UNIVERSITÄT
BERN

R COURSE

VITOR SOUSA, LAURENT EXCOFFIER

(based on slides and course materials from Laurent Excoffier)

Day 2 Data Types

Objectives

At the end of this session you should be comfortable using R and know how to get help when you get stuck.

Things you should learn:

- the basic data structures in R.
- some basic syntax.
- how to create data objects.
- how to use the help function.

Data Objects in R

- Data is stored as objects
- Standard data objects:
  scalars, vectors, matrices and arrays,
  characters and strings, factors,
  lists and data frames

- Data can be entered by hand or read from other sources
  (keyboard, files, internet)

Day 2 Data Manipulation

Adding elements

Vectors can be combined by using the **c()** function:

```
v1 <- 1:5; v2 <- 6:10
v3 = c(v1,v2)
```

We can add columns or rows to a matrix with the functions **rbind()** and **cbind()**, respectively:

```
mat1<-matrix(1:10,nrow=2)   # creates a 2x5 matrix (2 rows, 4
columns)
mat2<-matrix(11:20,nrow=2)
mat3 <- cbind(mat1,mat2)    # mat3 is a 2x10 matrix
mat4 <- rbind(mat1,mat2)    # mat4 is a 4x5 matrix
```

Remove elements of a vector or matrix:

Elements of vectors can be removed by using negative indices:

```
v <- v[-3]
```

removes 3rd element of vector v

For matrices:

```
mat <- matrix(c(1:100),nrow = 10)
mat <- mat[-3,]
```

removes 3rd row of matrix mat

RECALL: matrices are accessed with **[*rows_index, columns_index*]**

```
mat <- mat[-c(3,5),]
```

removes 3rd and 5th row

```
mat <- mat[-3,-5]
```

removes 3rd row and 5th column

NOTE: a single line of a matrix is a vector by default.

For matrices:

```
rows <- mat[2,]
str(rows) # This is a vector
```

This can create problems, if the functions you use expect matrices as input.

To ensure that you keep a matrix, use **drop=FALSE**

```
rows <- mat[2,,drop=FALSE]
str(rows)  # This is a matrix
```

Naming elements in a vector or matrix

We have seen previously that we can access elements by their indices. We can also give names to elements of objects and use the names to access the elements:

```
v <- c(1:3)
names(v) <- c("a","b","c")
v["c"] <- v["a"] + v["b"] # equivalent to v[3] = v[1] + v[2]
```

This also works with matrices:

```
mat <- matrix(1:4,nrow=2)
rownames(mat) <- c("row1","row2")
colnames(mat) <- c("col1","col2")
mat["row1","col2"]
```

Access elements using a vector of Booleans (True and False)

If we do not know the indices of the elements we want to access we can use a vector of Booleans (true and false) to access the elements that are true. For instance:

```
v <- c(0,-3,2,8,0,-1,3,2)
vectorTrueFalse <- v > 0 # get TRUE when v>0 and FALSE otherwise
vectorTrueFalse
[1] FALSE FALSE TRUE TRUE FALSE FALSE TRUE TRUE
```

The vector ***vectorTrueFalse*** is a variable with the same length as the vector **v** and contains TRUE for the elements of the vector ***v*** that are larger than 0. In our example these are the index (3,4,7,8)

We can now obtain the positive elements of ***v*** by:

```
v[vectorTrueFalse]
```

and the number of elements that are larger than 0 by using the function **sum()**:

```
sum(vectorTrueFalse)
[1] 4 # The number of elements that are true are the sum of
vectorTrueFalse (TRUE=1 and FALSE=0)
```

Access elements using index vectors

If we do not know the indices of the elements we want to access we can use the function ***which()***. This function returns the indices of the elements that satisfy a given logical condition. For instance:

```
v <- c(0,-3,2,8,0,-1,3,2)
ind <- which(v > 0)
```

The vector ***ind*** now contains all the indices of the elements of the vector ***v*** that are larger than 0. In our example this is (3,4,7,8)

We can get the positive elements of ***v*** by:

```
v[ind]
```

and the number of elements that are larger than 0 by:

```
length(ind)
```

Recycling

R has a peculiar behavior when doing operation on arrays of different lengths
For instance, suppose we want to add the following two vectors:

```
x=1:3
y=1:6
x+y
[1] 2 4 6 5 7 9
```

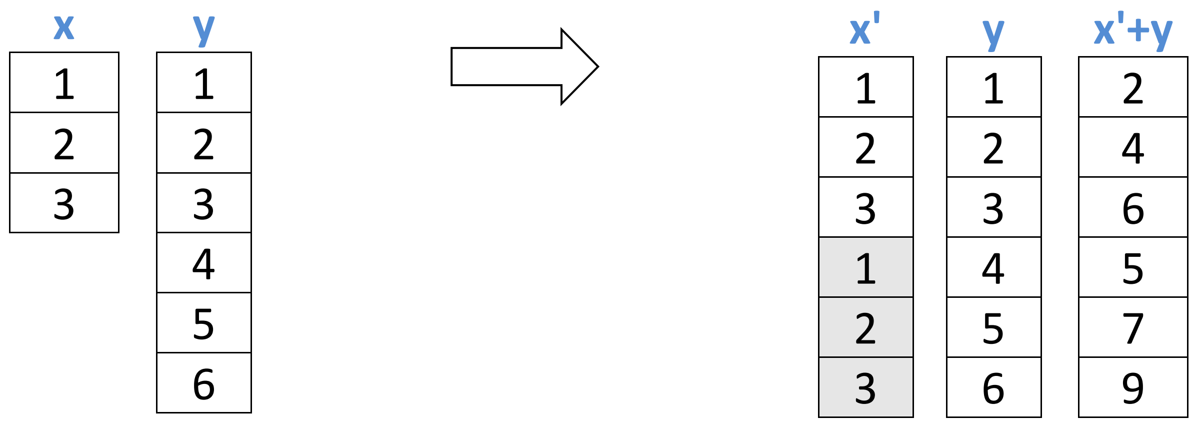
What happened is that the content of **x has been recycled** as x' to build a vector of size 6 to be added to y

Recycling

R has a peculiar behavior when doing operation on arrays of different lengths
For instance, suppose we want to add the following two vectors:

```
x=1:3
y=1:6
x+y
[1] 2 4 6 5 7 9
```

What happened was that the content of **x has been recycled** as x' to build a vector of size 6 to be added to y



Recycling

So in general, **the shorter vector is recycled to match the length of the largest vector**.
A warning message is issued if the length of long vector is not a multiple of the length of the short vector

You can use this property to do clever operations, like

```
1:5/1:10
```

Recycling can also occur when building matrices:

```
cbind(1:3,1:12)
```

or when concatenating vectors an scalars

```
paste("i", 1:3,sep=".")
```

but be careful when combining vectors and matrices

```
s=matrix(rep(1:6,3), ncol=3); t=1:9; s+t
```

Look at the result and explain what happened

Sample data

```
data=1:10
```

#permutation = sampling without replacement (default)

```
sample(data,size=10)
```

#sampling with replacement (bootstrapping)

```
samp=sample(data, size=10, replace=T); samp
```

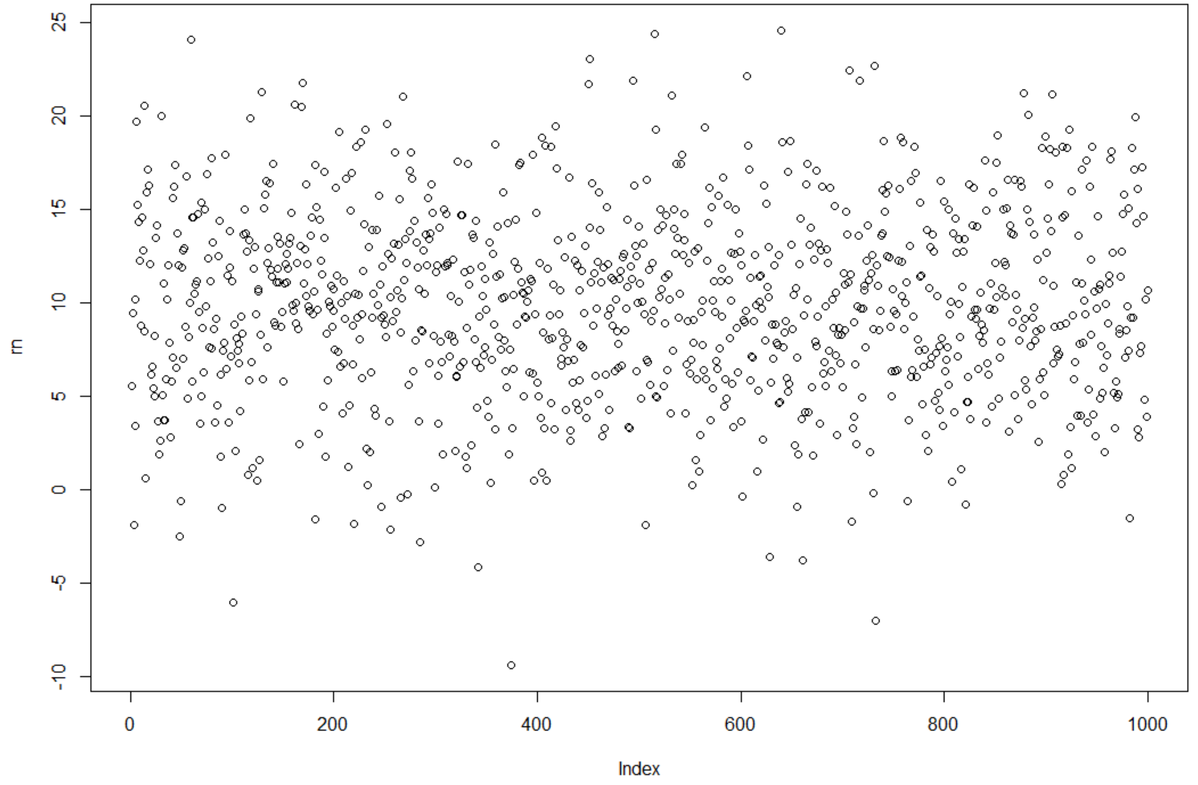
#quickly compute and plot how many times numbers have been drawn

```
table(samp)
plot(table(samp))
hist(samp, breaks=(1:11)-0.5, xlim=c(0,12))
```

Generate random normally distributed numbers

We shall look at that in more details on day 3

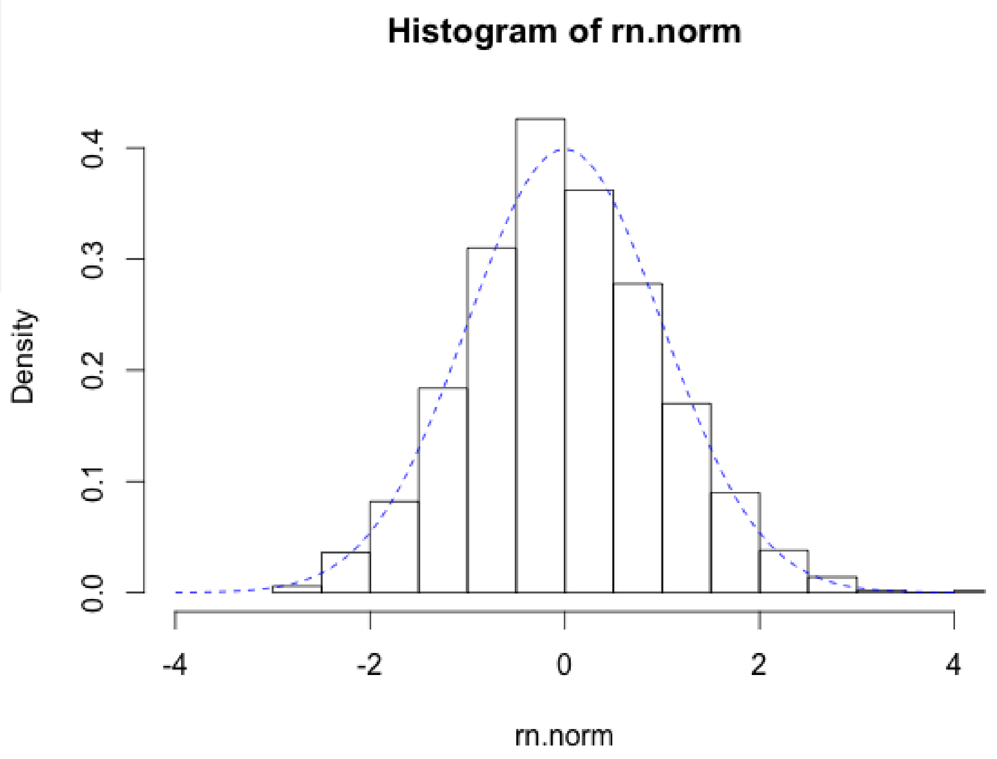
```
rn=rnorm(1000, mean=10, sd=5)
plot(rn)
```



Generate random normally distributed numbers

We shall look at that in more details on day 3

```
rn=rnorm(1000, mean=10, sd=5)
plot(rn)
rn.norm=(rn-mean(rn))/sd(rn)
hist(rn.norm, xlim=c(-4, 4), freq=F)
curve(dnorm(x), from=-4, to=4,
      add=TRUE, lty=2, col="blue",
      lwd=2)
```



Sorting

The function ***order*** can be used to sort a vector, a matrix or a data frame. ***order*** returns a permutation that rearranges its argument into ascending or descending order.

```
m <- matrix(sample(1:20),ncol = 2) # creates a random 10x2 matrix
```

We want to sort the matrix with respect to the first column without breaking ties between entries in rows.

```
ind<-order(m[,1],decreasing=FALSE) #m[,1] is the first column of m
```

ind now contains the order of the elements of the first column in ascending direction. We now rearrange the order of each column:

```
m <- m[ind,]
```

Missing Data

R denotes missing data by NA (not available).
We can ask if an object x is NA:

```
is.na(x)
```

Get indices of missing data:

```
x <- c(1:3,NA)
ind <- which(is.na(x))
```

Create new vector without missing data:

```
y <- x[!is.na(x)]
y <- x[-ind]
```

Most functions can handle missing data. We just need to tell them how to interpret NAs, e.g.:

```
mean(x,na.rm=TRUE)  # removes missing values before calculation
```

Data Objects in R

UNIVERSITÄT BERN

- Data is stored as objects
- Standard data objects (or data types) in R:
  - scalars,
  - vectors,
  - matrices and arrays,
  - characters and strings,
  - factors,
  - lists,
  - data frames

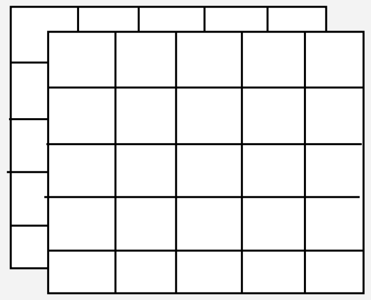
- Data can be entered by hand or read from other sources (keyboard, files, internet)
- You can check the type of variable using the following functions
  - str(), class(), attributes(), names(), colnames(), rownames()

Arrays

Arrays are like matrices but can have more dimensions.

For instance, you can combine 2 5x5 matrices to a 2x5x5 array:

```
mat1 <- matrix(c(1:25),nrow=5,ncol=5)
mat2 <- mat1 + 10
A <- array(dim=c(5,5,2))
A[,,1] <- mat1
A[,,2] <- mat2
```



Arrays can be useful to store and/or visualize multidimensional data, e.g., time evolution of 2D data.

Example: vid1.mp4

Characters, Strings

Characters are letters or symbols. Strings are a collection of several characters (not a vector of characters!).

Indicated by " ", e.g., "a", "@", "some text".

Note the difference between the character "1" and the scalar 1.

```
char <- "1"
number <- 1

mode(char)
mode(number)

char + char    # this yields an error: we cannot add two characters
as.numeric(char) + number   # we can, however, convert the
character to numeric with function as.numeric()
```

Strings

Strings can be entered directly:

```
S <- "Hello World!"
```

This is not the same as:

```
c("Hello"," ","World","!") # creates a vector of strings!
```

We can combine strings into a new string using “paste”:

```
paste("Hello"," ","World","!"),sep="")
```

We can also add a numeric (scalar) to a string:

```
paste("this is string", number, sep=" ")
```

Lists

Lists are “everything” data objects.

Combine any set of data objects in a single object:

```
v <- c(1:5)
l <- list (number = 1, word = "hello", vector = v, matrix = mat1)
```

Creates a list that consists of 1 (scalar), “hello” (string), v (vector) and mat1 (matrix). Note that the elements have names (mynumber, myword, etc.)!

Access elements with $-operator or indices [[ ]]:

```
#get the second element of the list:
l$myword
l[[2]]
```

Data Frames

- Most versatile objects in R
- Somewhat like a spreadsheet.
- Each column of the data frame is a vector.
- Within each vector, all data elements must be of the same mode.
- Different vectors can be of different modes but have to be of the same length!

organism	GenomeSizeBP	estGeneCount
"Human"	3000000000	30000
"Mouse"	3000000000	30000
"Fruit Fly"	135600000	13061
"Roundworm"	97000000	19099
"Yeast”	12100000	6034

Create a data frame by hand

We can create a data frame by combing several vectors into one object. Each vector represents a column in a spreadsheet.

Create vector of strings:

```
vector1<-c("Human","Mouse","Fruit Fly", "Roundworm","Yeast")
```

Create two vectors of scalars:

```
vector2<-c(3000000000,3000000000,135600000,97000000,12100000)
vector3<-c(30000,30000,13061,19099,6034)
```

Combine them into a data frame and store it in a variable:

```
GenomeSize <-data.frame(organism=vector1,genomeSizeBP=vector2,
                                      estGeneCount=vector3)
```

Create a data frame by hand

UNIVERSITÄT BERN

We can create a data frame by combing several vectors into one object. Each vector represents a column in a spreadsheet.

Create vector of strings:

```
vector1<-c("Human","Mouse","Fruit Fly", "Roundworm","Yeast")
```

Create two vectors of scalars:

```
vector2<-c(3000000000,3000000000,135600000,97000000,12100000)
vector3<-c(30000,30000,13061,19099,6034)
```

Combine them into a data frame and store it in a variable:

```
GenomeSize <-data.frame(organism=vector1,genomeSizeBP=vector2,
                      estGeneCount=vector3)
```

Name of first column

Create a data frame by hand

We can create a data frame by combing several vectors into one object. Each vector represents a column in a spreadsheet.

Create vector of strings:

```
vector1<-c("Human","Mouse","Fruit Fly", "Roundworm","Yeast")
```

Create two vectors of scalars:

```
vector2<-c(3000000000,3000000000,135600000,97000000,12100000)
vector3<-c(30000,30000,13061,19099,6034)
```

Combine them into a data frame and store it in a variable:

```
GenomeSize <-data.frame(organism=vector1,genomeSizeBP=vector2,
                      estGeneCount=vector3)
```

Element of first column

Access elements of a data frame

Columns of the data frame can be accessed using “$” and their names:

```
GenomeSize$organism    # the first column of the dataframe
GenomeSize[,1]
```

Elements of the columns can then be accessed with their index:

```
GenomeSize$organism[2]   # the second element of the first column
GenomeSize[2,1]
```

Example:
get organisms with small genomes

```
smallGenomes=which(GenomeSize$genomeSizeBP<10^8)
GenomeSize$organism[smallGenomes]
[1] Roundworm Yeast
Levels: Fruit Fly Human Mouse Roundworm Yeast
```

Get information about a data frame

The command “str” reveals information about the structure of our data frame:

```
str(GenomeSize)
```

Output:

```
'data.frame':5 obs. of  3 variables:
 $ organism    : Factor w/ 5 levels "Fruit Fly","Human",..: 2 3 1 4
5
 $ genomeSizeBP: num  3.00e+09 3.00e+09 1.36e+08 9.70e+07 1.21e+07
 $ estGeneCount: num  30000 30000 13061 19099 6034
```

Other useful functions to check what is in a data frame: **class(), attributes()**, **names()**, **mode(), dim(), colnames(), rownames(), plot** (for small data-sets).

Importing data from files

Most datasets are too large to enter by hand. Fortunately, R can read data directly from files:

```
read.table(“data.txt”,header =FALSE,sep = " ", na.strings = "XXX")
```

R is flexible and can read a variety of different formats. We need to specify the parameters so that R can interpret the data correctly. The most important settings include:

Parameter	Values
file	String with filename, e.g., “dataset.txt”
header	TRUE/FALSE, indicates if first row in data file are names of columns
sep	String that indicates how the elements are separated (" ", "\t", ";", etc.)
na.strings	String that is used for missing data, e.g., "NA", "?", etc.

Help

- There is a huge amount of R-functions and the number is growing.
- Impossible to know or remember everything.
- Using help function is crucial!
- If you know the name of the function you want to use, simply type

```
help(’nameoffunction’)
?’nameoffunction’
```

If you don`t know what you are looking for,

```
help.start()
```

is a good starting point. Alternatively,

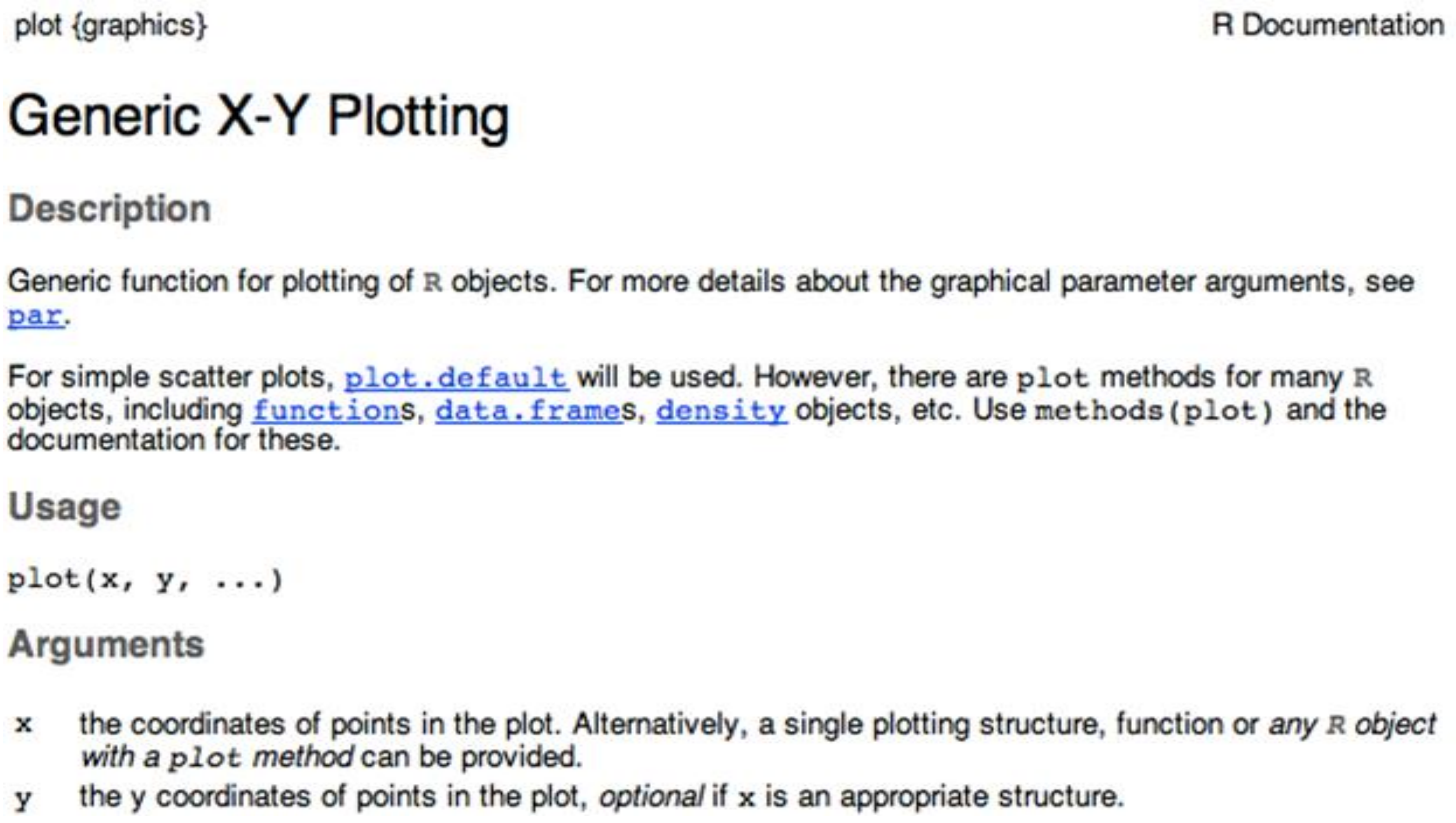
`help.search(xyz)` or

```
??xyz
```

will do a fuzzy search for entries that include or are similar to ‘xyz’.

Help

```
help(plot)
?plot
```



plot {graphics} R Documentation

Generic X-Y Plotting

Description

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see par.

For simple scatter plots, plot.default will be used. However, there are plot methods for many R objects, including functions, data.frames, density objects, etc. Use methods(plot) and the documentation for these.

Usage

```
plot(x, y, ...)
```

Arguments

x	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any* R *object with a* plot *method* can be provided.
y	the y coordinates of points in the plot, *optional* if x is an appropriate structure.

Reshaping vectors, matrices, and arrays

Objects are internally stored as vectors with “breakpoints”. Therefore, it is easy to reshape matrices and vectors using the function “dim” (**dim**ension).

The dimension of a matrix is represented by a vector where the first entry is the number of rows and the second the number of columns.

```
a <- 1:40
dim(a) <- c(4,10) # matrix with 4 rows, 10 columns
dim(a)  <- c(2,5,4) # 3 dimensional array
```

Note that vectors have dimension = NULL

```
dim(a)<-1 # doesn’t work, instead:
as.vector(ma)
dim(a) = NULL
```

Naming elements

We have seen previously that we can access elements by their indices. We can also give names to elements of objects and use the names to access the elements:

```
v <- c(1:3)
names(v) <- c("a","b","c")
v["c"] <- v["a"] + v["b"] # equivalent to v[3] = v[1] + v[2]
```

This also works with matrices:

```
mat <- matrix(1:4,nrow=2)
rownames(mat) <- c("row1","row2")
colnames(mat) <- c("col1","col2")
mat["row1","col2"]
```

Exercises

Hints for exercise 2.1:

- Use the help function for complete syntax of ***read.table***.
- Have a look at StatWiSo2003.txt before you decide which parameters of ***read.table*** you need to specify.

Hints for exercise 2.2:

- Have a look at the help entries for the functions ***str()*** or ***attributes()***.

Hints for exercise 2.3:

- Use the **$** operator to access a column of the data frame.
- Use the functions ***rbind()*** or ***cbind()*** to combine two vectors into a matrix.

Hints for following exercises:

2.4) Look at help for functions ***which()*** and ***length()***.

2.5) Create a new vector with only the non-zero entries from the column “MonMiete”. Use function ***mean()*** to calculate average.

2.6) Use function ***is.na()*** to find missing values.

2.7) Get an index vector with the indices of male/female students from the column “Geschlecht”.

2.8) Look at help for function ***order()***.

R COURSE

Day 2 Graphics

Introduction to Graphics in R

- R graphics functions display graphics on devices
- A device can be a graphic window or a file (for help)
- List of possible devices can be found by `?device` and `dev.list()` lists all currently open devices
- `dev.off()` closes the current device, `graphics.off()` closes all devices
- Two different types of graphics functions:
  - High level functions: create a new graphics device
  - Low level functions: add elements to an existing device

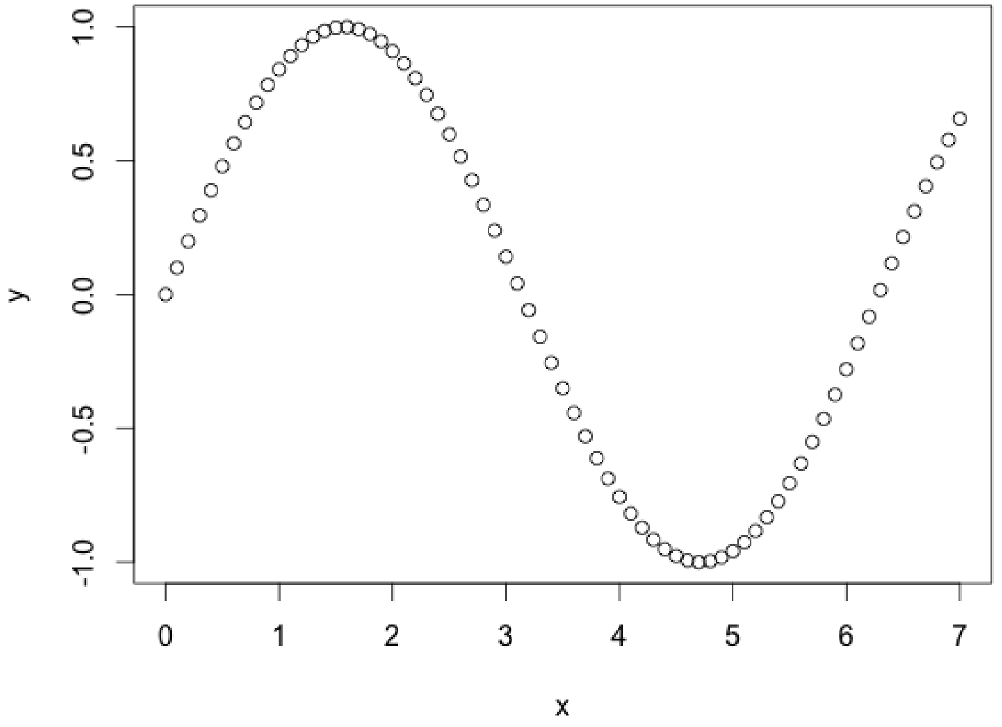
Plot

The most basic function to create a graph is ***plot***:

`plot(x,y)` creates a graphics device and plots the points with coordinates (x[1],y[1]), (x[2],y[2]), …

Example:

```
x<-seq(0,7,by=0.1)
# x = (0,0.1,0.2,0.3, … ,6.9,7)
y<-sin(x)
plot(x,y)
# plots y = sin(x) evaluated at the
# points specified in x
```



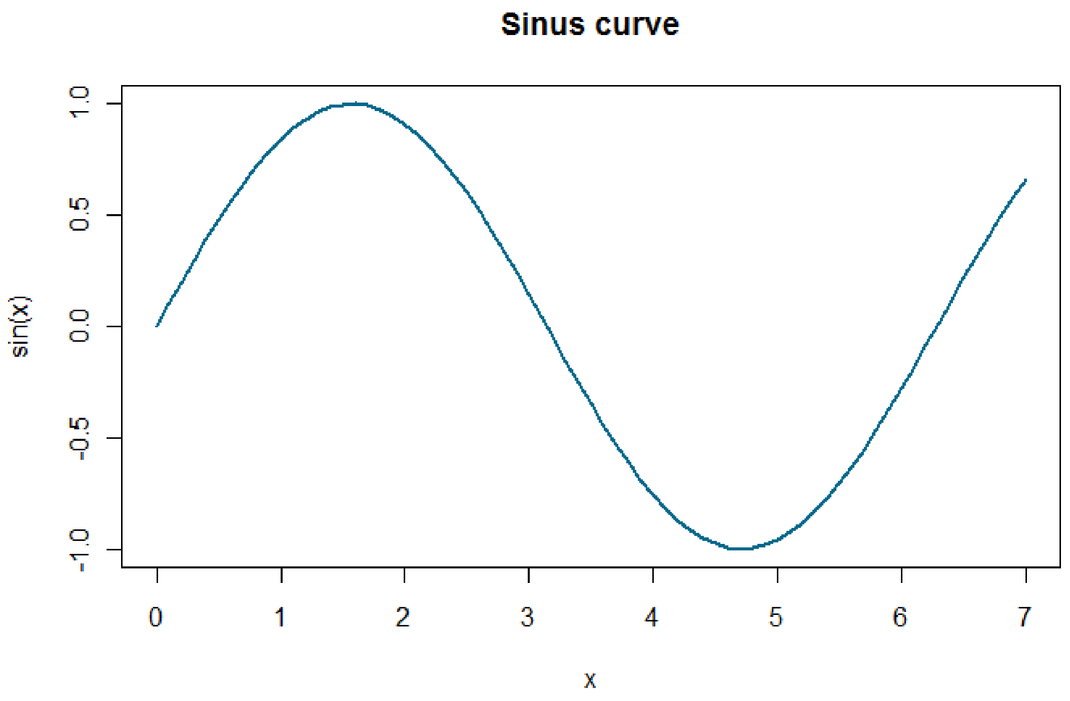
Parameters

We can set parameters such as col (color), lwd (line width), pch (plot-symbol), xlab (label on x-axis), main (title), etc.

R graphics are extremely flexible and almost everything can be controlled by parameters. Use help for a complete list of parameters!

Example:

```
plot(x,y,lwd=2,type=”l”,xlab=”x”,
ylab=”sin(x)”,main=”Sinus curve”,
col=”deepskyblue4”)
```



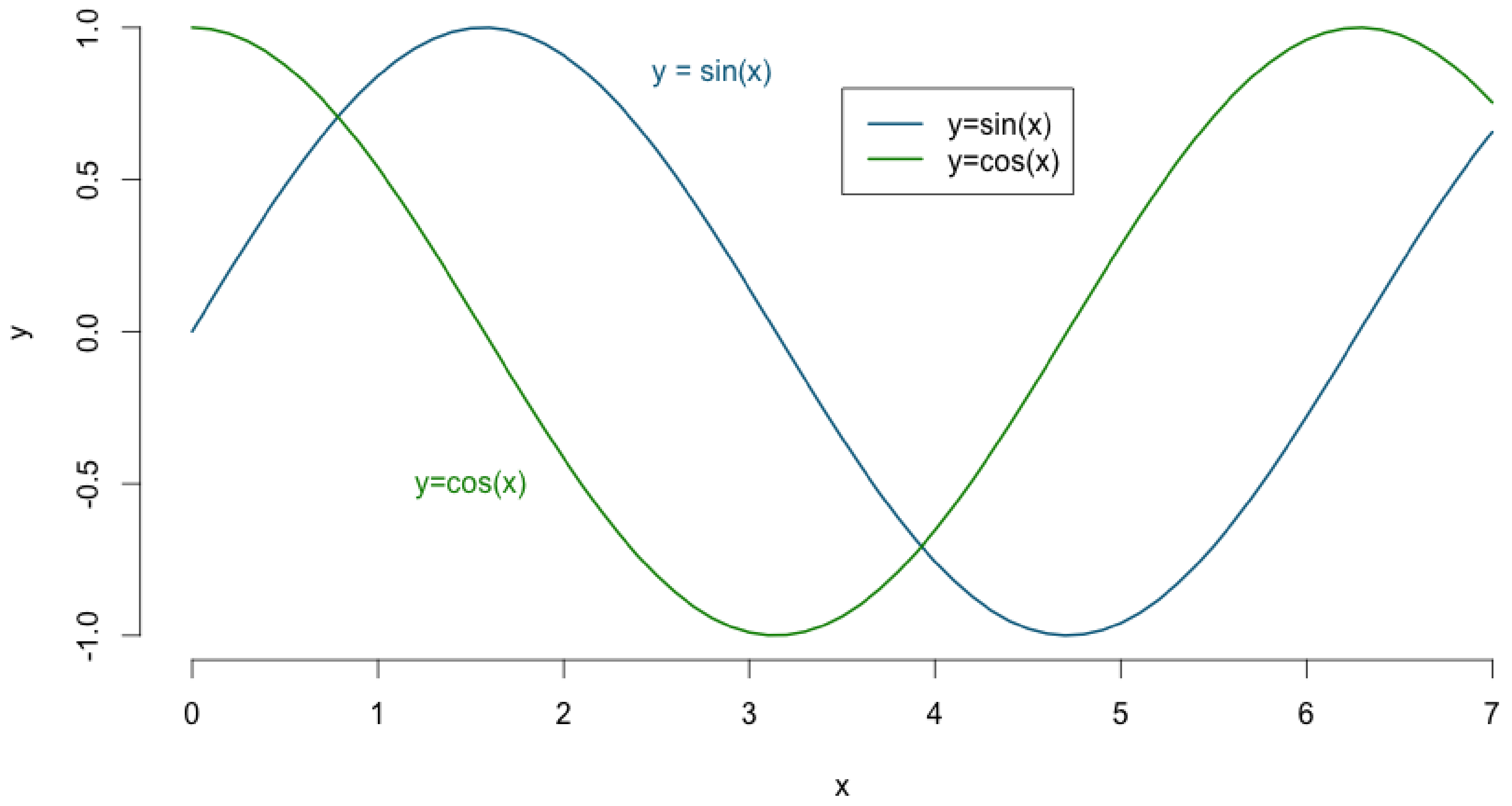
Adding Text, Legends, etc.

We can add text, additional graphs, legends, etc. to a plot. This is done using low level functions.

```
plot(x,y,lwd=2,type="l",col="deepskyblue4",axes=FALSE) #plot without
axes
axis(1) #add axes to plot
axis(2)

text(2.8,0.85,"y = sin(x)",col="deepskyblue4")
# add some text at coordinates x = 2.8, y = 0.85

lines(x,cos(x),lwd=2,col="green4")  # add a cosine plot …
text(1.5,-0.5,"y=cos(x)",col="green4")   # … and label it

legend(3.5,0.8,legend=c("y=sin(x)","y=cos(x)"),
col=c("deepskyblue4","green4"),lwd=2)   # add a legend to the plot
```

Adding Text, Legends, etc.

Result:

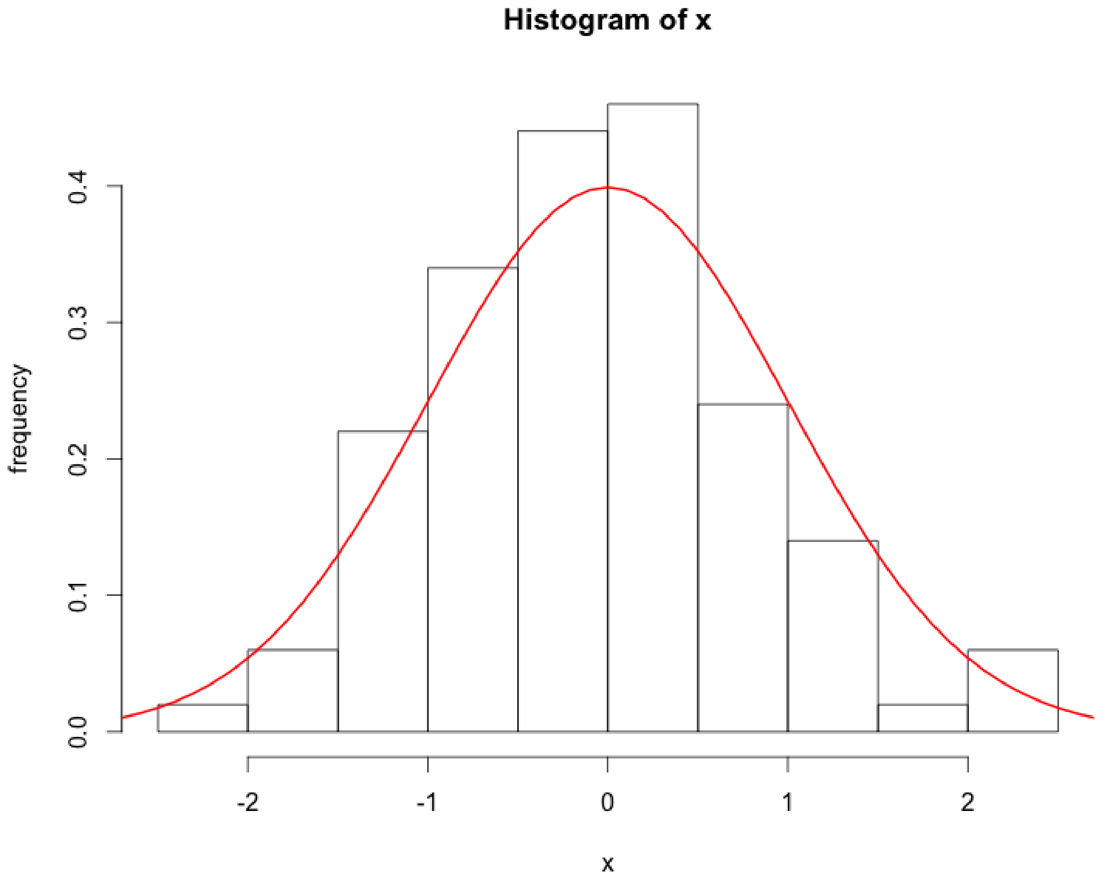


Histograms

A histogram is a graphical representation of the distribution of data. It is an estimate of the probability distribution of a continuous variable. Adjacent rectangles are erected over discrete intervals (bins) with an area proportional to the frequency of the observations in the interval.

Example:

```
# sample from a normal distribution
x <- rnorm(100,0,1)
hist(x,freq=FALSE)
lines(seq(-5,5,by=0.1),
dnorm(seq(-5,5,by=0.1)),
lwd=2,col="Red")
```



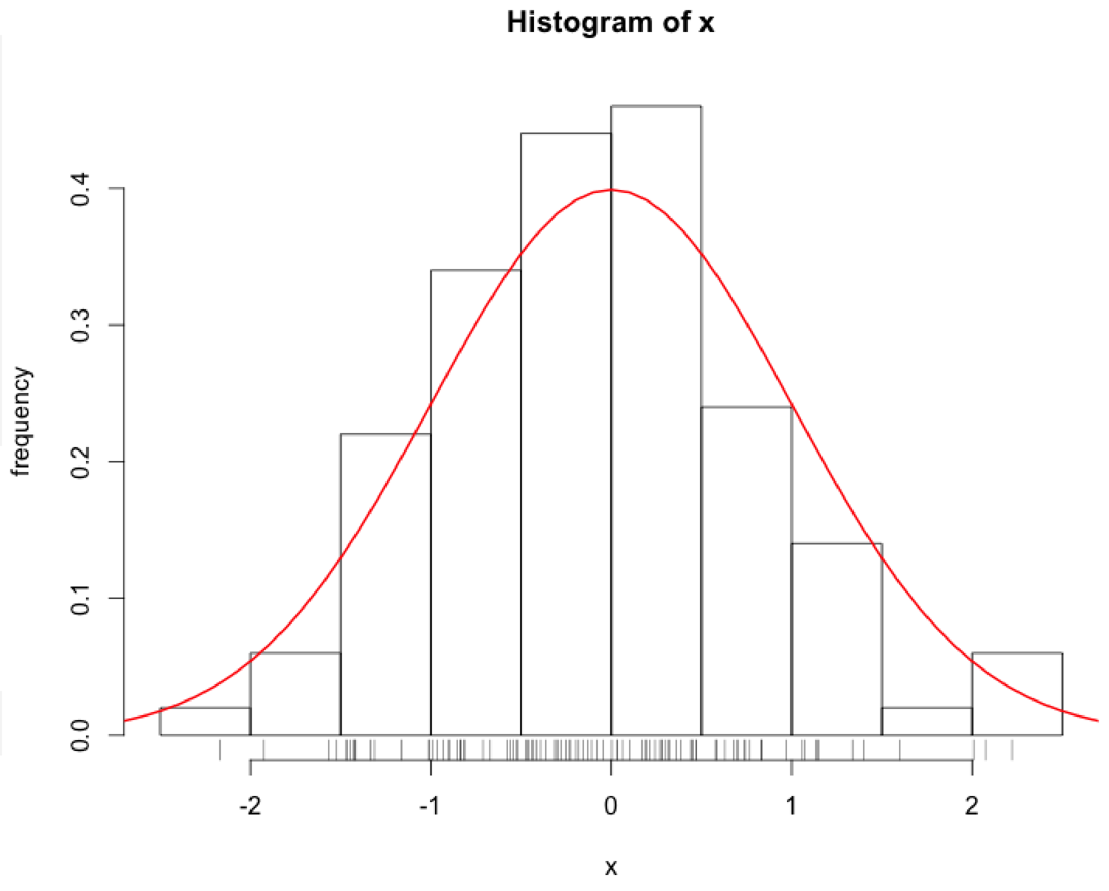
Histograms

A histogram is a graphical representation of the distribution of data. It is an estimate of the probability distribution of a continuous variable. Adjacent rectangles are erected over discrete intervals (bins) with an area equal to the frequency of the observations in the interval.

```
# sample from a normal distribution
x <- rnorm(100,0,1)
hist(x,freq=FALSE)
lines(seq(-5,5,by=0.1),
dnorm(seq(-5,5,by=0.1)),
lwd=2,col="Red")
```

We can add the data-points to the plot by:

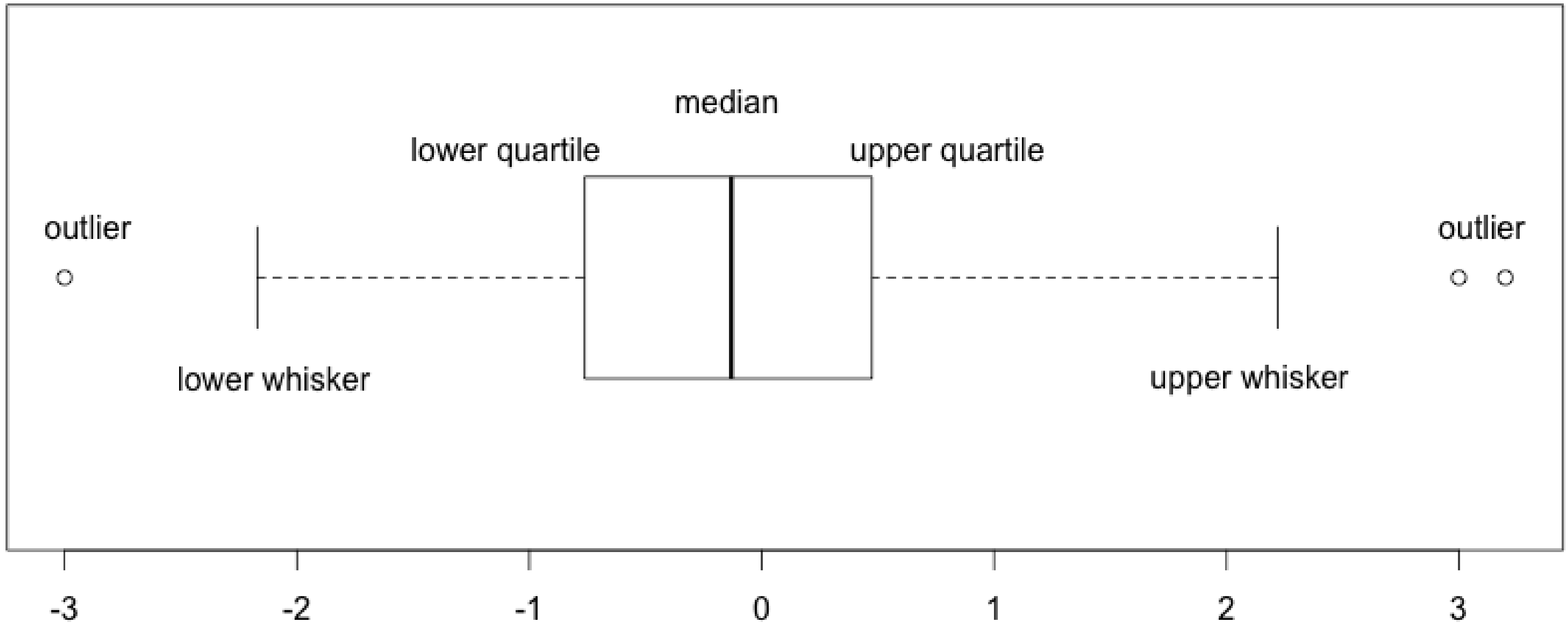
```
rug(x)
```

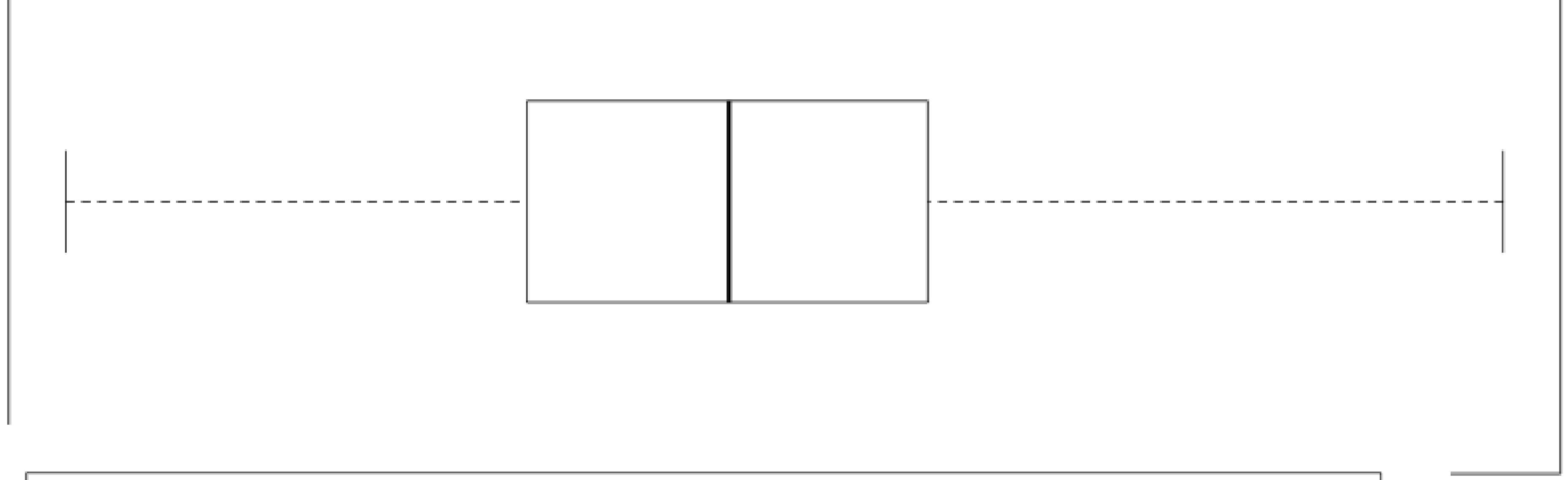


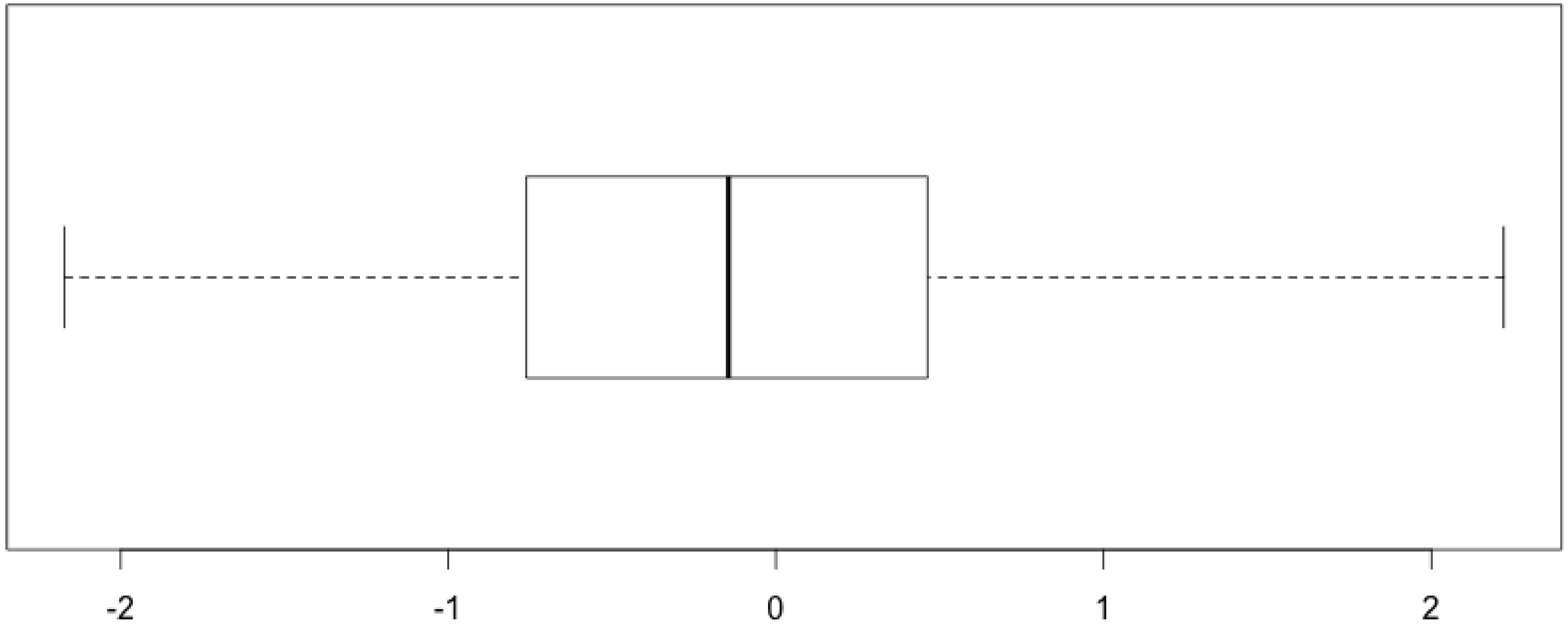
Boxplots

Summary of a distribution:

- 50% of the distribution lie above/below the **median**
- 25% (75%) of the distribution lie below the **lower (upper) quartiles**.
- The **box** contains 50% of the values.
- **Whiskers** extend to the extremes of the distribution
- **Outliers** are represented as circles



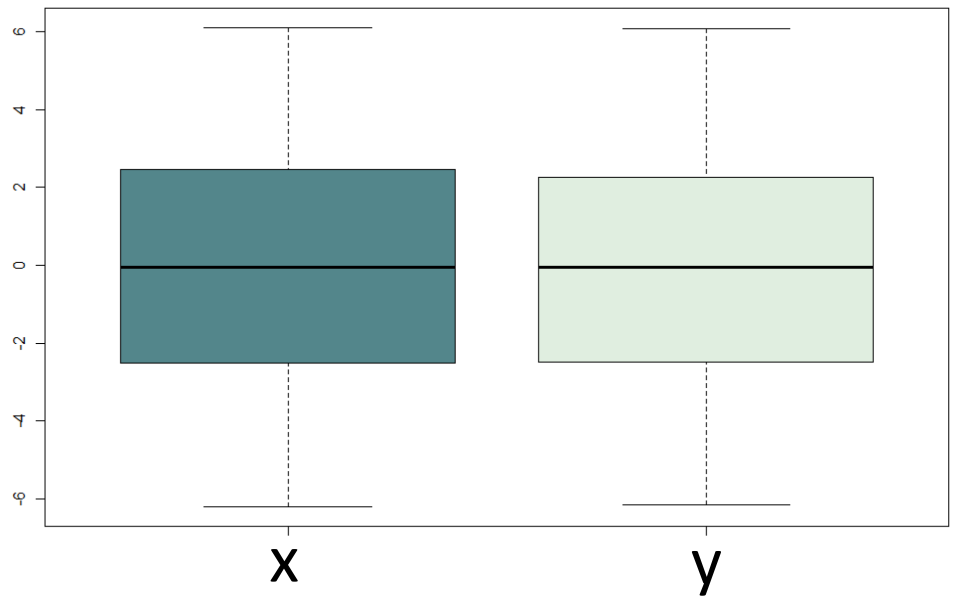



-2
-1
0
1
2

Boxplots

Boxplots are a good way to summarize distributions. However, they have limitations and can be quite misleading in some cases.

For instance:

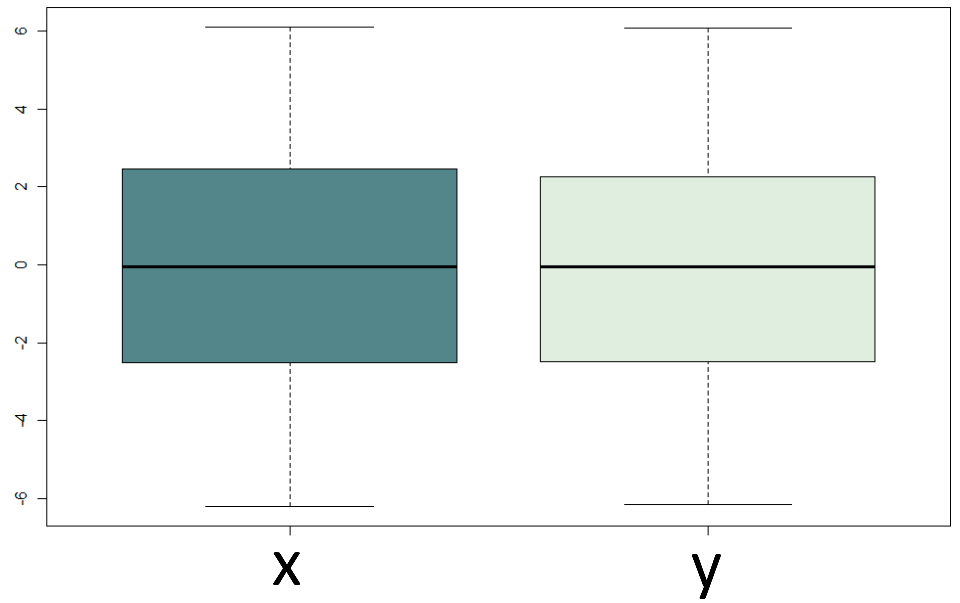


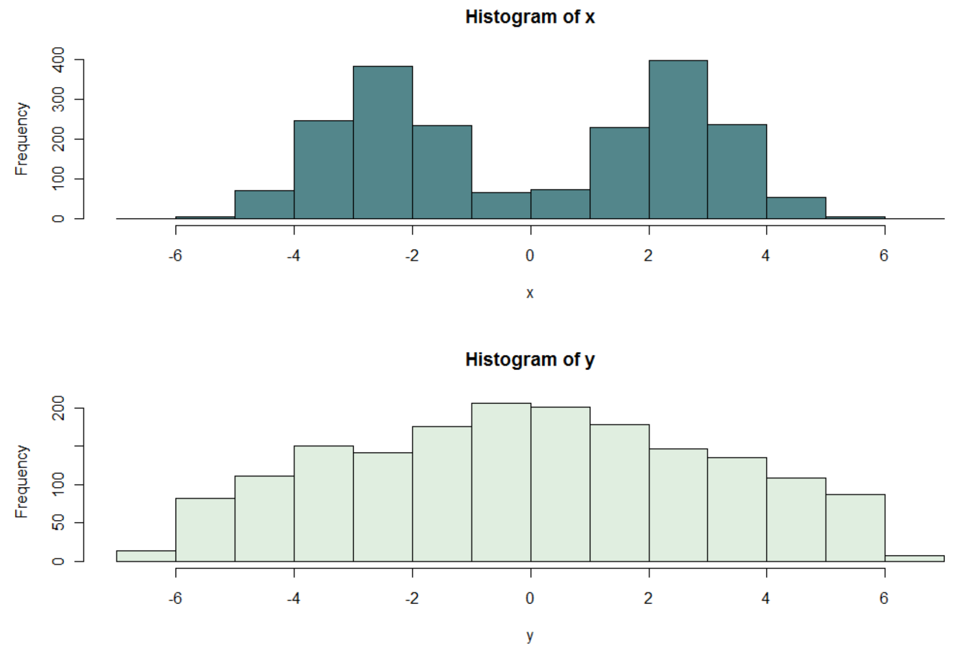
These distributions look quite similar, right?

Boxplots

Boxplots are a good way to summarize distributions. However, they have limitations and can be quite misleading in some cases.

For instance:





… but are actually very different!

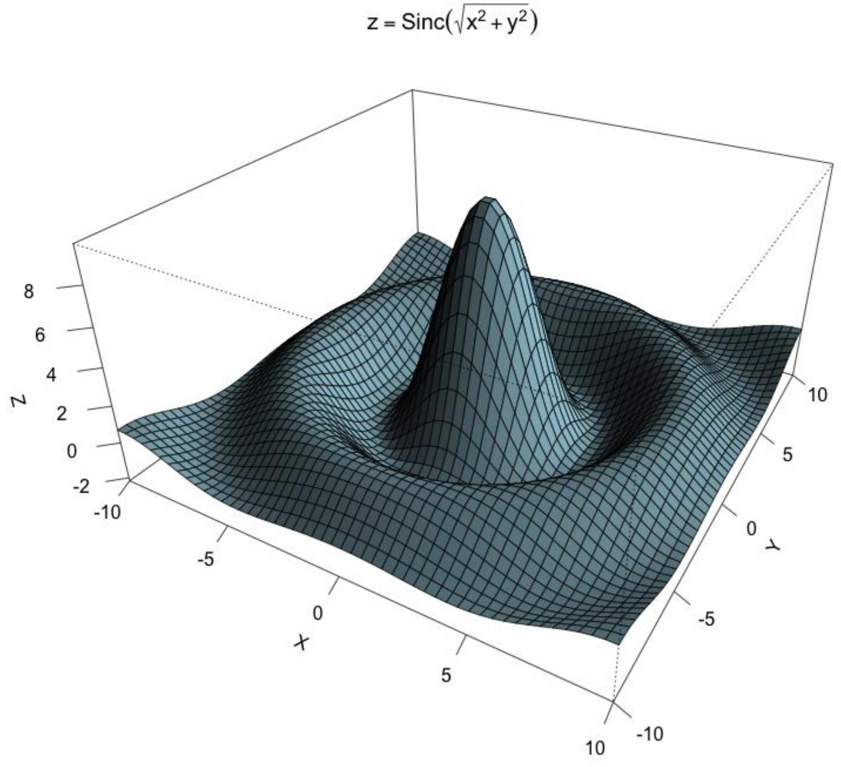
3D PLOTS

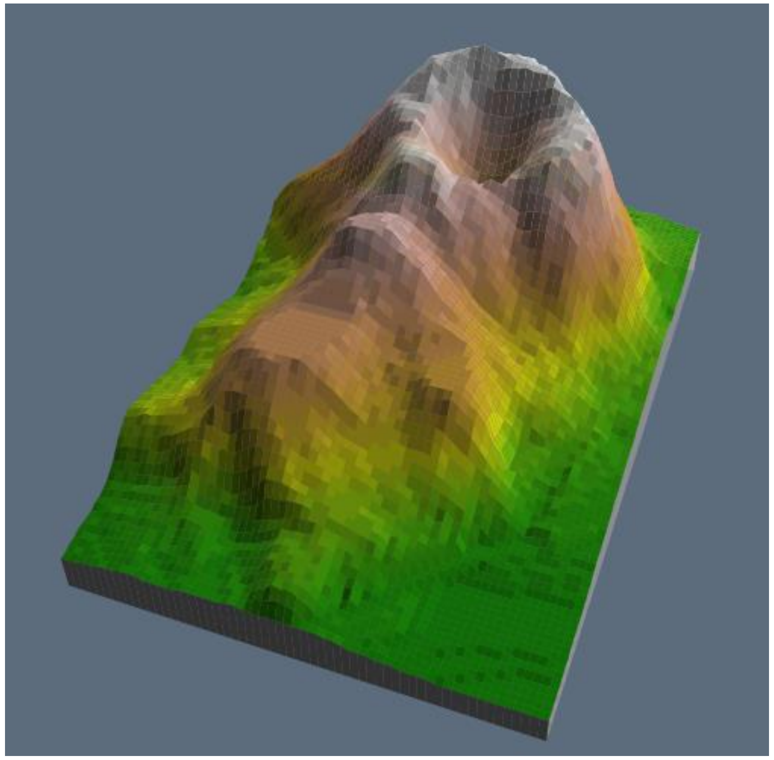
UNIVERSITÄT BERN

R has a variety of functions and packages to create high quality 3D plots.

Example: ***persp(x,y,z)*** plots the points with coordinates x, y and z

```
demo(persp)
```



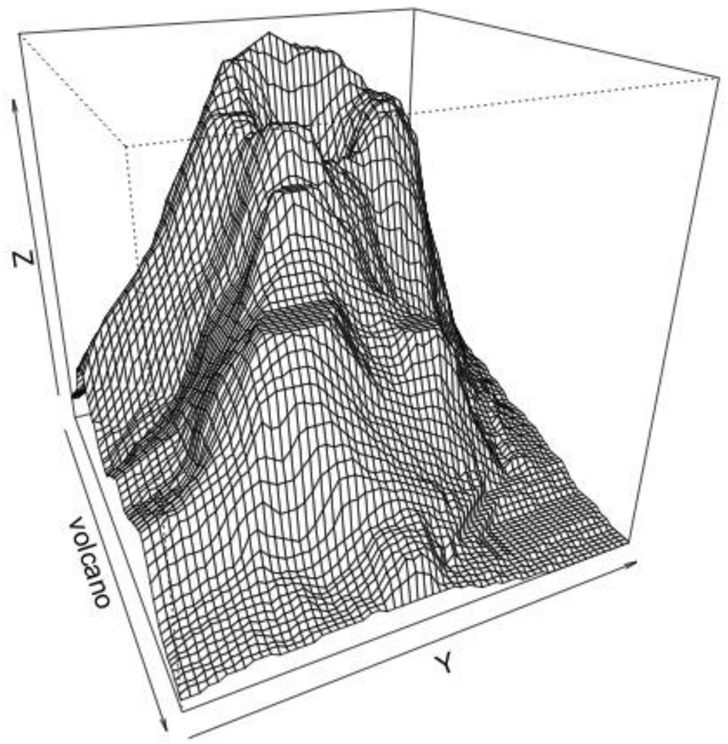


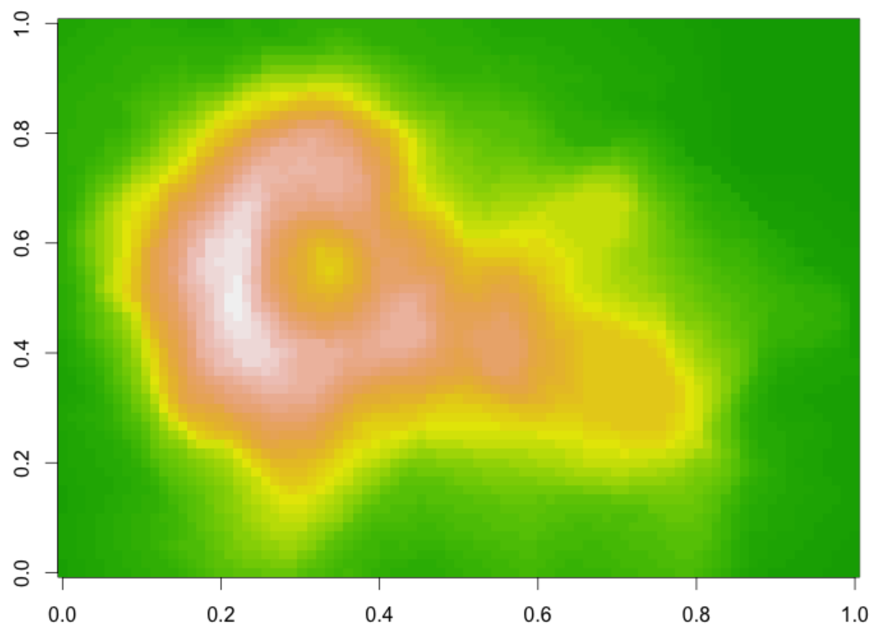
Alternatives to 3D Plots

Although 3D plots can look very nice, 2D plots are often more clear and easier to print.

Example:

```
persp(volcano,theta=45,phi=25)
image(volcano,col=terrain.colors(n=50))
contour(volcano,add=T,cex=2,lwd=2)
```



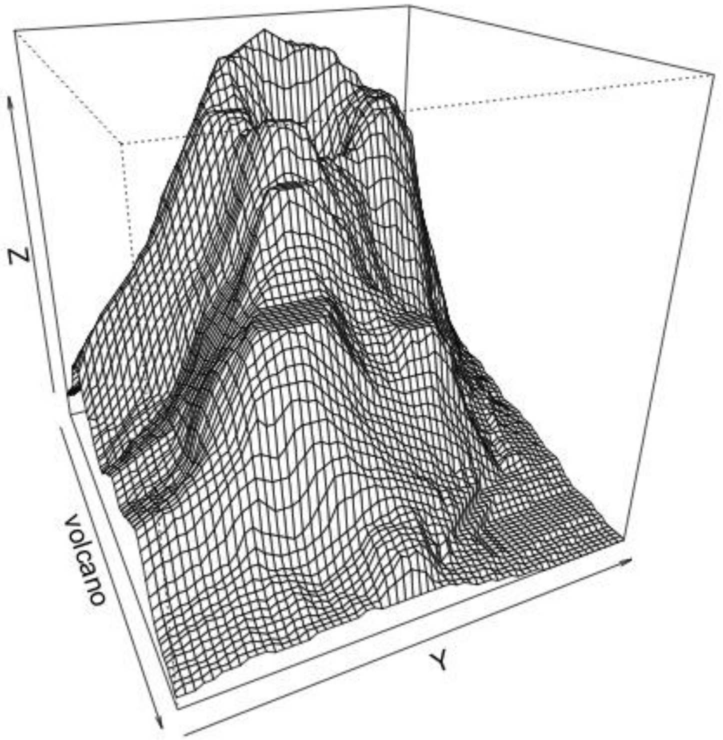


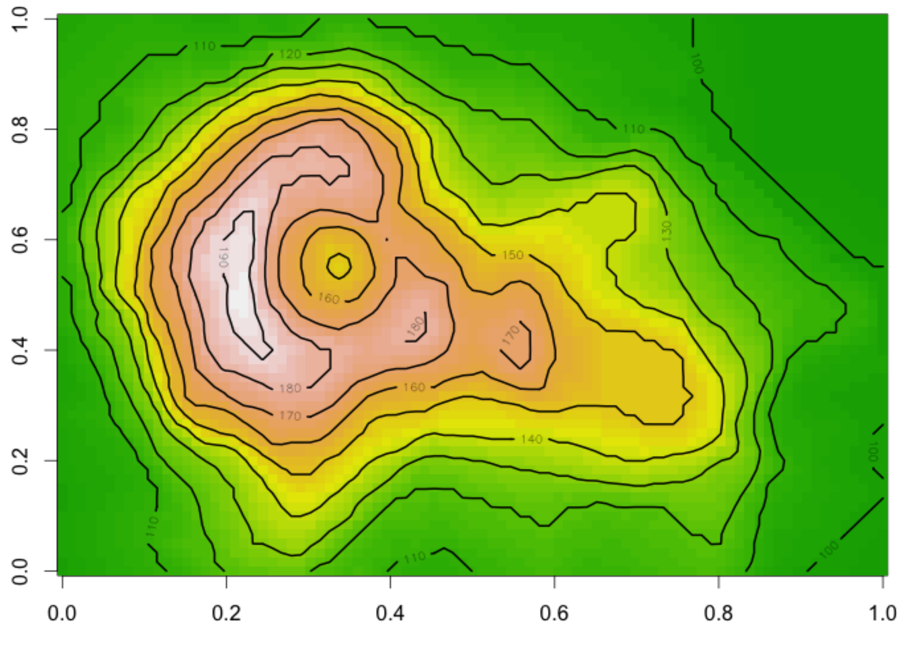
Alternatives to 3D Plots

Although 3D plots can look very nice, 2D plots are often more clear and easier to print.

Example:

```
persp(volcano,theta=45,phi=25)
image(volcano,col=terrain.colors(n=50))
contour(volcano,add=T,cex=2,lwd=2)
```



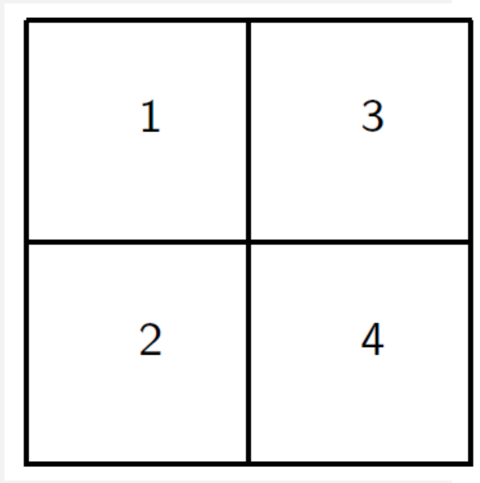


Layout

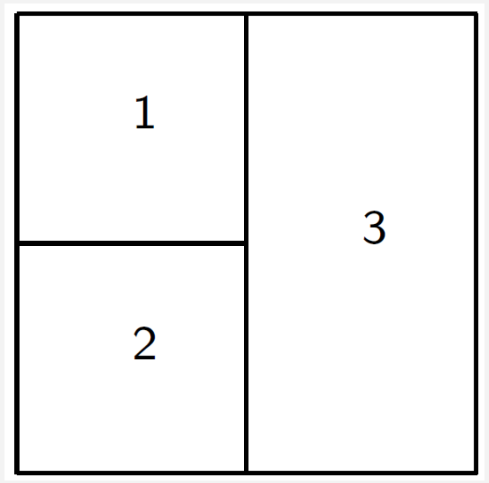
The function ***layout*** partitions the active graphic window in several parts where the graphs will be displayed successively. Its main argument is a matrix with integer numbers indicating the numbers of the “sub-windows”.

Examples:

```
mat <- matrix(1:4, nrow = 2)
mat
 1  3
 2  4
my.lo<-layout(mat)
layout.show(my.lo)
```

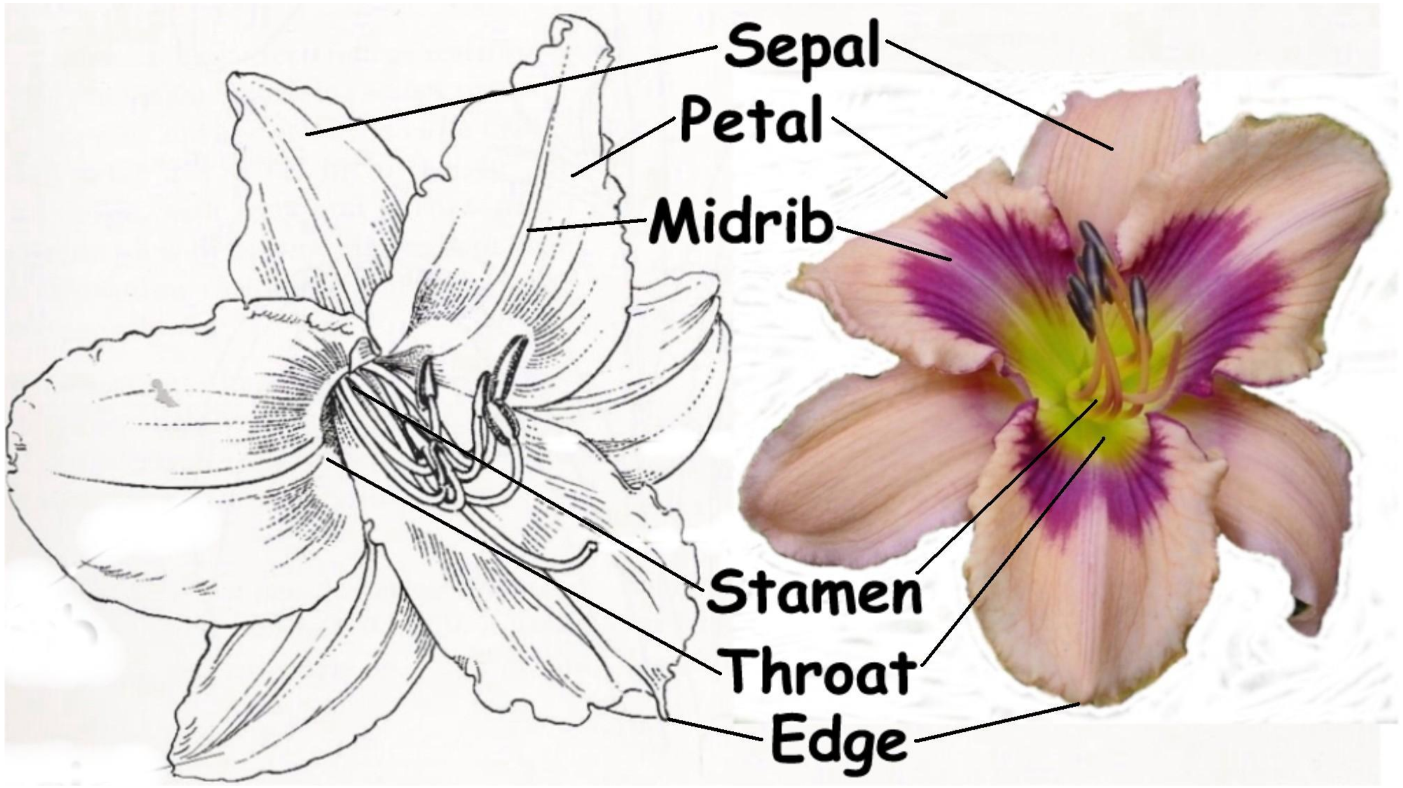


```
mat <- matrix(c(1:3,3), nrow = 2)
mat
 1  3
 2  3
```



Data frames and Graphics

Dataset “Iris”:
- 3 species: setosa, versicolor, viriginica
- 4 measurments: sepal length and width, petal length and width
- 50 samples from each species

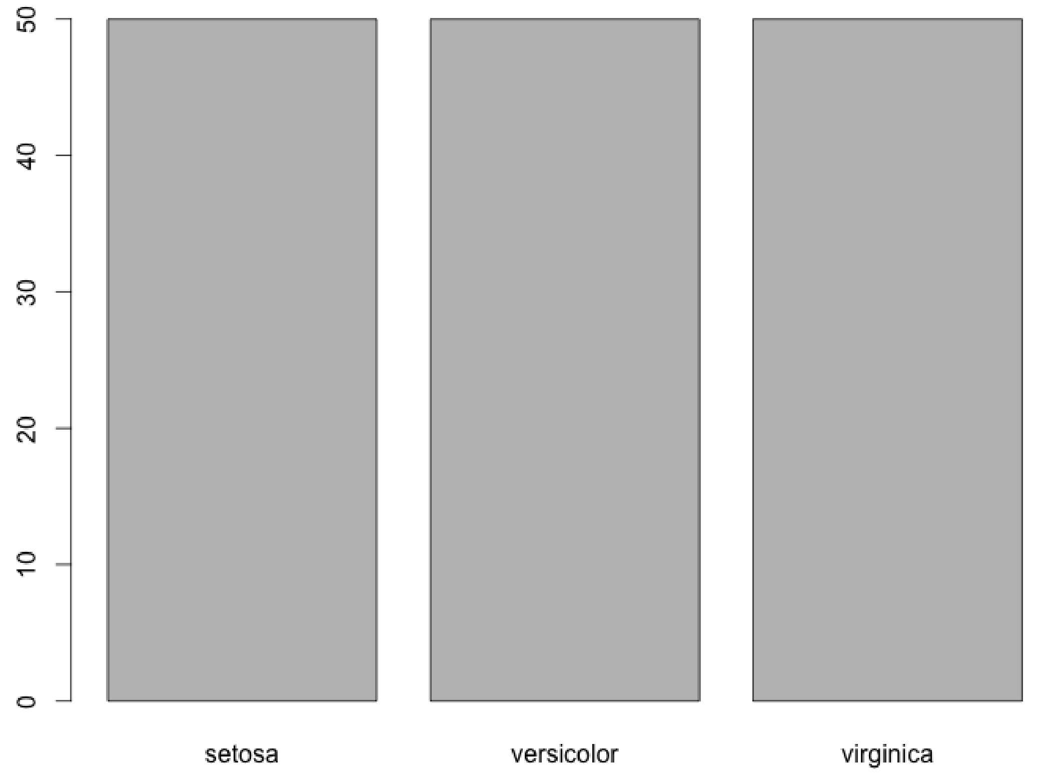


Data frames and Graphics

The function plot automatically determines the data type of the objects that should be plotted. ***iris$Species*** is of type factor, i.e., category types. Plot therefore produces a barplot of the numbers of samples for each species:

```
plot(iris$Species)
```

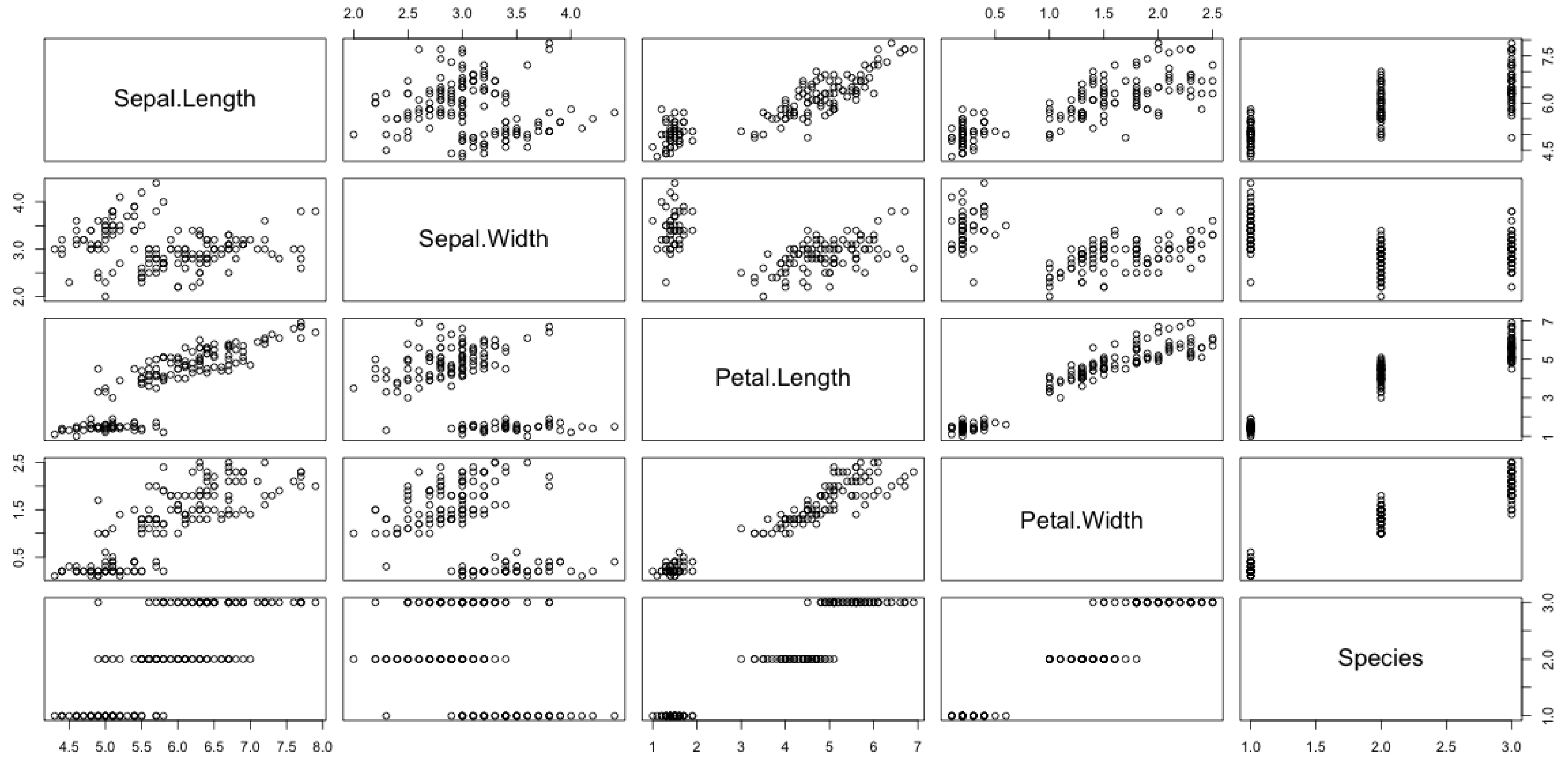
The plot shows that the data set contains 50 samples of each species.



Data frames and Graphics

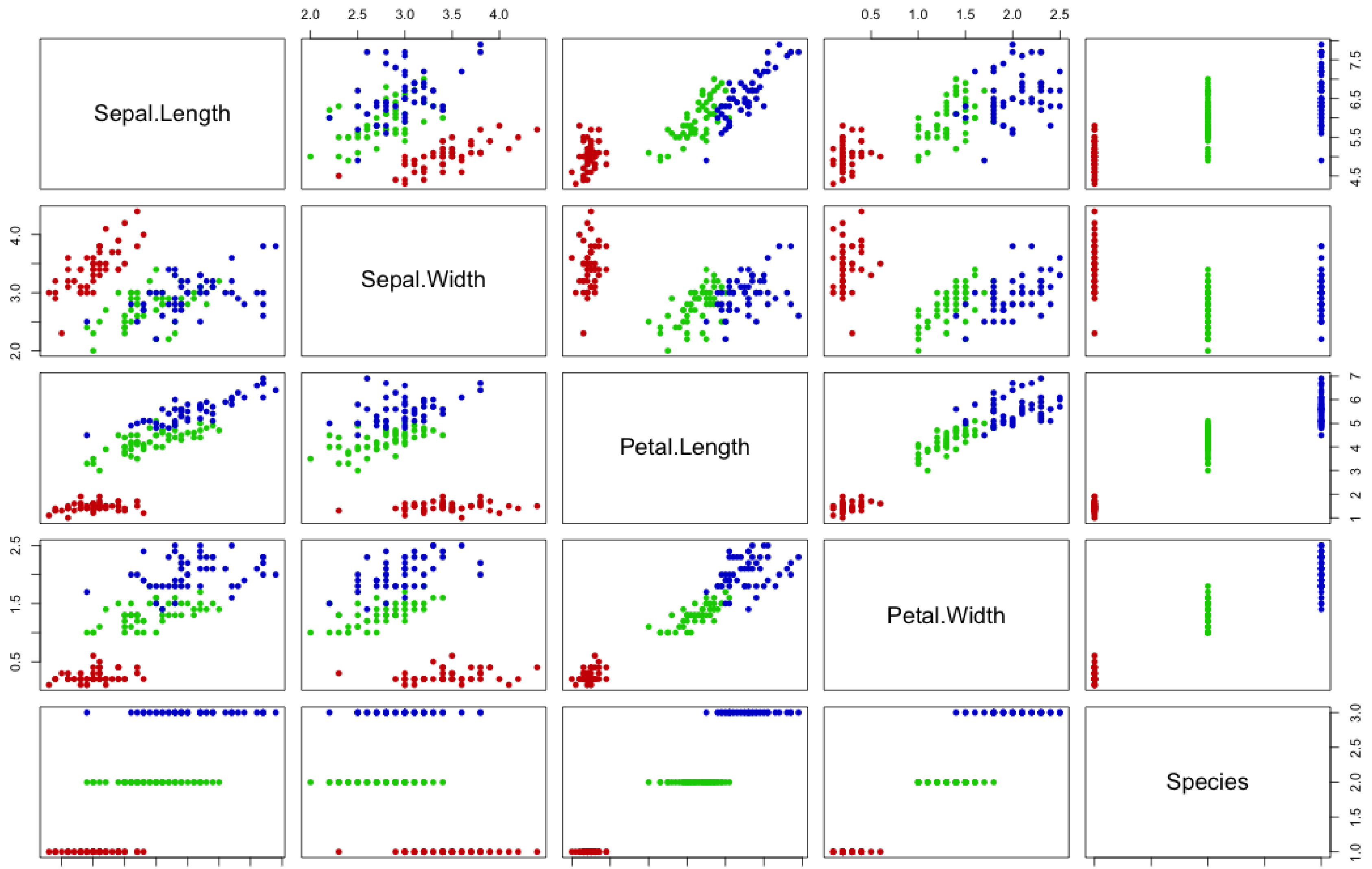
We can plot an overview of the whole data frame:

```
data(iris)      # load the data set
plot(iris)      # plot pairwise combinations of columns
```



Data frames and Graphics

```
plot(iris,pch=16,col = rep(c("red3", "green3", "blue3"),each=50))  # add
color to make plot more clear
```

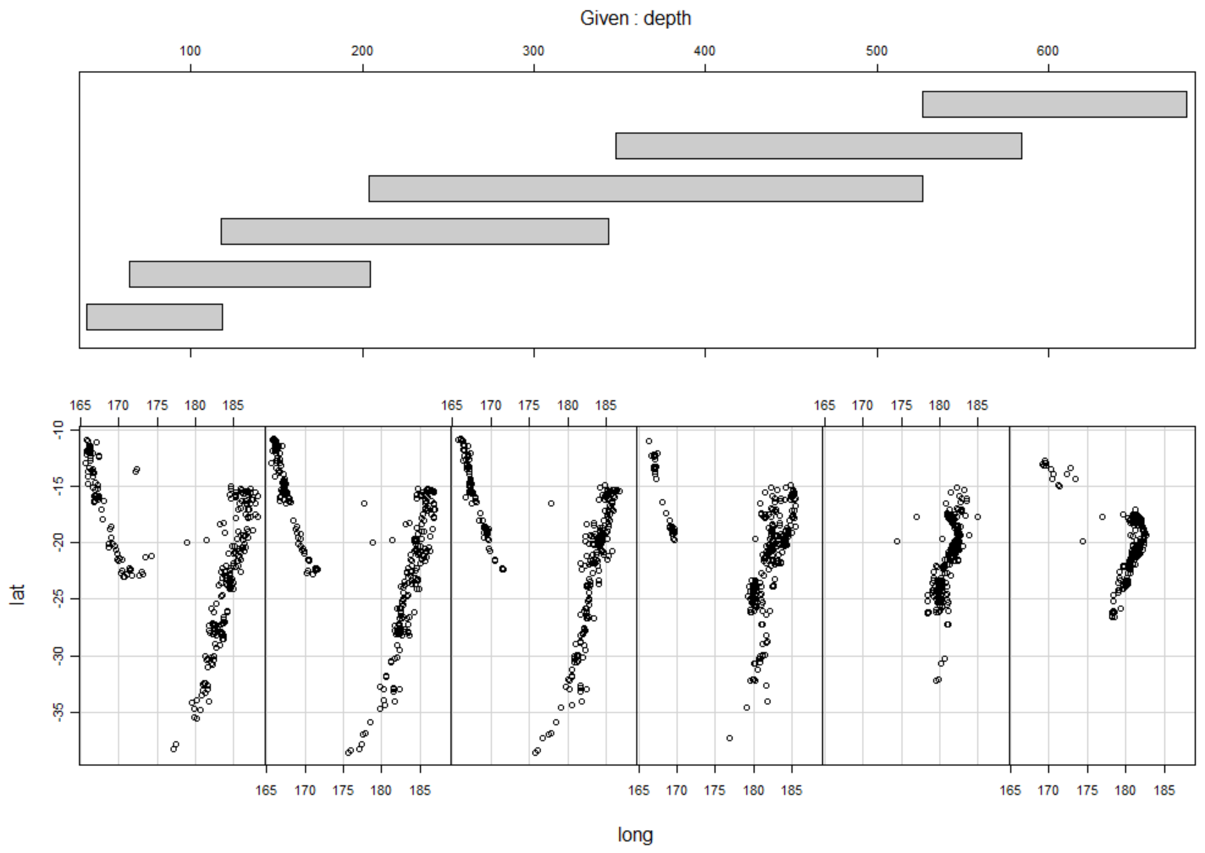


Conditioned plots

R allows us to easily plot one variable against another, conditioned on the value of a third variable.

```
coplot(lat ~ long | depth, data = quakes,rows=1)
```

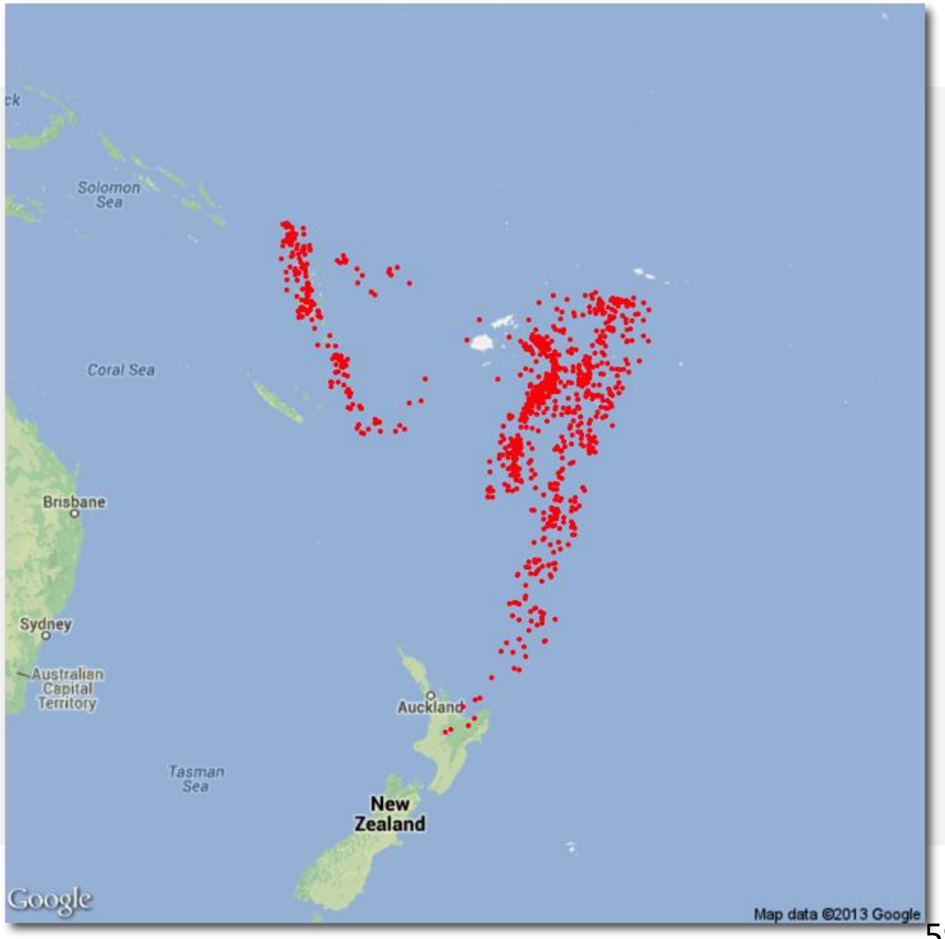
Plots location of earthquakes in the Pacific Ocean, conditioned on depth.



Graphics packages

R is a continuously growing open source project. People write their own functions and release them as packages. These packages can be extremely useful to produce sophisticated graphics, often with only a few lines of code.

```
data(quakes)
center<-c(mean(range(quakes$lat)),
mean(range(quakes$long)))
library(RgoogleMaps)

my.map<-GetMap(center=center,zoom=4,
destfile="tonga.png")

PlotOnStaticMap(my.map,lat=(quakes$lat),
lon=(quakes$long),pch=16,cex=0.5,
col="red")
```



R Color Brewer

R-Package that is helpful for creating color palettes.

A list of predefined colorsets:

```
“Blues","BuGn","BuPu","GnBu","Greens","Greys","Oranges","OrRd",
"PuBu","PuBuGn","PuRd","Purples","RdPu","Reds","YlGn","YlGnBu","YlOrBr","
YlOrRd",”BrBG","PiYG","PRGn","PuOr","RdBu","RdGy","RdYlBu","RdYlGn","Spec
tral","Accent","Dark2","Paired","Pastel1","Pastel2","Set1","Set2","Set3"
```

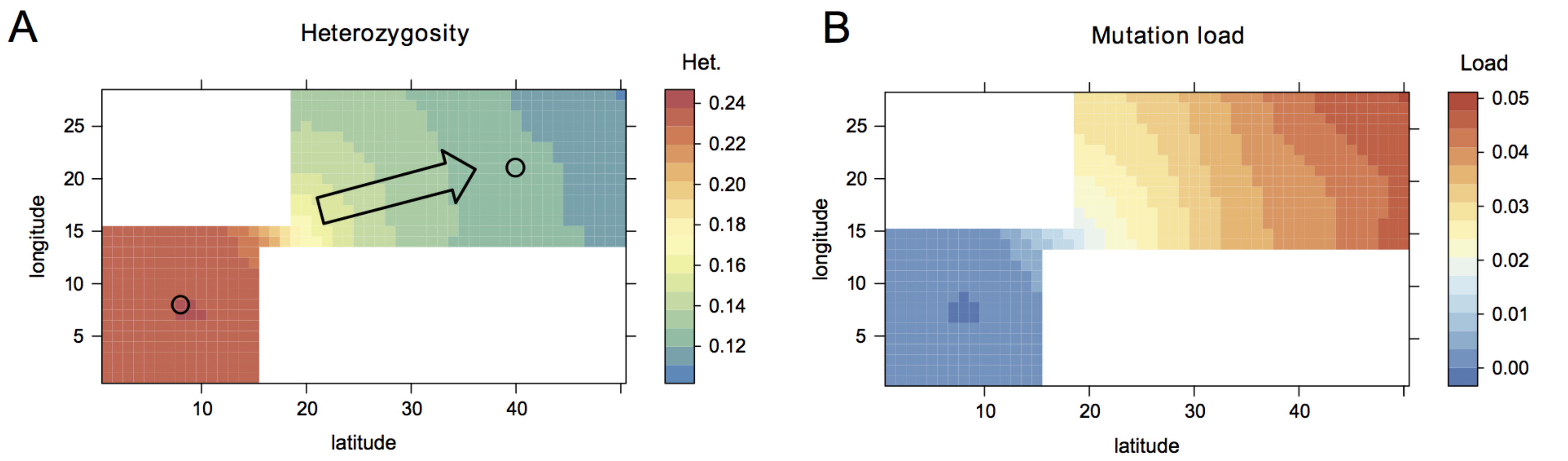
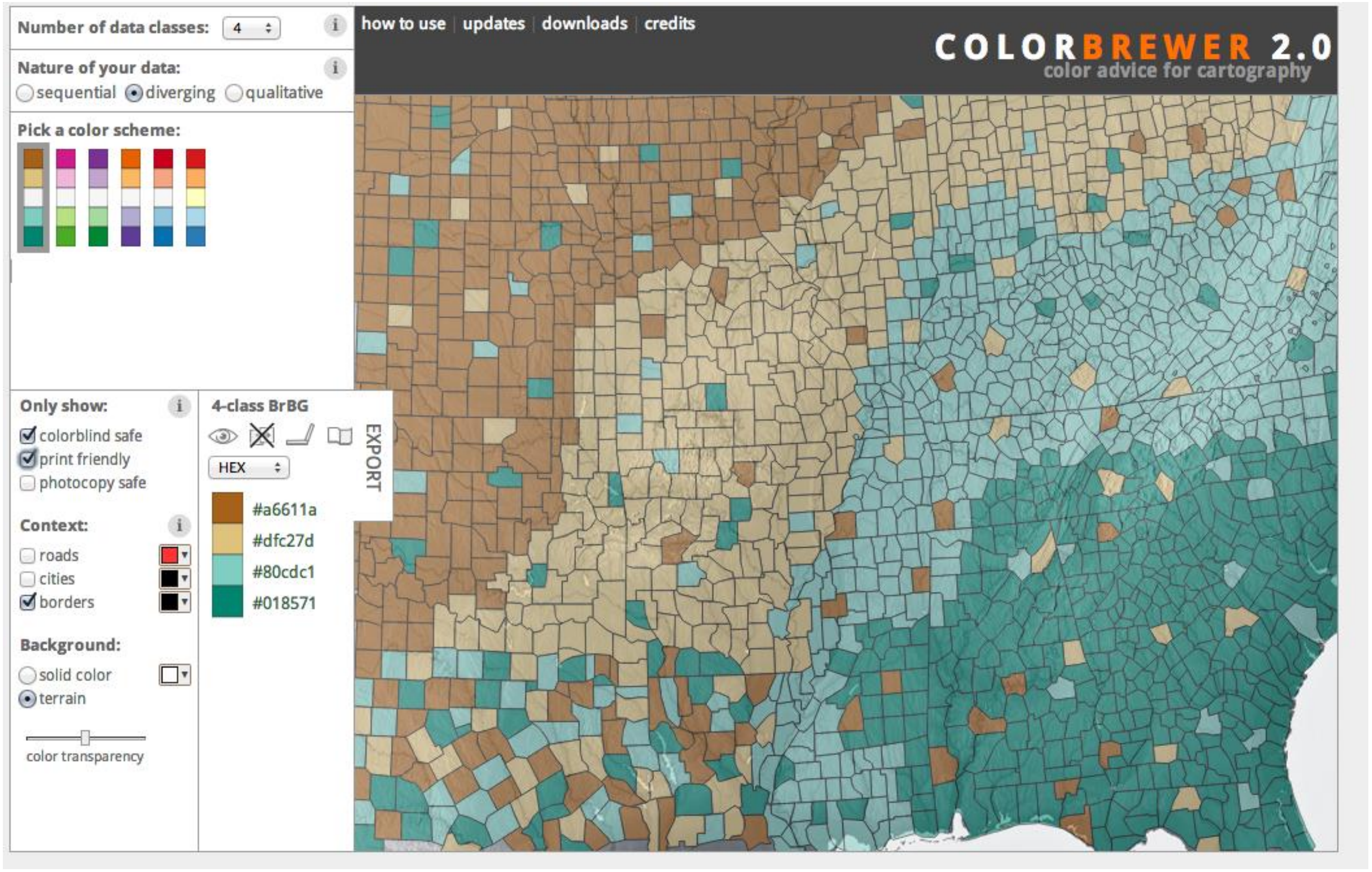


Image taken from Sousa, Peischl and Excoffier 2014 (Current Opinions in Gen. and Dev.)

R Color Brewer

You can also use the webpage to create color-sets:
http://colorbrewer2.org/


how to use | updates | downloads | credits
COLORBREWER 2.0
color advice for cartography
EXPORT
color transparency

Number of data classes: 4

Nature of your data:
sequential [✓ diverging] qualitative

Pick a color scheme:

Only show:
✓ colorblind safe
✓ print friendly
photocopy safe

Context:
roads
cities
✓ borders

Background:
solid color
[✓ terrain]

4-class BrBG

HEX

🟫	#a6611a
🟨	#dfc27d
🟦	#80cdc1
🟩	#018571

R Color Brewer Example

```
### Load the package or install if not present
if (!require("RColorBrewer")) {
  install.packages("RColorBrewer")
  library(RColorBrewer)
}

### Show all the colour schemes available
display.brewer.all()

### Generate random data matrix
rand.data <- replicate(8,rnorm(100,100,sd=1.5))

### Draw a box plot, with each box coloured by the 'BrBG' palette
boxplot(rand.data,col=brewer.pal(8,"BrBG"))

### Draw a box plot, with each box coloured by the 'Spectral' palette
boxplot(rand.data,col=brewer.pal(8,"Spectral"))
```