BIO311: Population Ecology
Practical 6: An introduction to R

Timothée Bonnet &* Koen van Benthem

timothee.bonnet@ieu.uzh.ch
koen.vanbenthem@ieu.uzh.ch

Spring 2016

Contents

We do not claim to teach you the most efficient way to use R. If you at some point during the computer practicals encounter a code that you could make more efficient or elegant, please do let us know! We do want to learn from you as well.

However, do try to understand exactly what we are doing and how the functions we use work. The best way to learn how functions work is by either using the R-manaul (type `?functionname`) or by creating dummy data (just make up a small amount of data yourself) and analyse what the function does to this data.

*This document was co-authored by Tina Cornioley

1 Getting to know R

Before exploring the data that you collected in the lab, we recommend you to get familiar with R. Please go over the following tutorial up to and including section 6 to learn or refresh the basics of R:

http://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf

Note: in this tutorial the word vector is used for all lists of numbers and the word matrix is used for all arrays of numbers, not only for transformations.

In this document you will find both normal text as well as R-code. Copying code from this pdf into R may results in errors. This has to do with the encoding of the file. Instead we recommend to retype the commands.

2 Dataset preparation

Prepare you dataset in excel before importing it into R. Open your rotifer data in excel and create a copy. Never directly modify your original data file. There is always the chance that you make a serious mistake which results in losing all or part of your data. Your file must not contain any space (especially check the column names, they should have no space between words) or any special character of the type "é" or "ö". Remove the comments, they are not in a form that can be analysed numerically but keep them in mind when writing your report. Avoid empty cells by filling them with NA if you do not know the value for that cell or zero if appropriate.
Once you have checked your document, save it as a .csv file delimited with ",". Carefully select the folder you save your document into. You will have to set this folder as your working directory later. If you are new to R, the easiest is that you save it in a folder name "Pop_Ecol" on your desktop.

3 Getting data into R

3.1 Step 1: Set the working directory

This is the folder where R will search for files and where it will save your data afterwards. You must insert your own working directory path.
Making a mistake when typing the directory path from memory is very likely. Instead, you can right-click your file and select "Properties". From here, you can copy and paste the full directory structure (and the file name).

```
setwd("C:/Desktop/Pop_Ecol")
```

3.2 Step 2: Load your dataset

The next step is loading your dataset into `R`. The line of code below shows how to read your dataset if it is saved as `'rotifer_data.csv'` file seperated with ",". The dataset must be in the working directory you specified above otherwise this command won't work.

```
rot <- read.csv("rotifer_data.csv", sep=",", header=T)
```

3.3 Step 3: Check your data

3.3.1 Initial overview

The next functions allow you to take a first look at the data you have loaded. If you see anything strange at this stage, you may have to go back to your original document in excel.

```
head(rot) # Output the first few rows of rot
tail(rot) # Output the last few rows of rot
str(rot)  # Describe the structure of rot
summary(rot) # Calculate basic information for each column in rot
```

What happens if you type `head(rot,10)` instead of `head(rot)`? Try the same thing for the function `tail`.

3.3.2 Finding typos

While typing in the data, a typo is easily made, especially for the columns `Copper` and `Population`. For the rest of the analysis it is crucial that the spelling in these columns is consistent. For this, we focus on these specific columns. To access one specific column, we use the notation `dataframe$columnname`. Thus, to extract all the values of the `Population` column, you can use `rot$Population`. Now use the command `table()` to obtain a summary of this data:

```
table(rot$Population)
```

You will notice that the post pollution populations are indicated with either `Postpollution` or `Post-pollution`. This will be very inconvenient in the further analysis. Correct this as follows:

```
rot$Population[rot$Population=="Post-pollution"] <- "Postpollution"
```

What happens here is that we first select the column `Population` by typing `rot$Population`. Next, the square brackets indicate that we are only interested in a part of the contents of that column. The part that we are interested in, are the entries that have the value `"Post-pollution"`. Precisely these elements

are selected by `rot$Population[rot$Population=="Post-pollution"]`. Subsequently these elements are replaced by the properly spelled version by the command `<-"Postpollution"`. Now we check whether the problem is solved:

```
table(rot$Population)

##
##    Commercial     Pollution Postpollution      Recovery
##            54            54            54            54
```

Repeat these steps now for the `Copper` column, and correct any typos you encounter in that column.

> *Please note that we have included these errors manually, just for practicing purposes.*

3.3.3 Average values per type

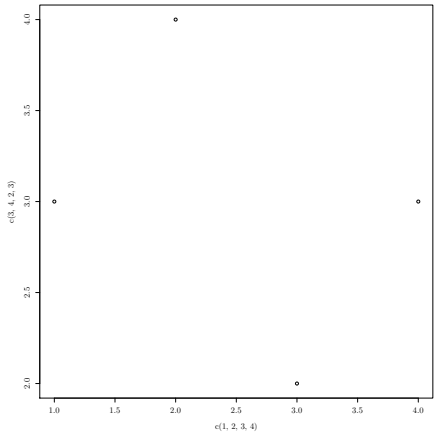
So far we have looked at a set of summary statistics for the `rot` dataframe. These statistics were however all calculated for the full dataset, regardless of the parameters. In reality we would like to show such statistics per type (that is unique combination of `Population`, `Copper` and `Day`). Below we use the `aggregate()` function to calculate the mean number of alive juveniles for unique combinations of `Copper` and `Population`. Please extend this code to also account for the different days.

```
aggregate(rot$Alive_Juv,by=list(rot$Population, rot$Copper), mean)

##          Group.1 Group.2        x
## 1     Commercial    high 5.888889
## 2      Pollution    high 2.055556
## 3  Postpollution    high 1.722222
## 4       Recovery    high 2.611111
## 5     Commercial     low 7.111111
## 6      Pollution     low 2.666667
## 7  Postpollution     low 1.666667
## 8       Recovery     low 2.944444
## 9     Commercial  medium 5.833333
## 10     Pollution  medium 3.833333
## 11 Postpollution  medium 2.444444
## 12      Recovery  medium 1.888889
```

Now repeat this procedure to also calculate the average number of alive adults per `Population`, `Copper`-treatment and `Day`.
Finally adapt the script to also calculate other parameters, such as the median, maximum counts, minimum counts, standard deviation and variance.

3.4 Step 4: Plot your data

Let us have a look at how your rotifer population does over time. For this we use the `plot` function in `R`. The first argument contains the `x` variables and the second the `y` variables.



However, if we call the function like this, `R` wil automatically set the axis (the limits and the location of the tick marks). This is okay for quick plots, but for reports you will probably want a bit more control. One way to do so is shown in the following.
We will focus first on the populations with the low copper treatment. For simplicity we will only look at the total population numbers. Start by making an empty plot with the correct labels then draw the axes in this plot. Later we will lines add lines to this plot:

```
plot(0, 0, type="n", xlab="Day",
     ylab="Total population size",
     main="Low copper treatment",
     axes=FALSE, xlim=c(1, 3), ylim=c(0, 30))
axis(side=1, at=c(1, 2, 3))
axis(side=2, at=c(0, 5, 10, 15, 20, 25, 30))
```

Now we need to prepare the data that we are interested in. First we use the `factor()` function in the beginning to make sure that the variable `Population` will be plot in the correct order. Furthermore we add a new variable `Total` that corresponds to the total number of individuals (adults and juveniles).

```
rot$Population <- factor(rot$Population,
                         levels=c('Commercial', 'Postpollution',
                           'Recovery', 'Pollution'), ordered=T)

rot$Total <- rot$Alive_Juv + rot$Alive_Adult
```

To select only the low copper treatment entries, use the function `subset()` and store this part of the dataset in a new variable `temp`. Make sure you understand what this function does.

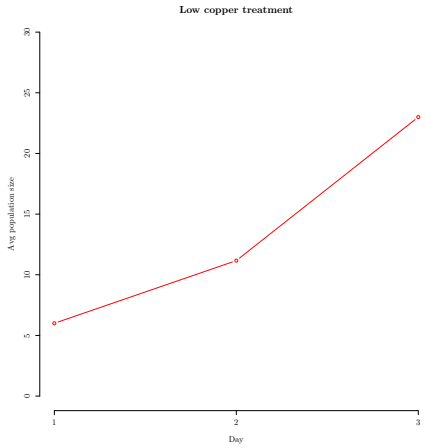
```
temp <- subset(rot, Copper=='low')
```

Next use `aggregate()` to calculate the mean number of total individuals alive per day and population. Store this information in a variable named `Tot_mean`. This should give you:

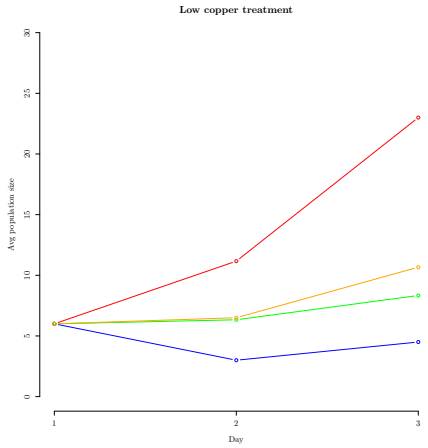
Group.1	Group.2	x
Commercial	1	6.00
Postpollution	1	6.00
Recovery	1	6.00
Pollution	1	6.00
Commercial	2	11.17
Postpollution	2	3.00
Recovery	2	6.33
Pollution	2	6.50
Commercial	3	23.00
Postpollution	3	4.50
Recovery	3	8.33
Pollution	3	10.67

Now we use the command `lines()` to add a line for the commercial populations to the empty plot:

```
temp2 <- subset(Tot_mean, Group.1=="Commercial")
lines(temp2$Group.2, temp2$x, col='red', pch=1, type="b")
```

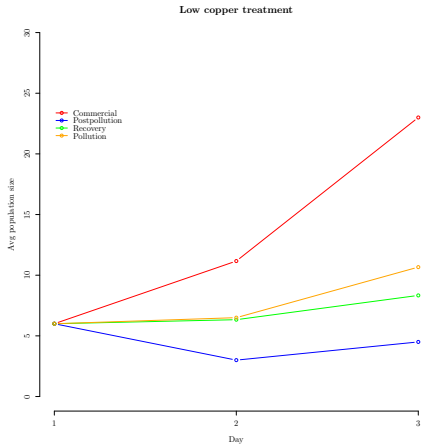


Use these methods to also add lines for the other populations.



We are getting closer, but we aren't there yet, now we want to add a legend to the figure. To do so we use the command `legend()`, with 7 parameters. Use `?legend` to find out what each parameter determines (or just play around with the parameters to find that out):

```
legend(x=1, y=24, col=c("red","blue","green","orange"),
legend=c("Commercial", "Postpollution", "Recovery", "Pollution"),
lty=1, pch=1, bty="n")
```



Finally we would like to introduce error bars in this graph. To do so, we first define a function for calculating the standard error. The standard error (SE) is defined as the standard deviation (sd) divided by the square root of the number of measurements (N) that the mean depends on:

$$SE = \frac{\sigma}{\sqrt{N}}.$$

We use this property to define a new function that calculates the standard error:

```r
SE <- function(x){
  return(sd(x) / sqrt(length(x)))
}
```

Extra: Functions so far we have worked with variables that represent a value or set of values, for example `rot` is a variable that we have defined that contains our data. A function is in a sense a variable that does not contain values, but instead contains a set of tasks. When you call a variable, `R` will show the value of that variable. When you call a function, `R` will perform the set of tasks in that function.

```r
x_sq <- function(x){
  return(x^2)
}
x_sq(3)

## [1] 9
```

Here we have first defined a function (`x_sq`) that takes one input variable (`x`). This function returns the squared value of the input and we can call it by using `x_sq(x)`. Similarly we can also make a function that adds up two numbers and returns the squared sum:

```r
x_y_sq <- function(x,y){
  ans <- x + y
  return(ans^2)
}
x_y_sq(3, 4)

## [1] 49
```

One important thing to realise is that things that happen in a function, usually stay in a function (there are ways around this, but in general it is good practice to keep it that way). This means that the tasks in the function are performed in their own 'world' and as soon as the function is executed, the changes are gone.

```r
x_y_sq <- function(x,y){
  ans <- x + y
  return(ans^2)
}
x_y_sq(3, 4)

## [1] 49

ans

## Error in eval(expr, envir, enclos):  object 'ans' not found
```

We now get an error, because `ans` only exists in the function and is deleted as soon as the function is completed.

```r
ans <- 4
x_y_sq <- function(x,y){
  ans <- x + y
  return(ans^2)
}
x_y_sq(3, 4)

## [1] 49
```

What will now be the value of `ans`?
We have seen that we can not access the version of `ans` that 'lives' inside the function anymore, once the function has run. If you change anything in a function, you best return these changes to `R` at the end of the function using `return()`. Please note that when a function executes `return(x)`, `x` is given back to `R` and anything that comes after it will not be executed.

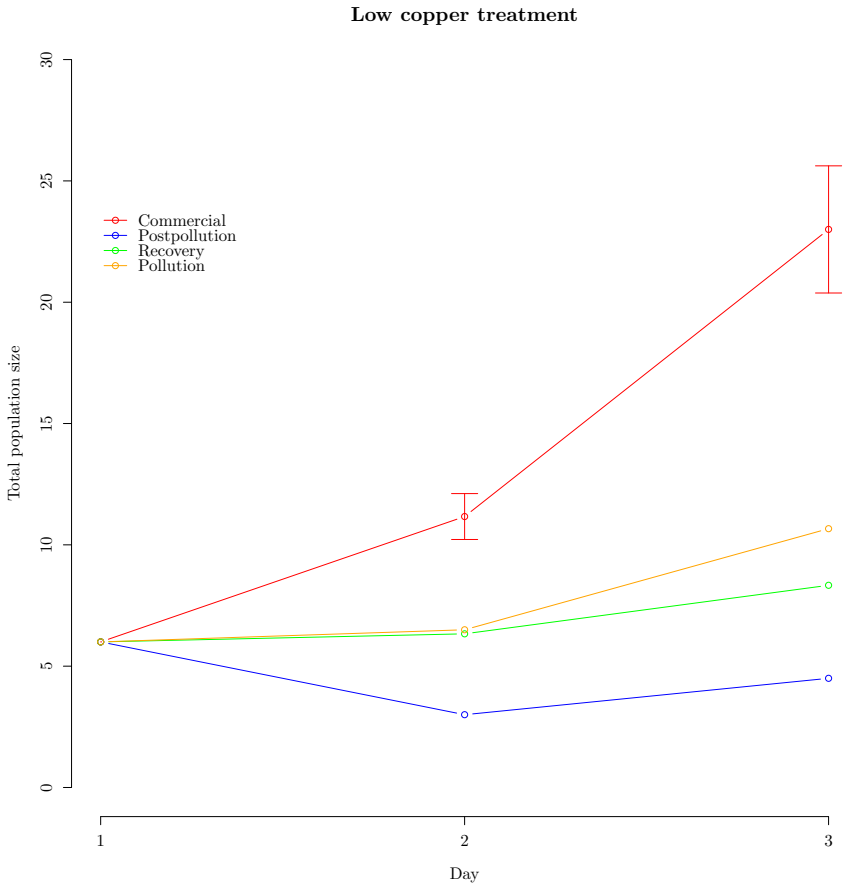
If we now call `SE(x)`, it will return us the standard error of `x`. We use this function together with `aggregate` to calculate the standard errors for all our datapoints in the low copper treatment:

```r
SE_tot <- aggregate(temp$Total, by=list(temp$Population, temp$Day), SE)
```

Group.1	Group.2	x
Commercial	1	0.00
Postpollution	1	0.00
Recovery	1	0.00
Pollution	1	0.00
Commercial	2	0.95
Postpollution	2	1.06
Recovery	2	0.49
Pollution	2	1.20
Commercial	3	2.62
Postpollution	3	1.54
Recovery	3	1.12
Pollution	3	2.14

Finally we use the function `arrows()` (use `?arrows` to understand how this function works or play around with the parameters to see the effects) to add the error bars. For the commercial populations this would work as follows:

```
temp2 <- subset(Tot_mean, Group.1=="Commercial")
temp3 <- subset(SE_tot, Group.1=="Commercial")
arrows(temp3$Group.2, temp2$x-temp3$x, temp3$Group.2, temp2$x+temp3$x,
       angle=90, length=0.1, col='red', code=3)
```

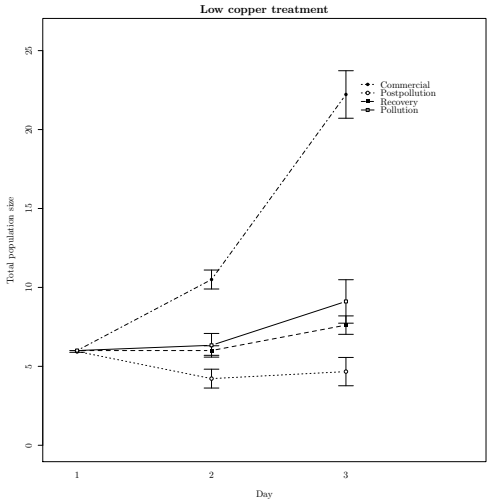


You will notice a warning message about zero-length arrows. What does this mean? Repeat this procedure to also include error bars for the other three lines. Can you now also generate the plots for the other two copper treatments?

The shortcut
First of all the code can be shortened by using for-loops, can you find two place where introducing a for-loop would make the code shorter? Alternatively, you could also write functions for parts of the code that you use

frequently. We are not the first people to deal with this problem and the sciplot package already contains such a function:

```
library(sciplot)
temp<-subset(rot,Copper='low')
lineplot.CI(temp$Day,temp$Total,
            group=temp$Population,fixed=T, ylim=c(0,26),
            xlab="Day",ylab="Total population size",
            main="Low copper treatment")
```



Using functions from packages such as the one above is a great way to increase your coding speed: there are a lot of things that you don't have to type yourself anymore. This also comes with a cost however: you will generally have less flexibility, you don't know which steps happen behing the scenes and you depend on the coding skills of the people that wrote the package.

3.5 Saving your graph

If you want to save your graph, write the code in R following the instructions below:

1. Choose a file name. This can be anything, for example, "Nameyourgraph.jpg".
2. Open a jpeg file by typing jpeg(file = "Nameyourgraph.jpg").
3. Use the plot command to make graphs. Because you typed the jpeg command, R will send all graphs to the jpeg file, and the graphic output will not appear on the screen.

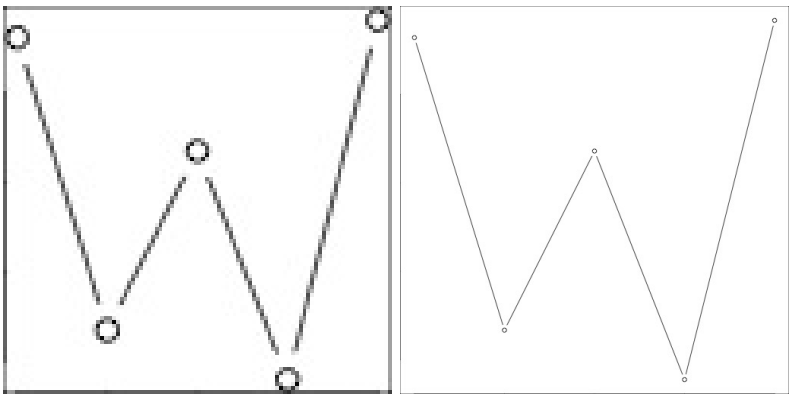
4. Close the jpeg file by typing: `dev.off()`.

File types for images

In the example above we have described how to save a graph as a `jpg`. Saving your graph as a `jpg` is not always the best way to go. As you may notice the quality of the output is not always as good. You can try to increase the quality of the image by adding the parameters `width` and `height` to the `jpg` command. We will now compare two graphs, each with different numbers for the `width` and `height:`. We have cut out the axis labels for comparison purposes.

```
x<-1:5
y<-runif(5)
jpeg('jpg1.jpg',width=100,height=100)
par(mar=c(0,0,0,0))
plot(x,y,type="b")
dev.off()

jpeg('jpg2.jpg',width=512,height=512)
par(mar=c(0,0,0,0))
plot(x,y,type="b")
dev.off()
```

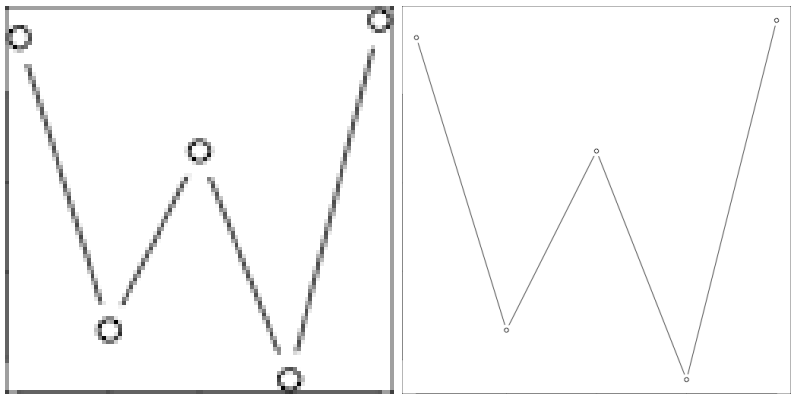
Compare the following two plots:



Although the plot on the right hand side already has a higher quality, if you zoom in you will still notice that it contains some noise around the lines. Instead you could opt for a `png` image.

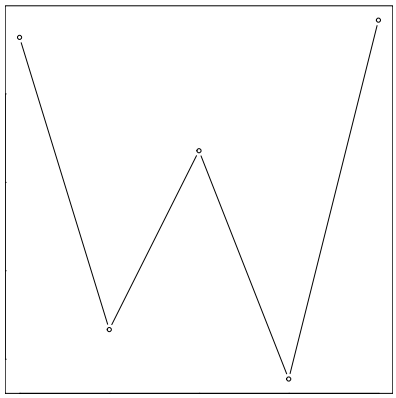
```
png('png1.png',width=100,height=100)
par(mar=c(0,0,0,0))
plot(x,y,type="b")
dev.off()

png('png2.png',width=512,height=512)
par(mar=c(0,0,0,0))
plot(x,y,type="b")
dev.off()
```

Compare the following two plots:



Notice how the noise is now gone. The quality is however still not perfect. You can increase the quality by further increasing the `height` and `width` of the image. Alternatively you could also opt for a different file format: `pdf`: (Note that for this file format height and width can be specified but have a different meaning than for the previous file formats)

```
pdf('pdf1.pdf')
par(mar=c(0,0,0,0))
plot(x,y,type="b")
dev.off()
```



Try to zoom in until you see the pixels: that's right, you can not find them. The `pdf` format is very high quality, but unfortunately this adventage is usually lost when you include it in a MS word document. You can however use them with full quality in LaTeXdocuments. LaTeXis also the program that we used for writing this pactical. If you want to try it without having to install anything and not even having to create an account:
`www.overleaf.com`
Some more file formats are available for creating pictures, for example `TIFF` and `BMP`. Realise how annoying it is if you made the perfect graph but saved it as a low quality file...