Brief Introduction to R

(using R 3.0)

by Petr Šmilauer

Introduction

R is a non-commercial, free-to-use software for data exploration and statistical analysis. More precisely, it is a programming environment and its full and effective use requires at least rudimentary knowledge of the S programming language that represent the core of the R software.

This tutorial explains only the basics of the R program environment and S language use that are required to accomplish and understand the examples present in our book and to apply the same (and similar) functions to your own data. If you need to know more – and you will, if you decide to continue to work with R – I suggest that you start first with *An Introduction to R*, about 100 pages long and downloadable from the R primary resource site (*www.cran.r-project.org*). As of August 2013, the document URL was this:

`http://www.cran.r-project.org/doc/manuals/r-release/R-intro.pdf`

Extensive overview of many of the additional, freely available tutorials can be found here:

`http://www.cran.r-project.org/other-docs.html`

To progress even further, you might also like to check the book *W.N. Venables & B.D. Ripley (2002): Modern Applied Statistics with S. Fourth Edition. Springer*, which provides not only a detailed treatment of the R (and commercial S-Plus) software, but also of many useful statistical methods available in this program.

Installation

Because Canoco 5 is primarily a Windows software (and on other platforms it works only in the Windows environment emulators), I discuss here the R installation only on the Windows platform. There are, however, available installers for MacOS and various Linux flavours.

While you can install R from your own account even without administrator rights, I do **not** recommend doing so in the case a separate administrative account exists on your computer. In such case, the installer will likely fail in its attempt to create new R folder in the *C:/Program Files* folder. So if you have an administrator account on your computer, install R and also the additional packages (see below) from that account.

In the case there is just one account on your laptop or desktop computer[1], this account is likely blessed with administrative rights and you can safely proceed with the installation.

You need about 150 – 200 MB of free hard disc space to install and use R. You can download and start the installer from the following URL:

http://www.cran.r-project.org/bin/windows/base/

The link to the installer is at the page top (it reads “Download R 3.0.1 for Windows” at the time this tutorial is written), so you can click it, store the installer at your disc and ask for its execution. Your operating system is (rightfully) wary of executing programs downloaded from web, so before executing the installer, you will see a message like “The publisher of R-3.0.1-win.exe couldn’t be verified. Are you sure you want to run the program?” Of course, if you do not agree, your brief encounter with R is at its end ☺. Otherwise, you must then select the installation language, agree to displayed licence, and confirm installation folder, parts of the program to be installed and choose further customisation of program starting options. I suggest you stick with offered defaults in all these steps, essentially just pushing the *Next* button.

After you successfully installed R, you should start it (either using the “blue R” icon newly placed at your computer desktop or using the *Start / All Programs / R 3*.x /*R* 3.x menu command) to install additional packages. **Packages** are collections of specialised statistical functions and most of the R strength lies in them – very few, most essential packages are installed by default, but additional ones can be dynamically added and this is what you will do now to provide the functionality used in our book examples.

From the main program menu, choose the *Packages / Install package(s)* command and first select the server you want to use. Choose a server located in your (or nearby) country or – if this fails - use the “Austria” choice, as this is the master server at Vienna Technical University. After a while (R is connecting to chosen server and inquiring about available packages), a list of packages available for installation appears. This is usually a shocking moment for novice users – the number of available packages is huge and you might wonder whether there are really so many statistical methods in the world. Although they probably are, the primary reason is the democratic (or shall I say anarchistic?) approach to R software development. Anyway, your uneasy task is now to select just two package from the list, namely *multcomp* and *vegan*. To select both at the same time, make sure you press the *Ctrl* key on the keyboard, while selecting the second item with the mouse button. Click then the *OK* button and the packages are downloaded and installed.

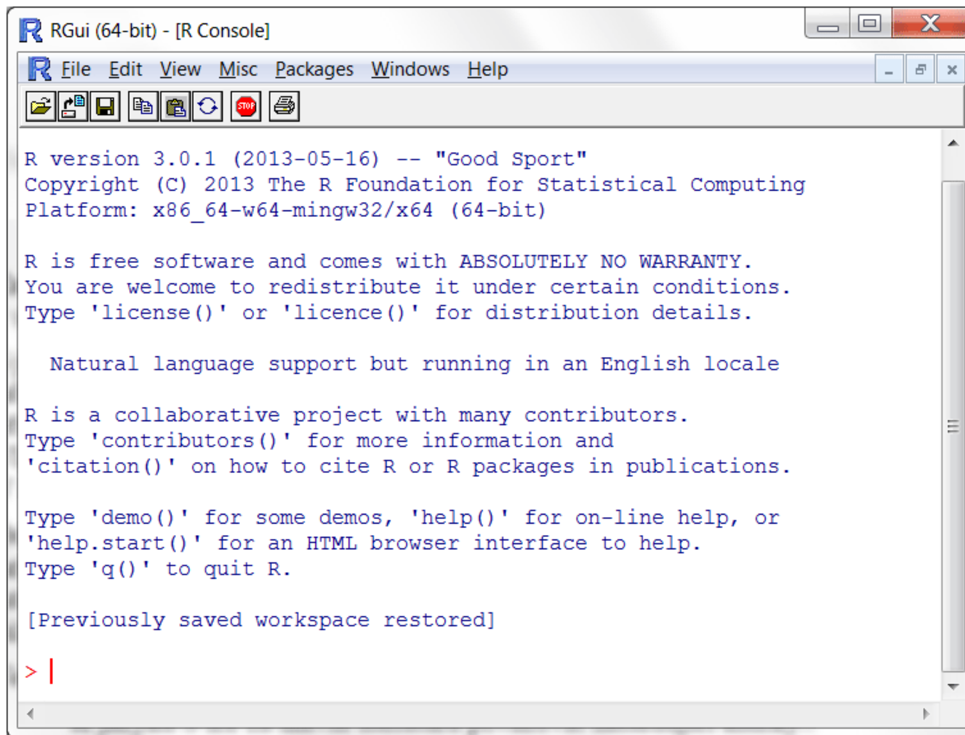
First Touches

I will not take you too far during your first session with R, but it is still a lot of new concepts to learn. Although you have presumably already practised the opening of R software (while installing additional packages, as described above), it is necessary to do it again ☺. The easiest start is by double-clicking the R icon from your desktop:

[1] This is how Windows are installed in most of its editions, except those aimed at the business environment.

If you do not find it there, you can start the R program from the *Start / All Programs / R 3.x* menu (see preceding section).

R displays its window that looks similarly to this one:



On the last line of the R Console window is a **>** character with a vertical text cursor. The **>** character represents so called *command prompt*. In this way, R is telling you that it waits for your commands. The S language (used by the R program) is a *functional* programming language with a relatively simple grammar. Like most of the software that was originally written for Unix operating system, it distinguishes lower- and upper-case letters, so that if you name one thing `A` and another one `a`, they will be considered different. Similarly, if you know that the name of function fitting linear regression models is `lm`, you cannot use it by typing `LM` or `Lm`.

Basically, your use of R consists of writing *commands* at the command prompt and exploring the text or graphical output that R creates in response. There are two types of commands in R – *expressions* and *assignments*. If you enter, as your command, an expression, it is evaluated (i.e. its value is calculated) and the result is shown in the window and immediately forgotten. As an example, type now following expression (you type only the bold red parts, the rest is shown to provide a context for you) and press the *Enter* key:

```
> 2+3
[1] 5
>
```

The meaning of the "[1]" preceding the correct answer (i.e. 5) might be slightly confusing. It is perhaps the right time to learn that R does not normally work with individual numbers, but with their groups called *vectors*.[2] The result 5 is taken simply as a vector of length one (with one entry) and if you ever create a longer vector (you will, soon), its output might not fit on a single line of the window. R therefore puts the index of a starting entry for a particular output line into square brackets, like it did here (all vectors start with index 1). If, say, 20 values fit on a single output line and there would be more than 20 values printed[3], the second line of output would start with `[21]`.

You will now create your own vector (albeit shorter) and also try the second form of command, the assignment. An assignment starts with the name of an object (which is called a **variable**), to which you assign a value, then an assignment operator (`<-`) and finally an expression. R evaluates the expression and if it succeeds (i.e. if you have not made a mistake in it), it takes its value and stores it in the variable. Here is your command to type:

```
> y <- c( 1, 4.5, 3.2, 2.8)
>
```

From now on, I will always omit the trailing line with the new prompt. You will note that the value assigned to `y` was not printed, but you will see soon how to do it. The assignment operator is formed by two characters: a "less than" character < and "minus" character –, written one next to the other. Otherwise, you can add as many space characters as you wish. In our example, the expression uses (*calls*) a *function*, this one named `c`. R has hundreds[4] of functions and you will learn about few of them in this tutorial. What all the functions share is how you use them: you type their name (remember, R is case-sensitive), then left round bracket, followed by a list of parameters (separated by comma character), and then right round bracket, closing the list. And then you press the *Enter* key. The function `c` combines (**c**oncatenates) the values of its parameters, forming a vector, and returns to you its value (which you store in the variable `y`).

To see what the function `c` has really done, simply type the variable name and press *Enter*:

```
> y
[1] 1.0 4.5 3.2 2.8
```

Here you go – all four *entries* of the vector y are printed. You can do more with them, however:

```
> mean(y)
[1] 2.875
```

Function `mean` calculated an average of the four values stored in `y` and printed its value. Suppose you want to calculate so-called variation coefficient for your four important numbers. You can do so using following commands:

```
> m <- mean( y); v <- var( y)
> m / sqrt( v)
[1] 1.989076
```

---

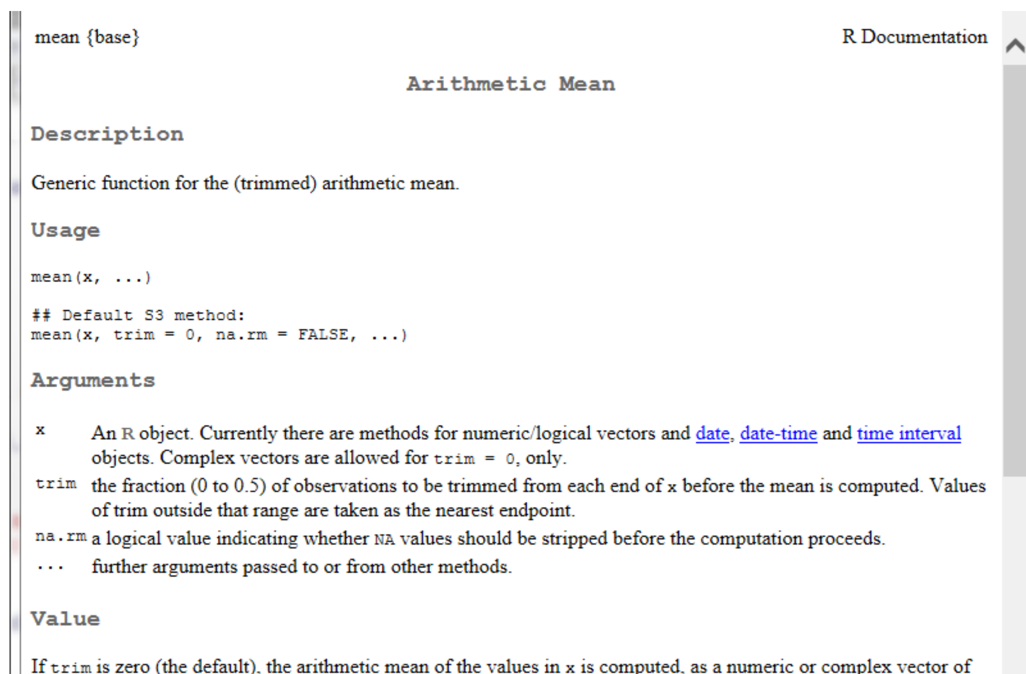
[2] More precisely, vector is a group of ordered values, all of the same type.
[3] I will use here the word *printed* in the sense of "displayed by R in the console window".
[4] Or thousands, if you count those present in packages.

The first line contains actually two commands (two assignments), defining two new variables, m and v (representing mean and variance of the y values). The commands are separated by a semicolon character. The expression in the second line divides the mean by the variance, but the variance is first square-rooted (using function sqrt), so R is actually dividing by standard deviation, as it should when calculating the variation coefficient. The result of that expression is then printed.

While the above example gives you an impression that functions mean, var, and sqrt take one parameter each, it is not quite so. In fact, they (and most other functions) have multiple parameters, but some of them are facultative: they already have some **default** (implicit) **values**, which are used when you do not specify the parameter in the function call. You can learn about all the parameters the function mean accepts by looking at its documentation. To display it, you use (no surprise here, I hope) another function, called help:

```
> help(mean)
starting httpd help server ... done
```

Following page (only a part is shown) is then displayed in your web browser:


mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of

After a brief summary and the description of use, individual function parameters (arguments) are listed, with their names. The names are important when you need to change the value of just one particular parameter. So, if you want to change, say, how the missing (NA) values are handled by mean, you can do so by calling it like **mean(y, na.rm = TRUE)**. You will also note (for the first parameter of mean) that the identity of unnamed parameters is deduced from their relative position in the function call. Unnamed parameters must therefore appear only before the first named parameter in a function call. Before you close the help window, scroll to its bottom. You will see there an *Examples* section that shows you how to use the function. While this is probably redundant information for such a simple function, you will find very useful to check the *Examples* section for more advanced functions, e.g. those importing data or calculating cluster analysis. You can even have these example commands executed, using function example, like **example(mean)**.

Two more important points before you close this simple, but tiresome first session. Type the following command:

```
> y <- 1+1
```

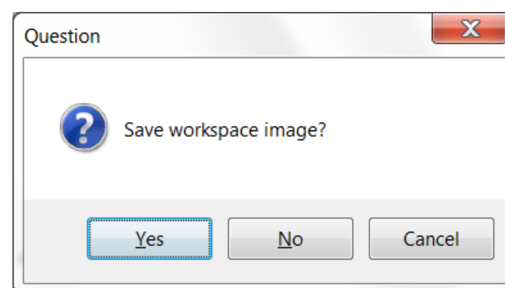
What has happened? I am sure you know the answer: the variable y has now value 2. But what has happened to your vector of four numbers? Well, it is gone ☹. You have asked to replace it with the result of 1+1 expression and R did it, as you wished. This is all right, but imagine that y did not contain four numbers, but say 1357 observations on 20 variables you have been collecting in the past two years from your experiment. Why, oh why, you did not backup your data elsewhere?

This is, in fact, another Unix legacy, where attitude towards software users is different from the one adopted by Windows software: here you are the **boss**, and when you ask for something, you know what you are doing, aren't you? You are not an idiot that needs to be constantly bothered with popup dialogs saying "Existing data can be deleted. Do you really want to do what you asked to do? [Yes] [No] [Cancel]".

Next, type the following command:

```
> q()
```

Now, you get a dialog box, after all:



But what the question actually means? First, I must tell you what function q does. Well, it quits (closes) R program. As in the other Windows programs, you can use the *File | Exit* menu command, but to be stylish (everything in R can be done with a function) or a snob, you can close R with the q function. You should also note that the call to q does not contain any parameter; even so, you must type the empty parenthesis after the function name, if you want to call it.

Now back to the **workspace**. It is a virtual storage place where the variables you have created during your session with R (or during previous sessions, indeed) are stored. The variables y, m and v are stored there and if you want to use them again the next time you open R, you must - at the end of your session[5] - store it to a file on disc – and this is what the above dialog is all about. To learn a little bit more about the workspace, abandon the program exit for a moment by clicking the *Cancel* button, and type the following command:

```
> ls()
  [1] "m"    "v"     "y"
```

[5] Or earlier, using the *File / Save workspace* command.

This shows the variables present in your workspace (if this is not your first session with R, you will probably see many more variables listed). If you want to remove variables that you no longer need, you can do so using function `rm` (from **rem**ove):

```
> rm(m,v)
> ls()
  [1] "y"
```

Again, no question is asked! You have requested `m` and `v` removal and they are gone now.

Finally, you can close the R program in peace (I hope).

Data Types and Data Import

So far the only type of R data you have seen was a vector. In fact, this is not a type at all, the real type is a *numeric* vector, or a vector of *logical values* (*TRUE*, *FALSE*), or a vector of *character strings* (e.g. labels). **Vector** is rather a particular type of *data structure*, in which the values of these basic types can be stored. Vector requires all of its entries to be of the same type. There are many other types of data structures, but I will mention here just two: *list* and *data frame*. A **list** is a hodge-podge of various data structures, even with various data types. You might have a list containing a vector of numbers, another vector of numbers with different length, a vector of character strings, or even another list. You will be rarely creating lists yourself, but they are the most usual type of values returned from more complex statistical functions, such as those computing ANOVA (`aov`) or linear regression (`lm`). List nature allows these functions to put together any kind of information representing things like function results or requested properties of the statistical model.

Data frame, on the other hand, is more regular in its contents. For a start, you can imagine that a data frame is simply one, two or many vectors, each of the same length (i.e. with the same number of entries), but not necessarily with the same data type. If you imagine these vectors as columns, you can see the data frame as a regular 2D-table, where each row represents one observation and each column represents one variable (in statistical sense, because whole data frame is one variable in R). Beside numeric and character vectors (columns), you will sometimes put *factors* into data frames. They are a special type of character vectors, with predefined set of possible values (factor levels); they are used in ANOVA models and elsewhere. Each data frame column has a name (rows can also have names), like the variables in Canoco 5 data tables have. If you have a data frame called, say, `exp.data` with columns `treatment` and `abundance`, you can refer to individual columns using frame name and column name, connected with dollar character: `exp.data$treatment`. In addition, most statistical functions allow to move-out the data-frame name to a parameter called *data*, so that you can then directly refer to individual columns. Here is an example call performing one-way ANOVA:

```
aov(abundance~treatment, data=exp.data)
```

Obviously, data frame is an ideal type of structure for submitting your data to statistical procedures (functions). Typically, you will start from an Excel spreadsheet table, import it to R as a data frame, and pass it to chosen statistical function. Here I demonstrate one way of importing data into R – copying them to Windows Clipboard and fetching them from there, using `read.delim` function.

You will be importing a single data table from an Excel file, providing data for Case Study 4 (Chapter 15) and named *Seedl.xlsx*. Open the file in Excel (or your favourite spreadsheet software) and select the *seedldesign* worksheet.

	treatment	block	seedlsum
rel1	Cont	1	95
rel2	Litter	1	91
rel3	Nardus	1	64
rel4	Li+Mo	1	107
rel5	Cont	2	88
rel6	Litter	2	70
rel7	Nardus	2	51
rel8	Li+Mo	2	180
rel9	Cont	3	44
rel10	Litter	3	57
rel11	Nardus	3	55
rel12	Li+Mo	3	173
rel13	Cont	4	94
rel14	Litter	4	99
rel15	Nardus	4	53
rel16	Li+Mo	4	80

Select the area enclosing whole data table, including row and column labels, and copy it to the Clipboard (e.g. using *Ctrl-C* combination of keys). You will note that the first variable (*treatment*) contains level names and is clearly a factor, while the following two variables look like numeric data. The *block* is, however, also a factor, only it does not look so in the spreadsheet.

Switch to R software and enter following command:

```
> seedlings <- read.delim( "clipboard", row.names=1)
```

By calling the `read.delim` function, you ask for importing a TAB-delimited text "file" from the Clipboard (instead of the special "clipboard" word, you can specify path and name of a real text file) and placing it into variable `seedlings`, which will be a data frame. The `row.names` parameter specifies that the first imported column does not belong to data, but rather contains labels for individual observations (rows). The numbers present in imported Excel worksheet do not contain fractional parts, but most data sets do and there you must take care if you happen to live in a country where the dot character is **not** used as a decimal separator. In multiple European countries, comma character is used as a decimal separator and in such a case, it is simplest to use the `read.delim2` function instead. You will now check whether the import succeeded. Like for any other R variables, after you type the data frame name and press *Enter*, content of the variable (here data frame) is printed:

```
> seedlings
      treatment block seedlsum
rel1       Cont     1       95
rel2     Litter     1       91
rel3     Nardus     1       64
rel4      Li+Mo     1      107
...
rel15    Nardus     4       53
rel16     Li+Mo     4       80
```

Note, however, that this command would work not so nicely for a large data set. Anyway, the content seems correct, doesn't it? For this and other imported data frames, I recommend that you always use the `summary` function:

```
> summary(seedlings)
  treatment      block           seedlsum
 Cont   :4   Min.   :1.00   Min.   : 44.00
 Li+Mo  :4   1st Qu.:1.75   1st Qu.: 56.50
 Litter:4   Median :2.50   Median : 84.00
 Nardus:4   Mean   :2.50   Mean   : 87.56
             3rd Qu.:3.25   3rd Qu.: 96.00
             Max.   :4.00   Max.   :180.00
```

When you pass to `summary` function a data frame, it summarises separately each of its columns. For factors (like `treatment`) it shows its (first few) levels and the number of times they occur in the data. For numerical columns (like `seedlsum`, representing total count of plant seedlings in each plot), it shows several statistical summaries: data range, lower and upper quartiles, median and mean. But `block` is not a numeric variable, as I have already mentioned. The `read.delim` function simply cannot work-out it should be a factor, given it has numeric values. So you must turn it into a factor explicitly. The easiest way to do so is as follows:

```
> seedlings$block <- as.factor(seedlings$block)
```

When you do such an in-place change of data frame values, however, you should double-check you have specified the names of data frame and column correctly. The R is unforgiving (as I have already demonstrated): if you misspell column name at the left side, a new column with this name will be created; if you misspell column name at the right side, you will probably lose the original data in `block`. If you feel unsecure, make a copy of the *seedlings* data frame first (using <– operator), modify it and verify the changes before copying new data back. In any case, always re-check whether the data were changed correctly:

```
> summary(seedlings)
  treatment  block     seedlsum
 Cont   :4   1:4   Min.   : 44.00
 Li+Mo  :4   2:4   1st Qu.: 56.50
 Litter:4   3:4   Median : 84.00
 Nardus:4   4:4   Mean   : 87.56
                   3rd Qu.: 96.00
                   Max.   :180.00
```

Statistical Analyses

Here I will only briefly discuss how to use packages, as well as the functions implementing statistical models. Some of these functions (such as for linear models, ANOVA, or generalized linear models) are readily available and you do not need to specify any specific package for them, while others are inaccessible until you open (and perhaps install first) their package.

Your task will be to fit and test a statistical model that compares the average number of seedlings among four different experimental treatments, from an experiment described in Chapter 15. Here you will abandon the classical linear model and use a generalized linear model (GLM) with the number of seedlings fitted as a response variable with assumed Poisson distribution. Generalized linear models can be fitted in R using function `glm`. As you will need to work further with the fitted model, you will store the value returned by `glm` in a variable. There are no strict rules how to name such variables, but you should develop some strategy to keep your R workspace tidy. Here, foreseeing I will not fit so many models for this dataset, I suggest to keep things simple:

```
> glm.seedl<-glm(seedlsum~block+treatment,data=seedlings,family=poisson)
```

The first parameter of function `glm` is so-called **model formula**. For this and similar models, it starts with the name of response variable, followed by a tilde character (**~**) separating the list of predictors, and then the specification of model predictors. Here I use the other two variables of the data frame and specify the frame name using `data` parameter. Although not appropriate for this example (with no treatment replicates within each block), in other data sets you often need to specify, beside main effects, an interaction between two factors (say `A` and `B`). This can be done in the model formula either as `Y~A+B+A:B` or, more simply, as `Y~A*B`. If the effect of factor `B` is nested in factor `A`, this can be described in the model formula as `Y~A/B`.

The `family` parameter specifies the assumed distribution for the stochastic component of the model and it can be also used to specify the link function, if not using the default one.

Now you need to obtain model summary. For this, type the following command:

```
> summary(glm.seedl)
Call:
glm(formula = seedlsum ~ block + treatment, family = poisson,
    data = seedlings)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-4.3665  -2.0216   0.2572   1.6040   3.8844

Coefficients:
                Estimate Std. Error z value Pr(>|z|)
(Intercept)      4.40424    0.07213  61.060  < 2e-16 ***
block2           0.08584    0.07329   1.171    0.242
block3          -0.08168    0.07642  -1.069    0.285
block4          -0.09084    0.07661  -1.186    0.236
treatmentLi+Mo   0.52013    0.07048   7.380 1.58e-13 ***
treatmentLitter -0.01254    0.07918  -0.158    0.874
```

```
treatmentNardus -0.36427    0.08718   -4.179 2.93e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 244.02  on 15  degrees of freedom
Residual deviance:  89.75  on  9  degrees of freedom
AIC: 203.42

Number of Fisher Scoring iterations: 4
```

It looks as if you have used the same function `summary` as before (for the `seedl` data frame), yet with a completely different outcome. But this is not so – `summary` is a **polymorphic function** that is specialised for different types of data specified as its first parameter. For a data frame, a function called `summary.data.frame` is called, but here the `summary.glm` function is used. Similar versatility can be found also with other functions, most notable is the function `plot`.

You have started with the function `summary`, although it does not provide the most concise summarisation of the effect of the two predictors. But it helps to detect so-called overdispersion, which is indeed present in our model: comparing the size of *Residual deviance* (89.75) with the number of *Residual ... degrees of freedom* (9) suggests that the residual variation is much larger than one would expect for a Poisson distribution.[6] So you will refit the model, specifying somewhat different distribution family, called *quasi-Poisson*:

```
> glm.seedl2 <- update( glm.seedl, family=quasipoisson)
```

This command illustrates another commonly used pattern of work with statistical models in R. Instead of specifying a new call to `glm`, which would - beyond modifying the `family` parameter - repeat all the function parameters, you request here an update of an existing model, specifying only the changed parameter(s). The `update` function is frequently used to gradually develop a model through its formula (i.e. the first parameter of the function `glm`). Imagine you would have another explanatory variable - say `moisture` - in the `seedl` data frame and you would like to add it to the model. To do so, you call the `update` function in the following way: … **`update(glm.seedl2, .~. + moisture)`**. Because you change the first parameter of `glm`, you do not need to name the parameter (but you can alternatively use its name, i.e. `formula=.~.+moisture`). The dot at the left / right side of the tilde character means "response / predictor variable(s) as in the original model", while the `+ moisture` part represents the addition of a new predictor. Similarly, predictors can be taken out of an earlier model by preceding them with the – character.

Next you will perform correct test[7] for each of the two used parameters (output simplified):

---

[6] The overdispersion is briefly discussed in Section 8.3 of our book, but a detailed discussion of the dispersion parameter and how to detect an overdispersion is beyond the scope of both our book and of this tutorial. I simply do not want to present an incorrectly performed analysis, so I refer you here to more advanced textbooks about the use of generalized linear models.

[7] For GLMs with Poisson distribution, χ^2 test statistic is normally used, but in the case of overdispersion, the F statistic is more appropriate.

```
> anova( glm.seedl2, test="F")
Analysis of Deviance Table
Model: quasipoisson, link: log
Response: seedlsum

Terms added sequentially (first to last)
          Df Deviance Resid. Df Resid. Dev      F  Pr(>F)
NULL                         15     244.03
block      3    7.302        12     236.72 0.2514 0.85843
treatment  3  146.973         9      89.75 5.0601 0.02524 *
```

As with the ANOVA model presented in Chapter 8, the effect of *block* is not significant, while *treatment* has a significant (and here somewhat stronger) effect ($F_{3,9}$=5.06, p=0.0252).

Finally, you will reproduce the multiple-comparison tests for our fitted GLM and *treatment* factor. The functions you will use (`glht`, as well as a specialised version of `summary`) are stored in a separate package called *multcomp*. You must first open it with a call to `library` function. When you stop using a package, it is recommended to close it using function `detach` (the call used below closes the last opened package).

```
> summary(glht(glm.seedl2,linfct=mcp(treatment="Tukey")))
         Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: Tukey Contrasts

Fit: glm(formula = seedlsum ~ block + treatment,
        family = quasipoisson, data = seedl)

Linear Hypotheses:
                     Estimate Std. Error z value Pr(>|z|)
Li+Mo - Cont == 0     0.52013    0.21930   2.372  0.08141 .
Litter - Cont == 0   -0.01254    0.24638  -0.051  0.99995
Nardus - Cont == 0   -0.36427    0.27125  -1.343  0.53263
Litter - Li+Mo == 0  -0.53267    0.22016  -2.419  0.07237 .
Nardus - Li+Mo == 0  -0.88440    0.24768  -3.571  0.00196 **
Nardus - Litter == 0 -0.35173    0.27195  -1.293  0.56437
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
(Adjusted p values reported -- single-step method)
> detach()
```

Note another common pattern in the function call: as you will use the value, returned by function `glht`, just once, you do not store it in a variable, but rather you nest the call to `glht` within a call to `summary`, so that the latter function receives it as its first parameter.

So, this is the end of my brief introduction to R. I wish you lot of patience while learning it and I hope your statistical analyses will be empowered by the choice of this system.