x); sd(log10x)
[1] 0.7233614
[1] 0.3282556
```

```
> factorial(x)
[1]         1         2         6        24       120       720
[7]      5040     40320    362880   3628800  39916800 479001600
```

An important point to make here is that some functions work on the whole object and return a single value (*e.g.* **`sum`**, **`mean`**, **`sd`**), while others work on each element of the object one-by-one (*e.g.* **`factorial`**, **`sqrt`**). We are going to see later on that the behaviour of a function can also differ depending on the kind of object (vector, matrix, …) supplied to the function. So keep your wits with you. With a bit of practice you will soon know exactly what to expect from these functions.

At any point you can check which objects currently exist in your R session, using:

```
> objects()
[1] "log10x"     "nx"        "x"
```

`ls()` performs exactly the same action.

Box 1.2 Common Statistical Functions

`mean(x)`	the arithmetic mean of ***x***, $= \sum(x)/n$
`median(x)`	median value of ***x***, the one separating the 50% lowest values from the 50% highest values.
`var(x)`	the variance of ***x***, $= \sum(\bar{x}-x)^2/(n-1)$ [unbiased version, s^2] $= \left(\sum x^2 - \frac{(\sum x)^2}{n}\right)\Big/(n-1)$
`sd(x)`	the standard deviation of ***x***, $s = \sqrt{\mathrm{var}(x)}$
`min(x), max(x)`	the minimum (or maximum) value in ***x***
`range(x)`	the minimum and maximum values in ***x***
`rank(x)`	the rank (from smallest to largest) of each value in ***x***
`quantile(x)`	the minimum, first quartile, median, third quartile, and maximum in ***x*** (by default, but other quantiles can be obtained too).

1.4 The major kinds of object in R.

The most elementary kind of object that R can manipulate is a single value, formally known as a *scalar*, which can either be numerical (integer, real or complex), character (*e.g.* "hello") or logical (*i.e.* TRUE or FALSE). For instance:

```
> nx <- length(x); nx
[1] 12
```

the object ***nx*** holds a single value, here 12 (continuing our example from previous sections), representing the number of values held in the vector ***x***.

If you are unsure about the kind of value stored in your object, you can use **`class`**:

```
> class(nx)
[1] "integer"

> class("hello")
[1] "character"

> class(TRUE); class(FALSE)
[1] "logical"
[1] "logical"
```

Vector

A Vector is used to keep series of values of the same type. This is by far the most often used object in R (partly because it is the building block of other more sophisticated kinds of objects). Variables and sets of similar observations are typically stored in such vectors.

We have seen that vectors of regular sequences can be created using the function **`seq()`** (see also Section 1.5 Creating & Entering Data). More generally, a vector can be created with the function **`c()`** by concatenating a set of values.

```
> (y <- c(1, 0.4, 3, 100, 2, -1))
[1]   1.0   0.4   3.0 100.0   2.0  -1.0
```

Note here that all the figures have been automatically converted (*coerced*) to real numbers because a vector can only store values of the same type, and the presence of 0.4 in the series prevent the vector from being integers only. Actually, **`c()`** automatically converts any series of numbers (as long as they are not complex ones) into real numbers whether the series is only comprised of integers or not.

```
> class(y)
[1] "numeric"
```

More importantly if you mix characters, logical values and numbers in a vector using **`c()`**, all these will be automatically coerced to characters, probably not the behaviour that you wished for.

```
> (z <- c(TRUE, 10, -1, 0.4, "a"))
[1] "TRUE" "10"   "-1"   "0.4"  "a"

> class(z)
[1] "character"
```

After being created, you can manipulate vectors in a number of ways, using indexing (*i.e.* square brackets):

```
> y[2]
[1]  0.4
```

Extracts the second element of vector ***y***.

```
> y[-4]
[1]  1.0  0.4  3.0  2.0 -1.0
```

Extracts every elements of vector ***y*** except the 4th element.

```
> y[1:2]
[1]  1.0  0.4
```

Extracts the first and second elements of vector ***y***.

```
> y[c(1,3,5)]
[1] 1 3 2
```

Extracts the first, third and fifth elements of vector ***y*** through a vector of indices: c(1,3,5)

```
> y[-c(1,3,5)]
[1]   0.4 100.0  -1.0
```

Extracts every elements of vector ***y*** except the first, third and fifth elements.

```
> y[y>0]
[1]   1.0   0.4   3.0 100.0   2.0
```

Extracts all the positive values contained in vector ***y***. To understand how it works it may be useful to break down the operation in two steps:

```
> (y>0  ->  test)
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Each element of vector ***y*** is tested against the condition "bigger than 0", the individual results are then stored in a vector of logical values (which I only called ***test*** to be able to pass it on).

```
> y[test]
[1]   1.0   0.4   3.0 100.0   2.0
```

Passing this vector of logical values (either TRUE or FALSE) to another vector ***y*** (in-between square brackets) will then extract only the values of ***y*** for which the test evaluated as TRUE.

```
> y[y>0 & y<100]
[1] 1.0 0.4 3.0 2.0
```

A selection using a more complex logical test.

```
> rev(y)
[1]  -1.0   2.0 100.0   3.0   0.4   1.0
```

Displays the vector *y* in reverse order. Note however that the object ***y*** itself is unchanged. If you want this operation to permanently affect ***y*** you need to assign the result of this operation back to ***y*** itself. For instance: **`y <- rev(y)`**

```
> sort(y)
[1]  -1.0   0.4   1.0   2.0   3.0 100.0
```

Displays the vector ***y*** sorted in increasing order.

Matrix

A matrix is a rectangular array of values of the same type. It has a certain number of rows and columns and is typically created from a vector with the function **matrix**.

```
> maty <- matrix(seq(1,10), nrow=2, ncol=5); maty
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

By default, the matrix is filled column after column. If you want to fill the matrix row after row you can specify this with the option **byrow=TRUE**.

```
> matz <- matrix(seq(1,10), nrow=5, ncol=2, byrow=TRUE); matz
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

Beware though, if the vector you are using is not long enough to fill the entire matrix, the vector will be "recycled" to fill the remaining cells. Because this is not necessarily the behaviour that you may wish, a warning will be issued if the length of your vector is not a multiple of the number of rows and columns. For instance:

```
> matz <- matrix (seq(1,7), nrow=5, ncol=2, byrow=TRUE); matz
Warning message:
In matrix(seq(1, 7), nrow = 5, ncol = 2, byrow = TRUE) :
  data length [7] is not a sub-multiple or multiple of the number of rows [5]
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    1
[5,]    2    3
```

You can also create a matrix full of empty values and then fill it as you see fit.

```
> matx <- matrix(NA, nrow=2, ncol=2); matx
> matx[1,] <- c(3,8); matx
> matx[2,] <- c(2,1); matx
```

After creating your matrix, you can change the names of the columns and rows (if you find it useful). You could for example store the abundance (as integers) of a range of species across several sites (*i.e.* a community matrix) and give the names of your sites to the columns and the names of your species to the rows.

```
> colnames(matz) <- c("siteA","siteB")
> rownames(matz) <- c("sp1","sp2","sp3","sp4","sp5")
> matz
    siteA siteB
sp1     1     2
sp2     3     4
sp3     5     6
sp4     7     1
sp5     2     3
```

Just as for vectors, manipulating matrices is fairly easy. There are just a few additional quirks.

```
> matz[4,1]
[1] 7
```

Extracts the matrix cell from the 4th row and 1st column.

```
> matz[1, ]
siteA siteB
    1     2
```

Displays the entire first row by leaving blank the index about the column.

```
> matz[ ,1]
sp1 sp2 sp3 sp4 sp5
  1   3   5   7   2
```

Displays the entire first column by leaving blank the index about the row.

Make sure you include the `","` in the square bracket, or the result will surprise you. For instance:

```
> matz[7]
[1] 4
```

Which means that the 7th value in ***matz*** is the value 4. It only makes sense if you appreciate that matrices are "vectors" read by default columns by columns (it doesn't matter one bit that we filled it by rows). The first values are therefore 1,3,5,7,2 (first column) followed by 2,4,6,1,3 (second column). The seventh value in this sequence is indeed 4.

Instead of using numbers as index for your rows and columns, you can also use names (those you have given them). The result will be a vector of named values (just like when only numbers where used in your square brackets)

```
> matz[ ,"siteA"]
sp1 sp2 sp3 sp4 sp5
  1   3   5   7   2
```

```
> matz["sp1", ]
siteA siteB
    1     2
```

```
> t(matz)
      sp1 sp2 sp3 sp4 sp5
siteA   1   3   5   7   2
siteB   2   4   6   1   3
```

Transposes the matrix, *i.e.* converts the rows into columns and the columns into rows. Note that their names are properly transposed too.

```
> matz[matz>3]
[1] 5 7 4 6
```

Selects all the matrix elements that fulfil a particular condition (here being bigger than 3), but note that the result of this operation is returned as a vector (of unnamed values).

```
> sort(matz)
[1] 1 1 2 2 3 3 4 5 6 7
```

All matrix elements sorted in increasing order but as a single vector.

In summary, matrices are really vectors "disguised" as rectangular arrays, so be careful when you handle them. Some functions will treat them as matrices, other functions may treat them as single vectors. Just be aware of the distinction. With experience you will be able to anticipate the outcome.

Data frames

Data frames are rectangular structures too (with a set number of rows and columns) but unlike matrices different types of data can be mixed alongside each other. In other words, each column must be a vector of values belonging to the same type, but different columns can store different types of data. To a large extent data frames correspond to the concept of "spreadsheet".

An example will help to clarify things. Let's first construct three vectors of equal lengths.

```
> x <- seq(1,12)  # existed already, just repeated here for clarity
> y <- c("A","D","B","D","B","C","A","C","C","D","A","B")
> z <- x>5         # a vector of logical values
```

Then we can create a data frame with these three vectors with the function **`data.frame()`**

```
> datafr <- data.frame(x, z, y)
> datafr
    x     z y
1   1 FALSE A
2   2 FALSE D
3   3 FALSE B
4   4 FALSE D
5   5 FALSE B
6   6  TRUE C
7   7  TRUE A
8   8  TRUE C
9   9  TRUE C
10 10  TRUE D
11 11  TRUE A
12 12  TRUE B
```

The vectors ideally need to be of equal length. If they are not, the shorter vectors will be recycled until they reach the size of the longest vector. R will not always warn you about this (depends on whether the longest vector is an exact multiple of the shorter vectors) so be careful. Notice that the order in which you choose to enter these vectors as arguments matters, it ends up being the order of the columns of your data frame.

Note also that the values of the third columns appear without quotes even though they were originally created as character values. This is because the **`data.frame`** function automatically transforms vectors of character values into factors. We will look at factors in detail later, for now just note that factors are useful to record membership to particular groups of observations.

Notice also that the column names in your data frame are the names of the original vectors. But if you want to changes these names, it is very easy to do so:

```
> names(datafr) <- c("A","B","C")
```

The function **`names()`** returns the vector of column names from a data frame. With it you can query as well as set the names of a data frame columns. Here we used it to change all three names to "A", "B" and "C".

```
> names(datafr)
[1] "A" "B" "C"
```

```
> names(datafr)[1] <- "one2twelve"
> names(datafr)
[1] "one2twelve" "B"          "C"
```

Here we just changed the first column name (leaving the others unchanged) by indexing the vector returned by the **`names()`** function

As with matrices, we can **extract individual values**:

```
> datafr[1,1]      # or    datafr[1, "one2twelve"]
[1] 1
```

Extracts the 1st value from the 1st column.

```
> datafr[6,2]      # or    datafr[6, "test"]
[1] TRUE
```

Extracts the 6th value from the 2nd column.

```
> datafr[1,3]      # or    datafr[1, "Group"]
[1] A
Levels: A B C D
```

Extracts the 1st value from the 3rd column. Because this column is a factor, its levels (*i.e.* its possible values) are listed too.

You can also **extract an entire column**:

Using a column index in-between single square brackets:

```
> datafr[1]      # or    datafr["one2twelve"]
   one2twelve
1           1
2           2
3           3
4           4
5           5
6           6
7           7
8           8
9           9
10         10
11         11
12         12
```

Which will return a data frame object with a single column (the one you selected).

Using a column index in-between double square brackets:

```
> datafr[[1]]      # or    datafr[["one2twelve"]]
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Which will return a vector of values (without names).

Or using **$** after the name of your data frame to access a specific column by name.

```
> datafr$one2twelve
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Which also returns a vector of values (without names) and not a data frame object.

Extracting a particular row is also possible, but the result will be a data frame object since it would be impossible for a vector to hold data belonging to multiple types.

```
> datafr[1,]
  one2twelve  test Group
1          1 FALSE     A
```

Transposing a data frame is possible in theory using the function **t()**, but it is rarely worth it, as the new columns (the former rows) will be coerced to the least restrictive type of data. In our case, we had data of integer, logical and character type, so the new columns will all be coerced to character type, which is hardly the behaviour one may wish for. Transposing a data frame therefore only make sense when the data type is the same across the whole data frame (and if that is the case, a matrix may be more appropriate in the first place).

Sorting a data frame is certainly possible but not directly through the **sort()** function. Sorting a data frame instead must involve two steps. In the first step, a vector of indices is generated that would result in the desired sorting. In the second step, we provide this vector of indices in-between square brackets to our data frame to get the new sorted data frame. An example follows.

First, Let's create a new column of 12 random numbers (between 100 and 200) as a fourth column.

```
> datafr$rand <- runif(n= 12, min=100, max= 200)
```

These 12 numbers are uniformly distributed between the minimum and maximum provided. Note that trying to access a non-existing column by name (using **$**), results in this column being created alongside the existing columns. A convenient trick to add more columns to your data frame (just make sure that the length of your new vector is compatible with the existing ones or it will be "recycled").

```
> datafr
   one2twelve  test Group     rand
1           1 FALSE     A 110.7049
2           2 FALSE     D 145.6516
3           3 FALSE     B 170.1251
4           4 FALSE     D 150.3184
5           5 FALSE     B 107.8595
6           6  TRUE     C 162.6974
7           7  TRUE     A 180.3089
8           8  TRUE     C 183.3282
9           9  TRUE     C 120.7864
10         10  TRUE     D 100.9432
11         11  TRUE     A 169.6334
12         12  TRUE     B 112.7998
```

Of course performing this **runif** call on your computer, will produce different values for ***rand***.

The first step to sorting our data frame according to the values in the ***rand*** column, consists in deriving a vector of indices through the function **order()** which will result in the desired sorting.

```
> sorted <- order(datafr$rand)
```

The function **order()** works exclusively on vectors so we had to extract *rand* with **$**. We could also have used **datafr[[4]]** (see above about double square bracket indexing).

```
> sorted
[1] 10  5  1 12  9  2  4  6 11  3  7  8
```

The first number stored in this vector, here 10, then gives us the position (*i.e.* the index or the row number) of the lowest value in ***rand***. You can verify that the lowest value in ***rand*** is indeed the 10th value and equals 100.9432. The second number stored in the vector, here 5, is then the position of the second lowest value in ***rand***, and so on until we reach eventually the position of the maximum value in ***rand***, 183.3282 (at position 8).

We can then provide this vector in-between square brackets (matrix-like indexing) to sort our original data frame accordingly.

```
> datafr[sorted, ]
     # equivalent to datafr[ c(10,5,1,12,9,2,4,6,11,3,7,8), ]
   one2twelve  test Group     rand
10         10  TRUE     D 100.9432
5           5 FALSE     B 107.8595
1           1 FALSE     A 110.7049
12         12  TRUE     B 112.7998
9           9  TRUE     C 120.7864
2           2 FALSE     D 145.6516
4           4 FALSE     D 150.3184
6           6  TRUE     C 162.6974
11         11  TRUE     A 169.6334
3           3 FALSE     B 170.1251
7           7  TRUE     A 180.3089
8           8  TRUE     C 183.3282
```

Note how the row names of our sorted data frame (margin) match the values in our ***sorted*** vector.

However, if you want this new arrangement to be permanent (and not be lost after this call), you need to assign the result back to the data frame itself: i.e. `datafr <- datafr[sorted, ]`. This almost applies to everything in R. If you want something to be permanent or be able to re-use it later you need to assign it to an object (either an existing one or a new one depending on what you wish to do).

Conditional selection. It is also possible to extract observations from a data frame that fulfil a certain condition (just like for vectors and matrices).

```
> datafr[datafr$rand>150, ]
   one2twelve  test Group     rand
3           3 FALSE     B 170.1251
4           4 FALSE     D 150.3184
6           6  TRUE     C 162.6974
7           7  TRUE     A 180.3089
8           8  TRUE     C 183.3282
11         11  TRUE     A 169.6334
```

Displays only the observations that have a ***rand*** value larger than 150. Don't forget the comma in the square brackets. Leaving the space blank after the comma ensures that all columns are selected. If you only want to display some of the columns you can specify which ones using another vector of indices (for the columns this time).

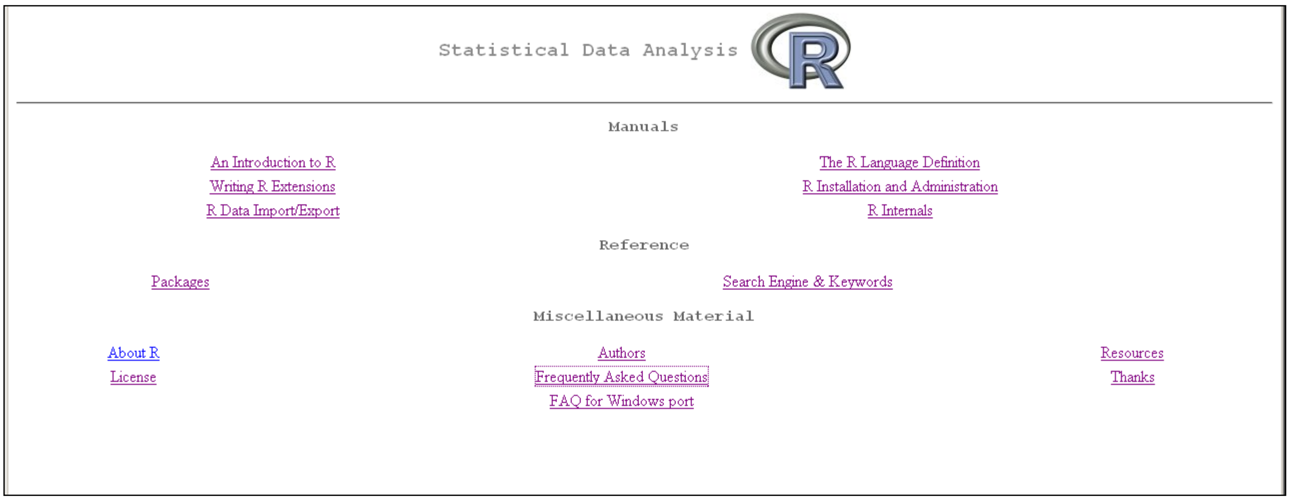
```
> datafr[datafr$rand>150, c(2,3)]
    test Group
3  FALSE     B
4  FALSE     D
6   TRUE     C
7   TRUE     A
8   TRUE     C
11  TRUE     A
```

Displays only those observations that have a ***rand*** value larger than 150, but only for columns 2 and 3. Note that selecting the columns by name *i.e.* `c("test","Group")` would be fine too.

1.5. Asking for Help (within R)

R comes with an impressive amount of documentation to help you make the most of its capabilities. You can access this help either from the menu or from the command line.

The central point of access is the HTML help page. You access it from the ***Help*** menu, under ***Html help***. You should then see the following webpage in the window of your web browser (figure 1.2).


Statistical Data Analysis

Manuals

An Introduction to R
Writing R Extensions
R Data Import/Export
The R Language Definition
R Installation and Administration
R Internals

Reference

Packages
Search Engine & Keywords

Miscellaneous Material

About R
License
Authors
Frequently Asked Questions
FAQ for Windows port
Resources
Thanks

Figure 1.2 The main help page for R.

On it, you can access the following documents and tools:

- **a series of 6 manuals** (in html format).

  An introduction to R will be the most directly useful to you. I encourage you to read it (in your spare time) to gain some background knowledge on how R works. The others may be skipped at this stage as they are rather more technical. Note that pdf versions (for printing) of these manuals can also be found in the ***Help*** menu under ***Manuals (in PDF)***.

- **Reference** tools.

  Search Engine & Keywords will open a new page (Figure 1.3) allowing you to search for keywords & functions (across all the packages that you have currently installed).

- **Miscellaneous Material**.

  In particular, two sets of *Frequently Asked Questions*, one to R in general and the other to R running under Windows. If you ever find yourself confronted by some strange behaviour of R, these FAQs might have the answer.

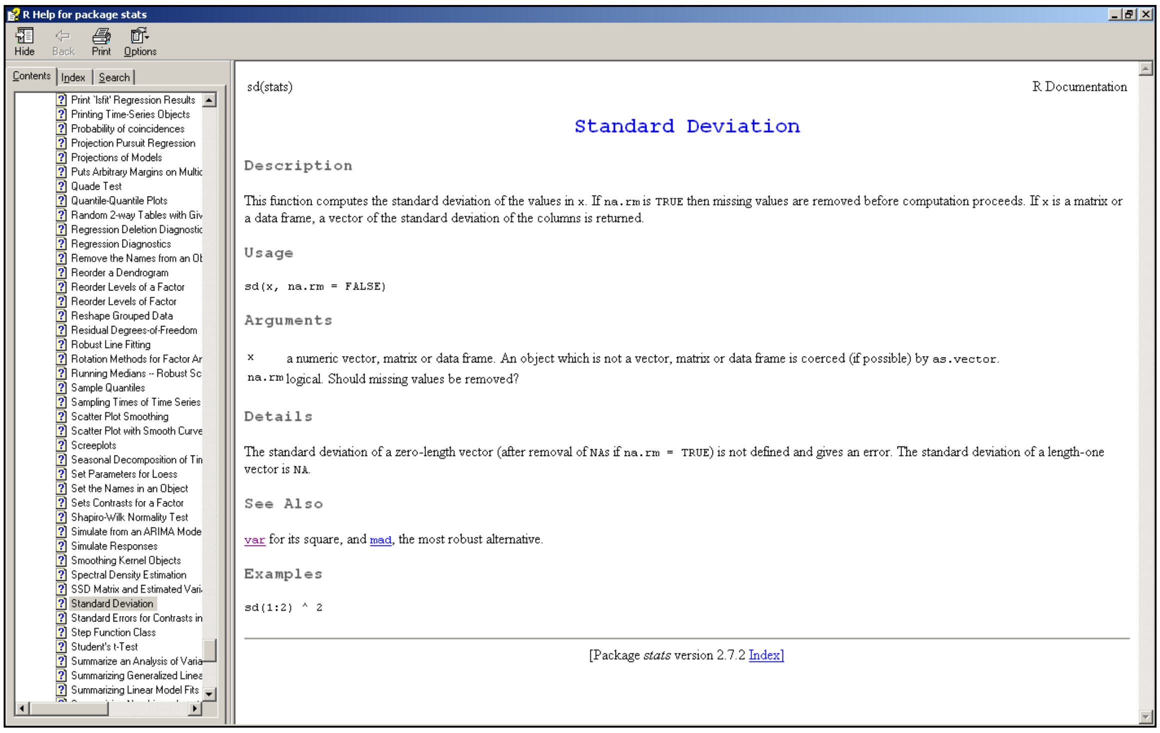
If all you need is a quick look at the help page of a particular function, then the best is to query the native help system at the command line in R

```
> help(sd)
```

Opens the help system (a sort of html "browser") and shows you the relevant page (Figure 1.4).

```
> ?sd
```

Performs exactly the same operation (but unlike `help` you can't pass on options to it, see below).


R Help for package stats
Hide
Back
Print
Options
Contents
Index
Search
Print 'lsfit' Regression Results
Printing Time-Series Objects
Probability of coincidences
Projection Pursuit Regression
Projections of Models
Puts Arbitrary Margins on Multic
Quade Test
Quantile-Quantile Plots
Random 2-way Tables with Giv
Regression Deletion Diagnostic
Regression Diagnostics
Remove the Names from an Ob
Reorder a Dendrogram
Reorder Levels of a Factor
Reorder Levels of Factor
Reshape Grouped Data
Residual Degrees-of-Freedom
Robust Line Fitting
Rotation Methods for Factor Ar
Running Medians -- Robust Sc
Sample Quantiles
Sampling Times of Time Series
Scatter Plot Smoothing
Scatter Plot with Smooth Curve
Screeplots
Seasonal Decomposition of Tin
Set Parameters for Loess
Set the Names in an Object
Sets Contrasts for a Factor
Shapiro-Wilk Normality Test
Simulate from an ARIMA Mode
Simulate Responses
Smoothing Kernel Objects
Spectral Density Estimation
SSD Matrix and Estimated Vari
Standard Deviation
Standard Errors for Contrasts in
Step Function Class
Student's t-Test
Summarize an Analysis of Varia
Summarizing Generalized Linea
Summarizing Linear Model Fits

sd(stats) R Documentation

Standard Deviation

Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds. If `x` is a matrix or a data frame, a vector of the standard deviation of the columns is returned.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

`x` a numeric vector, matrix or data frame. An object which is not a vector, matrix or data frame is coerced (if possible) by `as.vector`.

`na.rm` logical. Should missing values be removed?

Details

The standard deviation of a zero-length vector (after removal of `NA`s if `na.rm = TRUE`) is not defined and gives an error. The standard deviation of a length-one vector is `NA`.

See Also

`var` for its square, and `mad`, the most robust alternative.

Examples

```
sd(1:2) ^ 2
```

[Package *stats* version 2.7.2 Index]

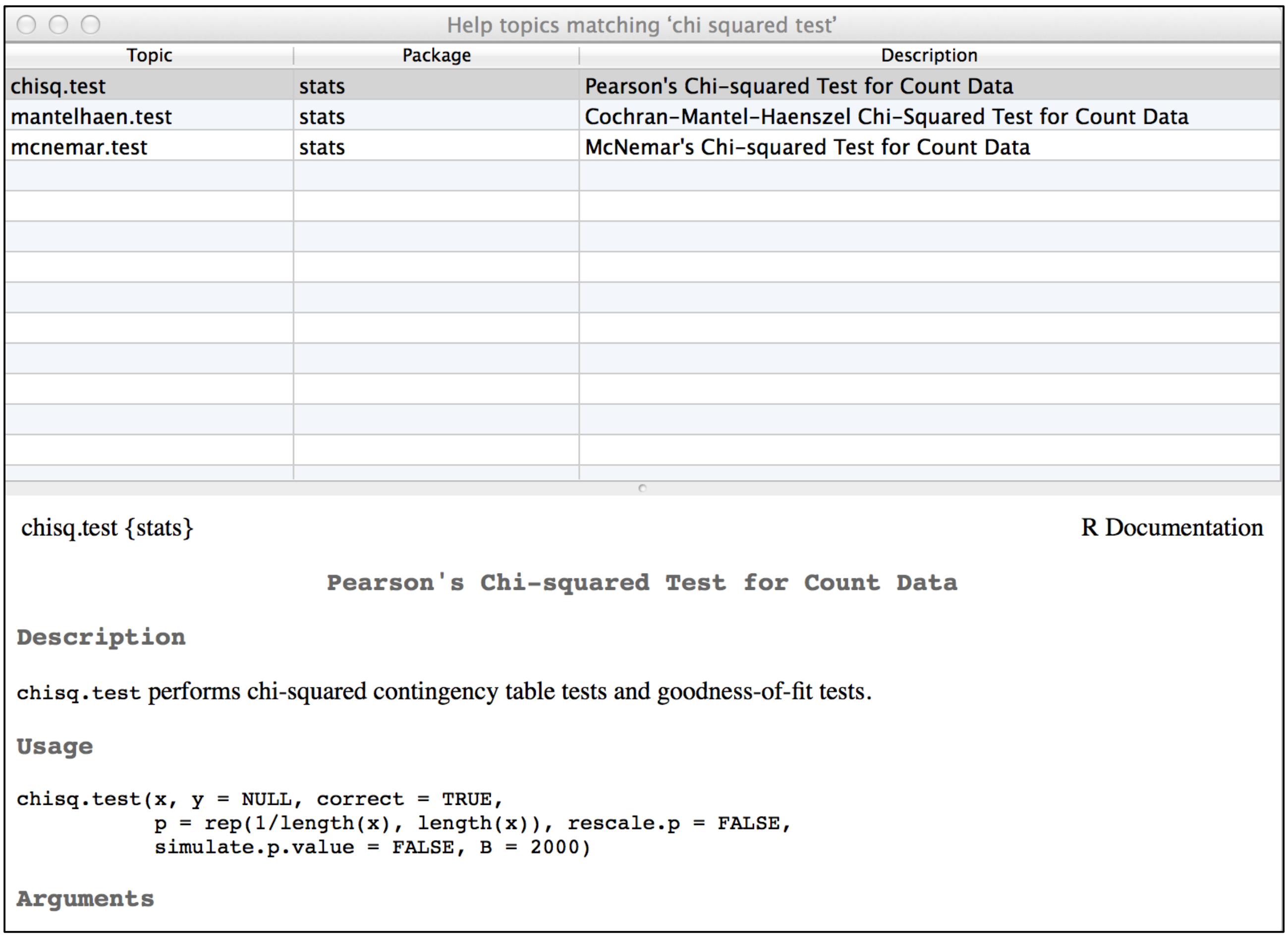
Figure 1.3 Example of a help page (here about **sd**) within R Help system.

Typically a help page is subdivided in several sections (Figure 1.4). At the very top in the left corner, is the function name and the package it comes from (in-between brackets). Next in large blue font is the title in plain English followed by a number of sections:

- Description explaining in plain English (well, as plain as possible) what the function actually does.
- Usage showing how the function should be typed in, with all of its options (though in practice many of these options can be left out). Options are followed by an equal sign and do not need to be specified unless you don't want to use the default setting. For instance, with the function **sd** the only option is **na.rm** and its default setting is **FALSE**, which will trigger an error message if the vector of values you are trying to use contains missing values (represented as **NA** in R). To avoid this error and obtain an estimate of the standard deviation using the non-missing values, you would need to specify the option as **na.rm=TRUE**.
- Arguments explaining the nature of each argument and option found in the function call.
- Details giving some information on how the calculations are carried out (which can be useful when you meet a problem).
- Value describing the nature of the object returned by the function call, especially when the object is not a single number or vector but a more complex object. Note that some function (*e.g.* **print**) don't return any value, they just perform actions.
- References listing a few key papers or books that are the basis of this function.
- See Also listing similar and related functions.
- Author(s) listing the authors of the function when it is not part of the base distribution (*i.e.* it comes from a third party package).
- Examples showing how these functions can be used in practice. You can actually run these examples in your **R console**, just typing **example(function)** replacing "function" by the name of the function you are interested in.

You may have noticed that this help function only allows you find help for a function for which you already know the correct name. However, in many cases, you won't to do something, like a chi-squared test, but you don't know what function will actually do that. In this case, there are several options. If you are not connected to the internet, the best option is probably the help.search() function. This function will search through all of the R libraries (or packages) that are on your computer and let you know which ones have documentation with the words in your search term. This produces a new window (Fig. 1.4), in which you can click on the individual functions and view the help page for each one.

```
> help.search("chi squared test")
```



Help topics matching 'chi squared test'

Topic	Package	Description
chisq.test	stats	Pearson's Chi-squared Test for Count Data
mantelhaen.test	stats	Cochran-Mantel-Haenszel Chi-Squared Test for Count Data
mcnemar.test	stats	McNemar's Chi-squared Test for Count Data

chisq.test {stats} R Documentation

Pearson's Chi-squared Test for Count Data

Description

chisq.test performs chi-squared contingency table tests and goodness-of-fit tests.

Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)), rescale.p = FALSE,
           simulate.p.value = FALSE, B = 2000)
```

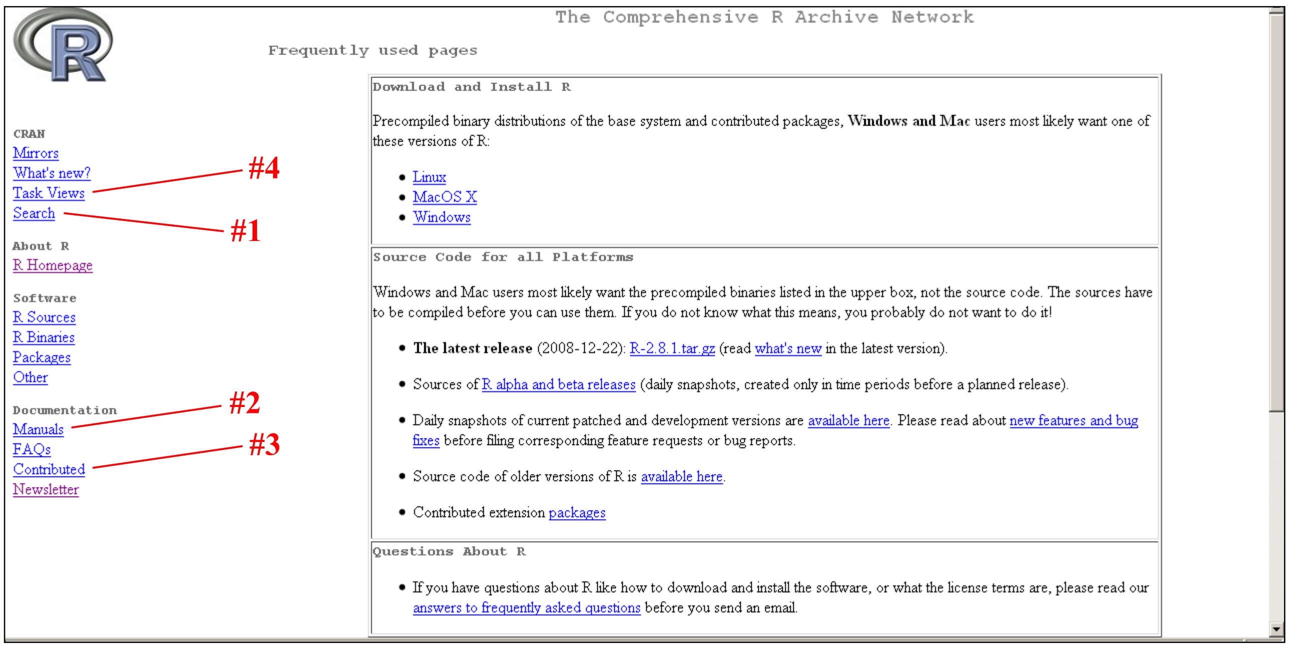
Arguments

Figure 1.4 Example of a help.search window (here with "chi squared test") within R Help system.

1.6 Sources of Information about R

When the help available within R is not enough, a vast amount of information on R can also be found on the web and in print (papers and books).

Your first point of call should be **CRAN** (**the Comprehensive R Archive Network**) located at http://cran.r-project.org/ (Figure 1.5), where you will find a number of useful resources:



The Comprehensive R Archive Network

Frequently used pages

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed
Newsletter

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Linux
- MacOS X
- Windows

Source Code for all Platforms

Windows and Mac users most likely want the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- **The latest release** (2008-12-22): R-2.8.1.tar.gz (read what's new in the latest version).
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

Figure 1.5 The CRAN homepage, the central source of information about R including the latest install version. The numbers refer to the resources listed below.

A selection of search engines dedicated to R (#1)

The very useful one is the **R Site Search** (**http://finzi.psych.upenn.edu/search.html**), allowing you to search help files, manuals, and mailing list archives found on the web (*i.e.* not just based on the packages you have installed on your computer). If your computer is connected to the web, you can actually use this engine from within R with the function `RSiteSearch("search term")`. You can also restrict your search to functions only and exclude the mailing list archives, with `RSiteSearch("search term", restrict="function")`. Or, see `help("RSiteSearch")` for a list of other available options.

My personal preference is the www.rseek.org page, which you can also readily add as a favourite on your toolbar. It searches through manuals, email archives, books, and other sources and provides a tabbed interface for accessing these. It also does a general search on a search engine (in this case google), and presents those results on the left-hand side of the page.

Official Manuals included with R (#2)

The 6 manuals covered in section 1.4 (Asking for Help) plus "**The R Reference Index**" a copy of all help pages of the R standard and recommended packages as a pdf (beware it is 2760 pages long and weighs 14MB).

Additional Documentation contributed by the R user community (#3)

Three categories of documentation is listed:

Long Documents with more than 100 pages (11 book-length documents).

Introductory Documents with less than 100 pages (16 tutorials)

Short Reference Cards (just summarizing the syntax of essential functions).

The list is actually too long to give here. For now, just remember that many of the recommended readings for further study can be found here. I particularly recommend "**R for Beginners**" by *Paradis* (only 76 pages) to start with.

Task Views (#4)

Task Views provide a quick summary of R capabilities broken down in 22 major statistical fields (e.g. *Envirometrics*, *Graphics*, *Cluster*). Each of them list for a particular statistical field the range of statistical techniques available in R and in which package they can be found. Don't expect to find detailed information on any technique in these task views. But if you know approximately what you would like to do with your data, the task views may give you an idea of where to start. Usefully, each package mentioned in the task views can be accessed through a web link. You can even (if you are mad enough) download and install all the packages mentioned in a particular task view from within R (see bottom of the Task View page for details on how to do just that).

Mailing lists

If you become a serious R user, you may find it useful to join eventually one of the dedicated e-mail lists, especially the *R-help* mailing list, where users from around the world can post their questions and get answers. You will vastly improve your understanding of R and collect many useful tips and tricks along the way. Beware though, this is a very active e-mail list with several dozen e-mails every day (you can opt for a single daily digest if you prefer) so only join the *R-help* list if you really get into R. Alternatively you can join one of the *Special Interest Groups* (SIG) which gets a smaller amount of traffic. The R-sig-ecology mailing list may be the most useful to you.

To join one of these mailing list, just follow the instructions at http://www.r-project.org/mail.html

1.7 Importing Data into R.

Entering manually large amounts of data directly within R is not usually recommended. It is cumbersome and prone to errors. Even using the **Data Editor** with **`fix`** and **`edit`** has its limitations. Most of the time, it will be more efficient and user-friendly to prepare your dataset in a spreadsheet application (let's say *Excel* for example) and then import your data in R. Fortunately, it is quite straightforward to do so, really. You just need to follow a few steps.

Saving your *Excel* spreadsheet as a csv.file

When preparing your *Excel* spreadsheet, it is worth knowing that only simple objects (vectors, matrices or data frames) can easily be imported in R. Also pay particular attention to the potential occurrence of missing values in your spreadsheet. If you have missing values, instead of leaving the entry blank, type in **NA** instead (the code given to missing values in R).

When your dataset is ready you will need to save it as a ***comma separated value*** file (*i.e.* **csv**) but as this format (text) strips off all your formatting and formulas, I recommend saving an **xlsx** version (the normal *Excel* format) beforehand for safe keeping (i.e. you will have two saved versions).

To save it as a **csv**, select ***Save As*** in the ***File*** Menu in *Excel* and select the **csv** format from the "*Save as type*" list at the bottom of the **Save As** window (depending on your version of *Excel*, the CSV option is about the 10th option down the list). Give your file a short and sweet name and save the **csv** file in a directory of your choice.

Setting your Working Directory in R

For R to be able to import your **csv** file, it needs to know where to find it. This can be done by specifying the whole path leading to your file (*e.g.* "M:/s0970908/PhD/R/example.csv"), but as this example suggests it can be quite cumbersome to type. An more practical solution would be to tell R which directory you are going to work from and where it can expect to find your files.

You can set your working directory with **`setwd()`**:

```
> setwd("M:/s0970908/PhD/R/files")
```

Note that the directories are separated by slashes here (whereas *Windows Explorer* contrary to all other operating systems use backslashes: \ ). Don't leave *Windows* backslashes in your path, or it will result in an error. This is because a backslash in R is a special character modifying the character that follows it. Alternatively you could use double backslashes (which basically means that the second backslash should really be treated as a backslash, ☹) as in this example:

```
> setwd("M:\\s0970908\\PhD\\R\\files")
```

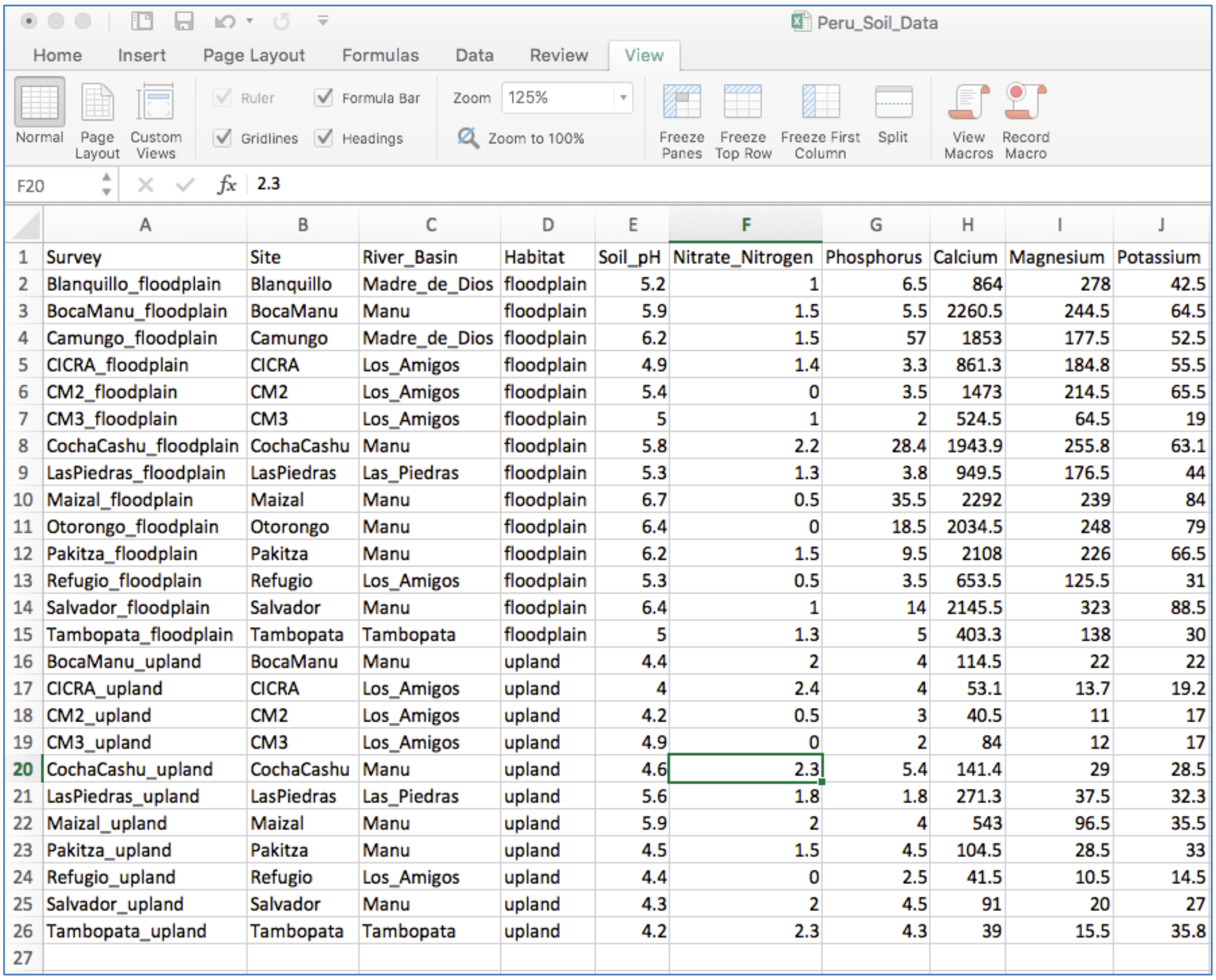
You might say it is still cumbersome to type, but at least you do the job only once per session, not every time you want to import a file. The added bonus of course is that if you keep a record of all your commands during a session (with an **.RHistory** file, see ***Save History*** in the ***File*** menu) you will automatically be able to trace back the location of all your files. A very useful thing when you come back to some analyses months or years later.

However, the easiest way to change your working directory is using the GUI interface outside of the R console. On macintosh machines, the working directory can be set under the 'Misc' menu tab. On Windows machines, the working directory is under the "Session" menu tab.

Reading your CSV file

Importing data is usually carried out with the function **`read.table`**. A few specialized functions also exist for more specific data formats but for now the **`read.table`** function will do just fine.

The spreadsheet ‘Peru_Soil_Data.csv’ has only 25 rows and 19 columns of data (Figure 1.8), with the top row representing variable names and the left-hand column giving sample names. The data come from 25 forest inventory plots in Amazon rain forest in Madre de Dios, Peru. These soils were analysed for various chemical and physical properties, of which 16 continuous variables are presented here. The data come from sites scattered across several catchments and the two major environment types in Madre de Dios, floodplain and upland forests, which are represented as categorical, or discrete, variables. Forests in floodplains receive nutrient-rich floods on a semi-annual basis whereas upland forests receive no fresh sediments and have been leached of many nutrients over the course of thousands of years of heavy rainfall. These and other patterns will become apparent as we visually explore the data.



	A	B	C	D	E	F	G	H	I	J
1	Survey	Site	River_Basin	Habitat	Soil_pH	Nitrate_Nitrogen	Phosphorus	Calcium	Magnesium	Potassium
2	Blanquillo_floodplain	Blanquillo	Madre_de_Dios	floodplain	5.2	1	6.5	864	278	42.5
3	BocaManu_floodplain	BocaManu	Manu	floodplain	5.9	1.5	5.5	2260.5	244.5	64.5
4	Camungo_floodplain	Camungo	Madre_de_Dios	floodplain	6.2	1.5	57	1853	177.5	52.5
5	CICRA_floodplain	CICRA	Los_Amigos	floodplain	4.9	1.4	3.3	861.3	184.8	55.5
6	CM2_floodplain	CM2	Los_Amigos	floodplain	5.4	0	3.5	1473	214.5	65.5
7	CM3_floodplain	CM3	Los_Amigos	floodplain	5	1	2	524.5	64.5	19
8	CochaCashu_floodplain	CochaCashu	Manu	floodplain	5.8	2.2	28.4	1943.9	255.8	63.1
9	LasPiedras_floodplain	LasPiedras	Las_Piedras	floodplain	5.3	1.3	3.8	949.5	176.5	44
10	Maizal_floodplain	Maizal	Manu	floodplain	6.7	0.5	35.5	2292	239	84
11	Otorongo_floodplain	Otorongo	Manu	floodplain	6.4	0	18.5	2034.5	248	79
12	Pakitza_floodplain	Pakitza	Manu	floodplain	6.2	1.5	9.5	2108	226	66.5
13	Refugio_floodplain	Refugio	Los_Amigos	floodplain	5.3	0.5	3.5	653.5	125.5	31
14	Salvador_floodplain	Salvador	Manu	floodplain	6.4	1	14	2145.5	323	88.5
15	Tambopata_floodplain	Tambopata	Tambopata	floodplain	5	1.3	5	403.3	138	30
16	BocaManu_upland	BocaManu	Manu	upland	4.4	2	4	114.5	22	22
17	CICRA_upland	CICRA	Los_Amigos	upland	4	2.4	4	53.1	13.7	19.2
18	CM2_upland	CM2	Los_Amigos	upland	4.2	0.5	3	40.5	11	17
19	CM3_upland	CM3	Los_Amigos	upland	4.9	0	2	84	12	17
20	CochaCashu_upland	CochaCashu	Manu	upland	4.6	2.3	5.4	141.4	29	28.5
21	LasPiedras_upland	LasPiedras	Las_Piedras	upland	5.6	1.8	1.8	271.3	37.5	32.3
22	Maizal_upland	Maizal	Manu	upland	5.9	2	4	543	96.5	35.5
23	Pakitza_upland	Pakitza	Manu	upland	4.5	1.5	4.5	104.5	28.5	33
24	Refugio_upland	Refugio	Los_Amigos	upland	4.4	0	2.5	41.5	10.5	14.5
25	Salvador_upland	Salvador	Manu	upland	4.3	2	4.5	91	20	27
26	Tambopata_upland	Tambopata	Tambopata	upland	4.2	2.3	4.3	39	15.5	35.8
27										

Figure 1.8 The ***Peru_Soil_data.xlsx*** spreadsheet before being converted to **csv** format for importing into R.

Now, save this spreadsheet as a **csv** file, keeping the original name: *i.e.* **Peru_Soiil_Data.csv**

The simplest call that you need to import this **csv** file into R is:

```
> soils<-read.table("Peru_Soil_data.csv",header=TRUE,row.names=1,sep=",")
```

here the data after being read is assigned to the object **soils**.

The first argument, **`"Peru_Soil_Data.csv"`** is the name of the file to read (in quotes). Just make sure that the spelling is spot on or R won’t find it.

The second argument, **`header=TRUE`** specifies that the first line of your file contains the names of the variables. By default, R would have assumed (**`header=FALSE`**).

The third argument, **`row.names=1`**, specifies that the first column contains the names for the rows.

The fourth argument, **`sep=","`**, specifies that the character separating the successive values on the same line is the comma. By default, R would have assumed (**`sep=""`**) that the separating character was "white space", which stands for one or more spaces, tabs, newlines or carriage returns.

As part of the **`read.table`** call several things would have happened.

(1) The object ***soils*** would have been automatically converted into a data frame.

```
> class(soils)
[1] "data.frame"
```

(2) The names of your variables in your **csv** file would have been passed on as the column names in the data frame ***soils***. Just a little note of warning, here. R doesn't like blank space in variable names (nor in an object name for that matter) so if you had included a blank space in one of your variable names, R would have automatically replaced it with a dot (`.`).

```
> names(soils)
[1] "Site"        " River_Basin"       " Habitat"         " Soil_pH"
[5] " Nitrate_Nitrogen"   " Phosphorus"      " Calcium"  " Magnesium"
[9] " Potassium"   " Sodium"      " Manganese"  " Copper"
[13] " Zinc"   " Boron"  " Cation_Exchange_Capacity"  " Total_Base_Saturation"
[17] " Sand"    " Silt"      " Clay"  "
```

(3) The numerical variables would have been stored as vectors of the appropriate type.

```
> class(soils$Total_Base_Saturation)
[1] "integer"            # i.e. it can take on whole number value
> class(soils$Soil_pH)
[1] "numeric"            # i.e. it can take on any numeric value
```

(4) The character variables however would have been automatically converted into factors whether this made sense or not.

If we didn't want R to automatically convert character variables into factors (leaving you the choice to do it yourself later when the need arises), the solution is to specify which variables should be left as is with the option **`as.is=c(1,3)`** (here variables 1 and 3) in **`read.table`**. This is a very flexible approach as you can then potentially allow some character variables to be converted into factors while others would be protected (with **`as.is`**).

Despite your best efforts, the steps described above could still fail. R will usually try to tell you what is wrong with your file, but I have found out that the most common reason for not being able to import a **csv** file is caused by *Excel* itself. Even though your spreadsheet looks perfectly rectangular, without any blank cells, *Excel* may sometimes include a few blank cells here and there just outside your main spreadsheet. This will invariable result in R throwing a wobbly (the number of variables would then appear unequal between rows). If you suspect this, my advice would be to open your spreadsheet in *Excel* again and to delete (not just clear, I really mean delete) half a dozen columns to the right and half a dozen rows below the "table" you want to import. Then resave it as **csv**.