Practicals Bioinformatics 2010-2011

Olivier Stern olivier.stern@ulg.ac.be

Tom Cattaert tom.cattaert@ulg.ac.be

4 October 2010: Introduction to R

Obtaining R

- R is a free, open-source statistical computing language and environment available for Windows, Max OS X, Linux and Unix and licenced under the terms of the GNU General Public Licence
- R is available from the Comprehensive R Archive Network (CRAN) website http://cran.r-project.org
- Download the pre-compiled binary distribution that is suitable to your operating system
- Working with R is facilitated by using a graphical user interface (GUI), possibilities include:
  - Tinn-R: http://sourceforge.net/projects/tinn-r/
  - JGR: http://www.rforge.net/JGR/

Running R commands

- Write commands directly at the prompt

```
> print("hello world")
[1] "hello world"
```

- Source a text file in which each line represents a command. For example hello_world.r might contain the text print(“hello world”)

```
> source("hello_world.r")
[1] "hello world"
```

- Run the text file at Unix or DOS prompt

```
$ R CMD BATCH hello_world.r hello_world.out
```

This will create a file hello_world.out containing the output

Workspace

- Check current working directory

```
> getwd()
[1] "D:/courses/Bioinformatics"
```

- Change current working directory

```
setwd("D:/courses/Bioinformatics")
```

- When specifying paths always use slashes (/), not backslashes (\), even under Windows, as the backslash is used as escape character
- It is possible to save the workspace to a file called .RData in your working directory at any time

```
> save.image()
```

Workspace continued

- When you exit R you will also be asked whether to save the image
- At the start of your next R session, this workspace can be loaded

```
> load("D:/courses/Bioinformatics/.RData")
```

- You can obtain a list of the variables stored in the workspace by

```
> ls()
character(0)
```

- At this moment, the workspace is empty
- You can always clear the entire workspace using

```
> rm(list=ls())
```

- Also individuals objects can be deleted using the rm() function

Arithmetic operations and numerical functions

- One of the simplest possible tasks in R is to enter an arithmetic expression and receive the result, e.g. the addition of two numbers

```
> 2+2
[1] 4
```

- Other arithmetic operations include - (subtract, sign), * (multiply), / (divide), ^ (raise to power), %/% (integer divide) and %% (remainder)
- R can do other standard calculations such as exponential

```
> exp(-2)
[1] 0.1353353
```

- Other available mathematical functions include log(x) (natural logarithm), log10(x) (base-10 logarithm), sin(x), cos(x), tan(x)

Simulating

- Simulations can only be replicated when setting a seed

```
> set.seed(1980)
```

- To obtain 5 random heights from a normal distribution with mean 1.70 and standard deviation 0.05

```
> rnorm(5, 1.70, 0.10)
[1] 1.821483 1.619895 1.615578 1.753910 1.650765
```

- To obtain 5 random minor allele frequencies from a uniform distribution with minimum 0.05 and maximum 0.5

```
> runif(5, 0.05, 0.5)
[1] 0.28514444 0.05171292 0.30269908 0.32165110 0.14621970
```

Assignment

- Suppose you want to create a new variable x equaling 1, you type

```
x <- 1
```

- Variables names can be built from letter, digits, and the period (dot) symbol, but should start with a letter
- The resulting object can be printed using

```
> print(x)
[1] 1
```

- Or even simpler by typing

```
> x
[1] 1
```

Classes

- Each object has an associated class
- The variable that we just created is a numeric variable

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

- A character variable is created using quotation marks

```
> y <- "Gene1"
> class(y)
[1] "character"
```

- Other object classes include factors, matrices and data frames
- The result of applying generic R functions such as print() and plot() depends on the class associated with the arguments to them

Vectors

- A numeric vector is an array of numbers and can be created as

```
> y <- c(1,0)
> y
[1] 1 0
```

- A character vector is an array of character variables

```
> z <- c("Female", "Male", "Female", "Female", "Male")
> z
[1] "Female" "Male"   "Female" "Female" "Male"
```

- Logical vectors can take the values TRUE, FALSE (or NA)

```
> z=="Female"
[1]  TRUE FALSE  TRUE  TRUE FALSE
```

Vectorized arhitmetic

- You can do calculations with numeric vectors just like ordinary numbers, supposing they have the same length

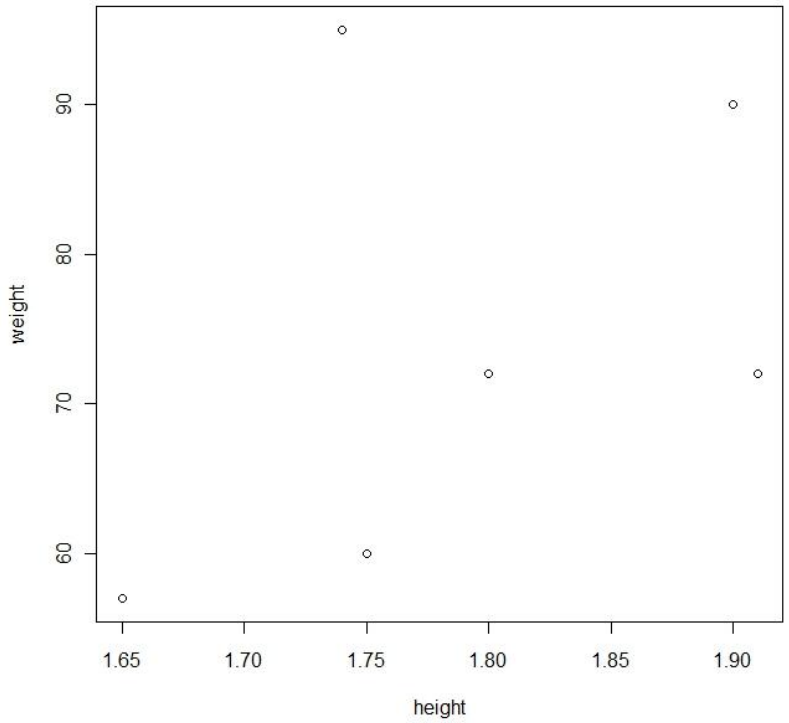
```
> # Vectorized arithmetic
> weight <- c(60, 72, 57, 90, 95, 72)
> height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
> bmi <- weight/height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

- The operation is carried out elementwise
- If the vectors have different lengths, the shorter vector is recycled giving a warning message if longer vector is not a multiple in length

Graphics

- To plot the relation between weight and height, type

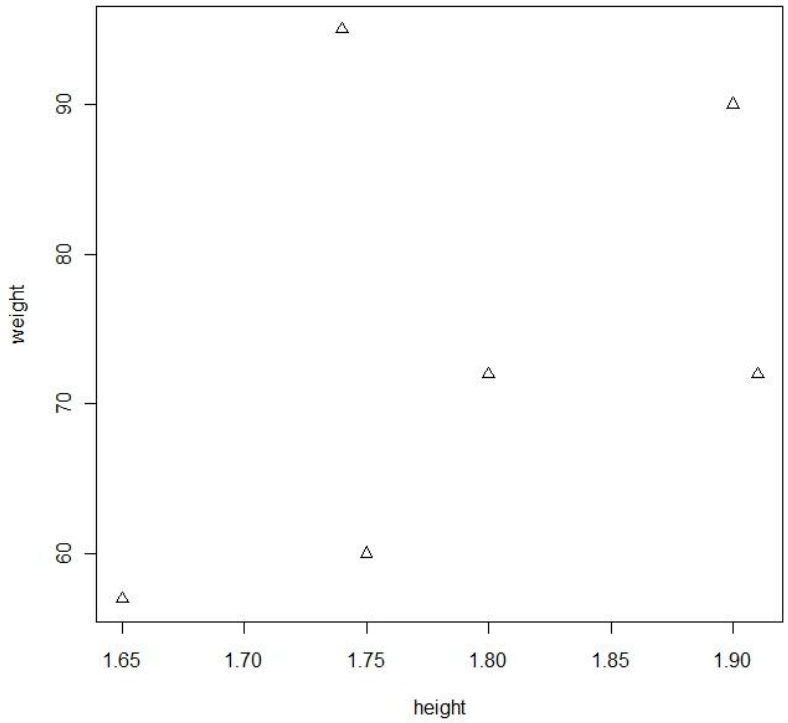
> plot(height, weight)



Graphics continued

- To change the plotting character use the keyword pch

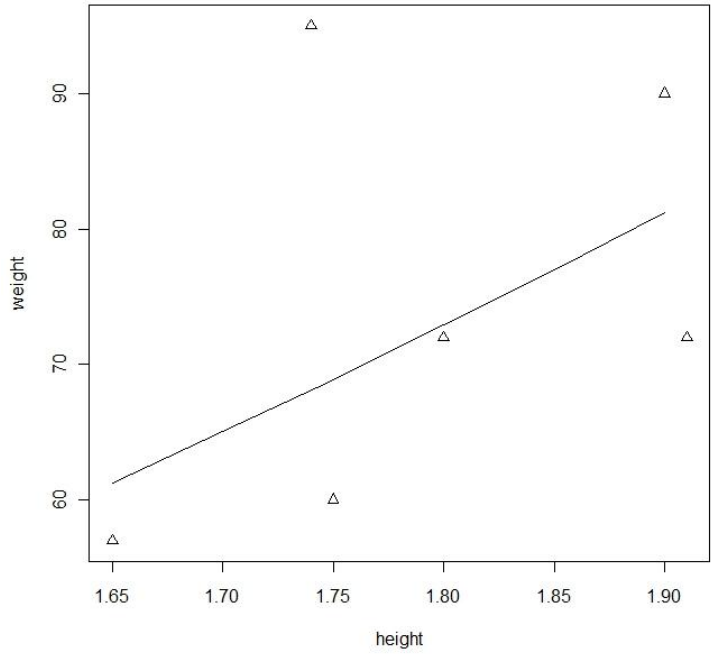
```
> plot(height, weight, pch=2)
```



Graphics continued

- You can superimpose a curve of expected weights at BMI 22.5

```
> hh <- c(1.65, 1.70, 1.75, 1.80, 1.85, 1.90)
> lines(hh, 22.5*hh^2)
```



Functions

- When you write `plot(height, weight, pch=2)`, R assumes that the first argument corresponds to the x-variable and the second to the y-variable, this is called positional matching
- For the third argument, the keyword `pch` is used, this is called named matching
- Brackets are also needed when calling a function without arguments, e.g. to display the content of the workspace you type

```
> ls()
[1] "bmi"    "height" "hh"     "weight" "x"      "y"      "z"
```

- But if you type `ls` without brakcets, R will display the piece of code underlying the `ls()` function

Vector functions

- Scalar functions also work on vectors, as we saw with height^2
- Some functions are only useful for vectors, e.g.

```
> min(bmi)
[1] 19.59184
```

- Other such functions include max(x), range(x) (range c(min(x), max(x))), and length(x) (number of elements in x)
- And of course statistical functions such as

```
> mean(bmi)
[1] 23.13262
```

- More statistical functions include sd(x) (standard deviation), var(x) (variance), and median(x)

Concatening vectors

- c() can also concatenate vectors of more than one element

```
> c(y, 0)
[1] 1 0 0
```

- If you concatenate vectors of different types, they will be converted to the least restrictive type (so that all elements have the same)

```
> c(y, z)
[1] "1"      "0"      "Female" "Male"   "Female" "Female" "Male"
> c(y, TRUE)
[1] 1 0 1
> c("Female",FALSE)
[1] "Female" "FALSE"
```

Creating sequences

- seq() is used to create equidistant series of numbers

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

- In our graphics example we could have more easily written

```
> hh <- seq(1.65, 1.90, 0.05)
```

- Special syntax for case with stepsize 1

```
> 4:9
[1] 4 5 6 7 8 9
```

Replicate vectors

- rep() is used to replicate vectors

```
> oops <- 0:2
> rep(oops, 3)
[1] 0 1 2 0 1 2 0 1 2
> rep(oops, 1:3)
[1] 0 1 1 2 2 2
> rep(oops, each=3)
[1] 0 0 0 1 1 1 2 2 2
```

Simulating a die

- If you want to simulate throwing a die 6 times, you can write

```
> sample(1:6,6)
[1] 6 5 1 2 4 3
```

- However, the default is sampling without replacement, so we need to correct this formula

```
> sample(1:6,6,replace=TRUE)
[1] 5 4 2 1 4 1
```

- Now suppose our die is a false one and has 50% chance to throw a 6, we can achieve this as follows

```
> sample(1:6,6,replace=TRUE,prob=c(rep(0.1,5), 0.5))
[1] 3 4 6 2 6 6
```

Factors

- A factor object is a type of vector with an associated attribute that describes the levels of the variable
- You can generate a factor object as follows

```
> x1 <- factor(c("AA", "AT", "TT", "TT", "AT", "AT", "AA", "TT"))
> x1
[1] AA AT TT TT AT AT AA TT
Levels: AA AT TT
```

- There are three levels attributed to this variable, AA, AT and TT
- These levels can also be obtained through

```
> levels(x1)
[1] "AA" "AT" "TT"
```

Matrices

- A matrix is a two-dimensional array of objects
- It can be created using the matrix() function

```
> X <- matrix(c("AA", "AT", "GG", "GG"), nrow=2)
> X
     x1   x2
[1,] "AA" "GG"
[2,] "AT" "GG"
```

- Alternatively, the cbind() or rbind() functions can be used

```
> x1 <- c("AA", "AT")
> x2 <- c("GG", "GG")
> X <- cbind(x1, x2)
```

Matrix functions

- A matrix can be transposed using

```
> t(X)
   [,1] [,2]
x1 "AA" "AT"
x2 "GG" "GG"
```

- This matrix could also have been created directly by

```
> matrix(c("AA", "AT", "GG", "GG"), nrow=2, byrow=TRUE)
```

- Dimensions and row / column names can be obtained

```
> dim(X)
[1] 2 2
> colnames(X)
[1] "x1" "x2"
```

Attributes

- The characteristics of the matrix can be printed

```
> attributes(X)
$dim
[1] 2 2
$dimnames
$dimnames[[1]]
NULL
$dimnames[[2]]
[1] "x1" "x2"
```

- The attributes can be printed separately

```
> attributes(X)$dim
[1] 2 2
```

Numerical matrix functions

- Create a numerical matrix

```
> A <- matrix(1:4, nrow=2)
```

- The inverse matrix is obtained as

```
> solve(A)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

- Other numerical matrix functions include `det(A)` (determinant), `sum(diag(A))` (trace) and matrix multiplication `A %*% B`

Lists

- A list can contain multiple objects of different types including vectors, matrices and other lists
- The dimnames attribute of X is an example of a list
- A list is created as follows

```
> list(trait=y, genotypes=X)
$trait
[1] 1 0
$genotypes
    x1   x2
[1,] "AA" "GG"
[2,] "AT" "GG"
```

Data frames

- Data frames are similar to matrices but can contain columns of different variable types including numeric, character and factor

```
> ExampleData <- data.frame(ID = c(1, 2, 3, 4, 5),
+                           SNP = c("AA", "AT", "TT", "TT", "AA"),
+                           Gender = c("Female", "Male", "Female", "Female", "Male"),
+                           DiseaseStatus = c(1, 1, 0, 0, 0))
> ExampleData
  ID SNP Gender DiseaseStatus
1  1  AA Female            1
2  2  AT   Male            1
3  3  TT Female            0
4  4  TT Female            0
5  5  AA   Male            0
```

Importing data

- Data can also be imported from text files
- Suppose the data ExampleData are in a file ExampleData.txt, then we can read them by

```
> ExampleData <- read.table("ExampleData.txt", header=TRUE, sep="\t")
```

- header=TRUE indicates that the first line of the file contains variable names, while header=FALSE indicates it contains the first record
- sep="\t" indicates the data tab-delimited, while sep="," indicates comma-separated and sep="" separation by one or more spaces
- You can specify dec="," or dec=".", to set the decimal point
- You can specify which strings represent missing values, e.g. if you want the period to represent a missing value you put na.strings="."

Importing data continued

- Non-numeric (textual) input is converted to a factor, with level ordered alphabetically, you can recode factors but the details of this are tricky and not given here
- If needed, this can be modified on all columns using `stringsAsFactors==FALSE` or on a per-item basis using `as.is`
- Automatic conversion is often convenient but is also inefficient in terms of computer time and storage, as a numeric columns is first read as character data, whereafter it checks that all elements can be converted to numeric
- You can bypass the mechanism by explicitly specifying the column classes using the `colClasses` argument (e.g. `"character"`)

Attaching data

- One can access each variable of the data frame as follows

```
> ExampleData$ID
[1] 1 2 3 4 5
> ExampleData$SNP
[1] AA AT TT TT AA
Levels: AA AT TT
```

- Alternatively, the data frame can be attached

```
> attach(ExampleData)
```

- This allows us to use the shorthand notation

```
> ID
[1] 1 2 3 4 5
```

Exploring data

- First we assess the class of the data object

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

- The names of all the variables in a data frame can be listed

```
> names(ExampleData)
[1] "ID"           "SNP"          "Gender"       "DiseaseStatus"
```

- The dimensions of the data frame can be assessed as well

```
> dim(ExampleData)
[1] 5 4
```

- This means we have 5 rows (individuals) and 4 columns (variables)

Tabulating

- We can tabulate the number of males and females

```
> table(ExampleData$Gender)
Female   Male
     3      2
```

- The frequencies of males and females can be obtained by

```
> table(ExampleData$Gender)/5
Female   Male
   0.6    0.4
```

- Alternatives producing the same output are

```
> table(ExampleData$Gender)/dim(ExampleData)[1]
> table(ExampleData$Gender)/sum(table(ExampleData$Gender))
```

Converting

- Gender is coded as a factor variable

```
> class(ExampleData$Gender)
[1] "factor"
```

- You may want to create a numerical version of this variable

```
> GenderNum <- as.numeric(ExampleData$Gender)
> GenderNum
[1] 1 2 1 1 2
```

- You may want to do this within the data frame

```
> within(ExampleData,{
+   GenderNum <- as.numeric(ExampleData$Gender)
+   rm(Gender)
+ })
```

Subsetting

- Suppose you want to print out the first row of our data

```
> ExampleData[1,]
  ID SNP Gender DiseaseStatus
1  1  AA Female             1
```

- Or you might print the third column

```
> ExampleData[,3]
[1] Female Male   Female Female Male
Levels: Female Male
```

- Lists are subsetted by double [[]]

```
> list(trait=y, genotypes=X)[[1]]
[1] 1 0
```

Subsetting continued

- Multiple rows and columns can be printed by replacing the scalar with a sequence of numbers

```
> ExampleData[c(2,4),1:3]
  ID SNP Gender
2  2  AT   Male
4  4  TT Female
```

- Negative subsetting is also possible

```
> ExampleData[-c(2,4,5),1:3]
  ID SNP Gender
1  1  AA Female
3  3  TT Female
```

- But positive and negative subsetting cannot be combined

Subsetting continued

- One can also indicate rows by specifying the level of a variable

```
> ExampleData[ExampleData$Gender=="Male",]
  ID SNP Gender DiseaseStatus
2  2  AT   Male             1
5  5  AA   Male             0
```

- Available comparison operators are < (less than), > (greater than), == (equal to), <= (lt or eq), >= (gt or eq), and != (not equal to)
- Logical expressions can be combined using (element-wise) logical operators & (logical AND), | (logical OR), and ! (logical NOT)

```
> ExampleData[ExampleData$Gender=="Male" & ExampleData$DiseaseStatus>0,]
  ID SNP Gender DiseaseStatus
2  2  AT   Male             1
```

Subsetting continued

- There is an elegant approach using the subset() function

```
> subset(ExampleData, Gender=="Female")
  ID SNP Gender DiseaseStatus
1  1  AA Female             1
3  3  TT Female             0
4  4  TT Female             0
```

- This can also be used to extract variables from the data frame

```
> subset(ExampleData, Gender=="Female", select=c("ID", "SNP"))
  ID SNP
1  1  AA
3  3  TT
4  4  TT
```

Subsetting continued

- If missing values appear in an indexing vector, then R will create the corresponding elements in the result but set the values to NA
- is.na() returns TRUE when the corresponding value is missing

```
> boo <- c(0,1,NA)
> is.na(boo)
[1] FALSE FALSE  TRUE
```

- This is really needed because it is impossible to make comparisons of the form ==NA as the comparison with an unknown is unknown

```
> boo==NA
[1] NA NA NA
```

Stratified tabulating

- Tabulate genotypes for individuals with and without disease

```
> table(ExampleData$SNP[ExampleData$DiseaseStatus==1])

AA AT TT

 1  1  0

> table(ExampleData$SNP[ExampleData$DiseaseStatus==0])

AA AT TT

 1   0  2
```

- Alternatively, the table() function also does cross-tabulation

```
> table(ExampleData$SNP, ExampleData$DiseaseStatus)

    0 1

 AA 1 1

 AT 0 1

 TT 2 0
```

Implicit loops

- tapply() can be used to calculate stratified tables

```
> tapply(ExampleData$SNP, ExampleData$DiseaseStatus, table)
$`0`
AA AT TT
 1  0  2
$`1`
AA AT TT
 1  1  0
```

- The result is a list with the number of elements equal to the number of levels of ExampleData$DiseaseStatus and each element representing the result of applying the table() function to the corresponding subset of the data

Implicit loops continued

- sapply() is used to apply a function to all variables in a data frame

```
> sapply(ExampleData, mean)
        ID        SNP     Gender DiseaseStatus
       3.0         NA         NA        0.4
```

- apply() allows to apply a function to the rows / columns of a matrix

```
> m <- matrix(1:6, nrow=2, byrow=TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> apply(m, 2, sum)
[1] 5 7 9
```

Implicit loops continued

- Sometimes you want to repeat something a number of times but still collect the results as a vector
- This makes sense only when the repeated calculations give different results, typically in simulations
- This can be done using the replicate() function

```
> set.seed(2006)
> replicate(5, mean(rnorm(100, 1.70, 0.10)))
[1] 1.705472 1.701831 1.699613 1.705690 1.697468
> replicate(5, min(runif(100, 0.05, 0.5)))
[1] 0.06393590 0.05327317 0.05148804 0.05007529 0.05023605
```

Sorting

- Vectors can be sorted as follows

```
> sort(ExampleData$Gender)
[1] Female Female Female Male   Male
Levels: Female Male
```

- Further variables can be specified to use when undecided from first
- You can obtain the ordering used in the sorting by

```
> order(ExampleData$Gender)
[1] 1 3 4 2 5
```

- This can be used to sort other variables by the same criterion

```
> ExampleData$SNP[order(ExampleData$Gender)]
[1] AA TT TT AT AA
Levels: AA AT TT
```

Plot layout

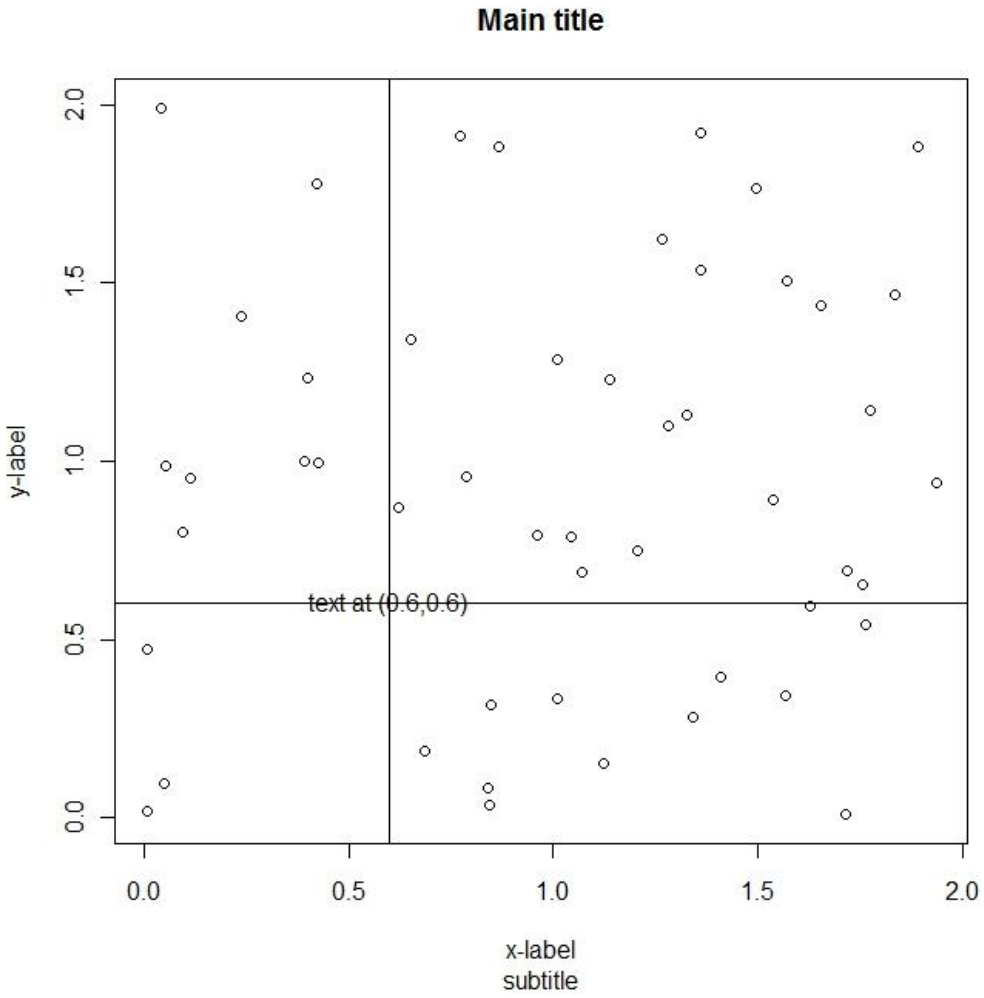
- A standard x-y plot has an x and a y title label generated from the expressions being plotted
- You may however override these labels and also give a main title above the plot and a subtitle at the bottom

```
> set.seed(2006)
> x <- runif(50, 0, 2)
> y <- runif(50, 0, 2)
> plot(x, y, main="Main title", sub="subtitle", xlab="x-label", ylab="y-label")
```

- You can place additional points and lines using points() and lines(), add text using text(), and straight lines using abline()

```
> text(0.6, 0.6, "text at (0.6,0.6)")
> abline(h=0.6, v=0.6)
```

Plot layout continued



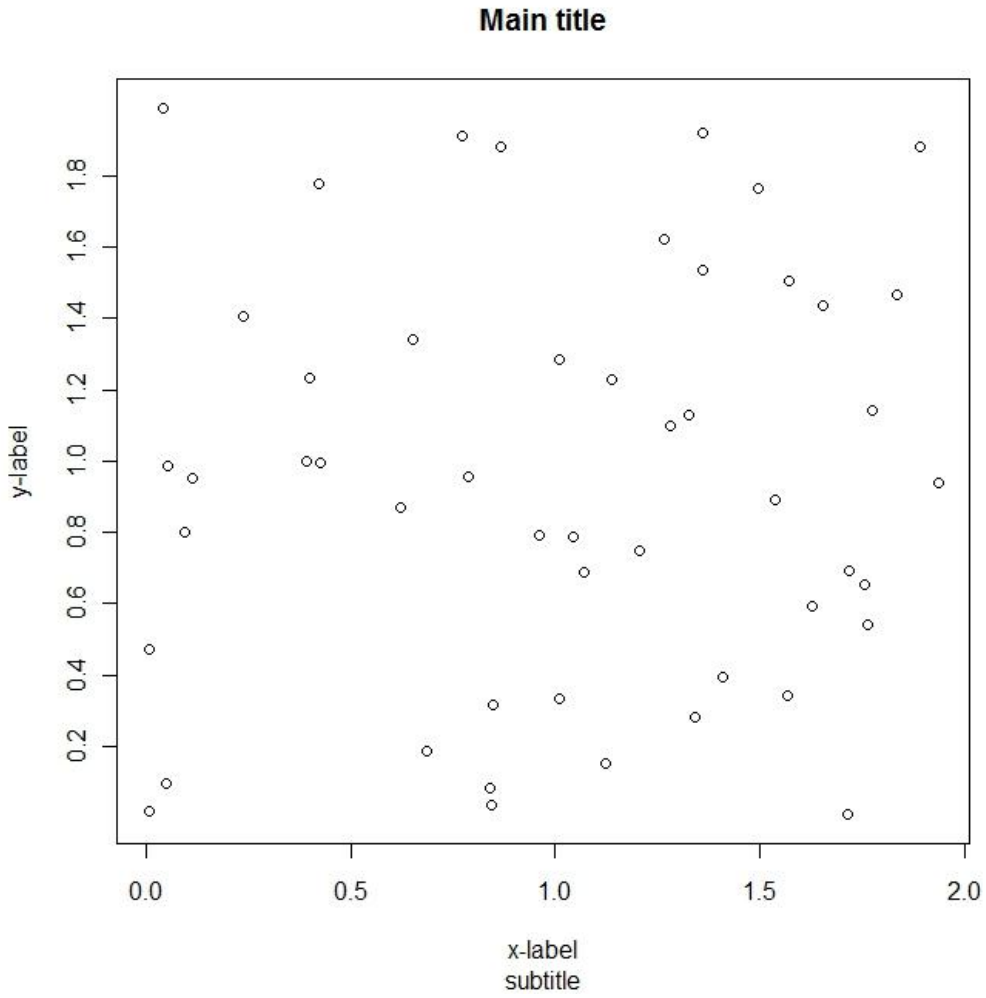
Building plots in pieces

- Plot are composed of elements, which can be drawn separately
- Separate drawing commands often allow finer control
- A standard strategy is to draw a plot first without a certain element
- In the extreme, one first plots absolutely nothing

```
> plot(x, y, type="n", xlab="", ylab="", axes=FALSE)
> points(x, y)
> axis(1)
> axis(2, at=seq(0.2, 1.8, 0.2))
> box()
> title(main="Main title", sub="subtitle", xlab="x-label", ylab="y-label")
```

- The second axis() call specifies an alternative set of tick marks
- Similarly one creates non-equidistant axes and non-numeric labels

Building plots in pieces continued



The par() function

- Allows incredibly fine control over the details of a plot
- But can be quite confusing
- Learn by solving problems one encounters
- Some parameters can also be set by arguments to plotting functions, the difference being that when set with par() it will stay set subsequently
- Aspects controlled include line width (lwd) and type (lty), character size (cex) and font (font), colour (col), style of axis calculation (xaxs), plot limis (xlim), margin sizes (mar) and line spacing in margins (mex)
- Using the mfrow and mfcol parameters it is also possible to divide a figure in several subfigures

Combining plots

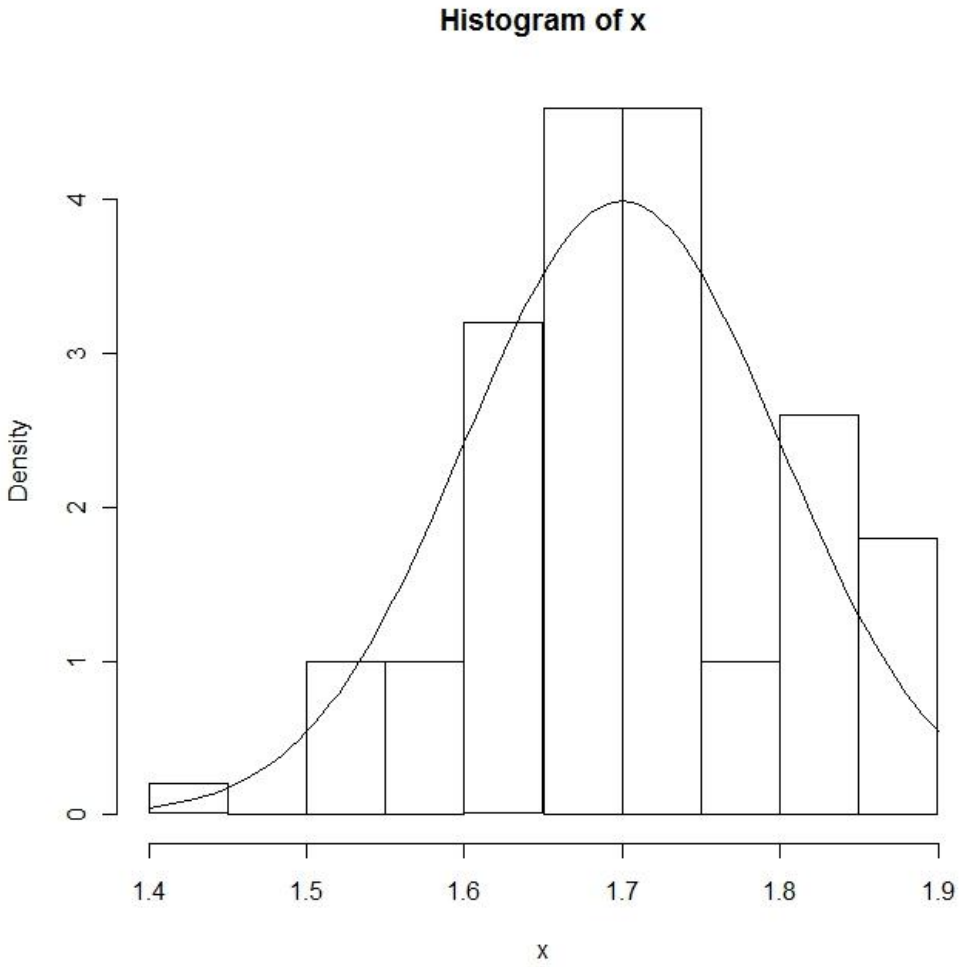
- We want to overlay a histogram with a normal density

```
> set.seed(2006)
> x <- rnorm(100, 1.70, 0.10)
> hist(x, freq=FALSE)
> curve(dnorm(x, 1.70, 0.10), add=TRUE)
```

- It is not assured that histogram and density have same range so the density curve (or the histogram if reversed) might be chopped off

```
> histo <- hist(x, plot=FALSE)
> ylimit <- range(0, histo$density, dnorm(1.70))
> hist(x, freq=FALSE, ylim=ylimit)
> curve(dnorm(x, 1.70, 0.10), add=TRUE)
```

Combining plots continued



Writing functions

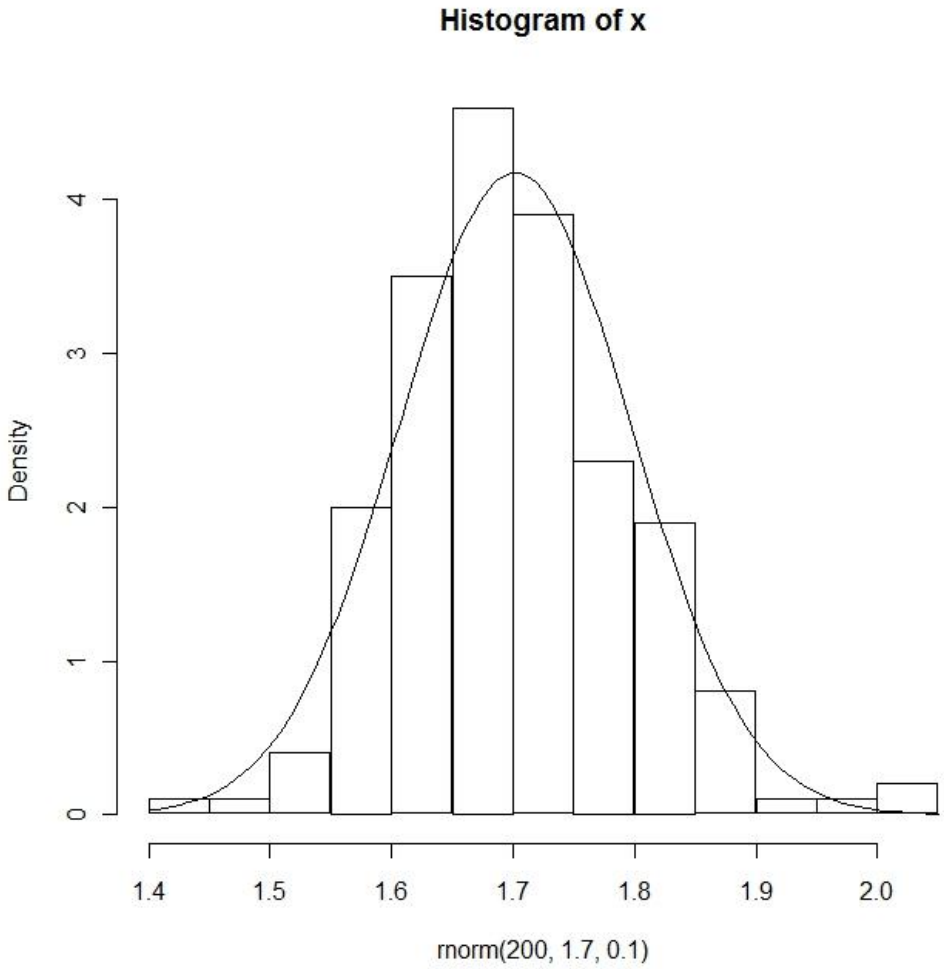
- Suppose we want to make a function for the plotting task

```
> hist.with.normal <- function(x, xlabel=deparse(substitute(x)), ...)
+ {
+   histo <- hist(x, plot=FALSE, ...)
+   sdev <- sd(x)
+   m <- mean(x)
+   ylimit <- range(0, histo$density, dnorm(0, sdev))
+   hist(x, freq=FALSE, ylim=ylimit, xlab=xlabel)
+   curve(dnorm(x, m, sdev), add=TRUE)
+ }
```

- This can then be called as follows

```
> hist.with.normal(rnorm(200, 1.70, 0.10))
```

Writing functions continued



Writing functions continued

- The following function calculates the unique square root of a symmetric positive definite matrix

```
> sqrt.mat = function(Mat){
+ e = eigen(Mat, symmetric=TRUE)
+ sqrt.Mat = e$vectors %*% diag(sqrt(e$values)) %*% t(e$vectors)
+ return(sqrt.Mat)
+ }
```

- We can test it with an example

```
> A <- matrix(c(1, 0.5, 0.5, 1/3), nrow=2)
> sqrt.mat(A)
          [,1]      [,2]
[1,] 0.9322858 0.3617226
[2,] 0.3617226 0.4499890
```

Flow control

- We want to find the square root of x using Newton's method

```
> y <- 12345
> x <- y/2
> while(abs(x^2-y) > 1e-10) x <- (x+y/x)/2
> x
[1] 111.1081
> x^2
[1] 12345
```

- The while(condition) expression construction says that the expression should be evaluated as long as the condition is TRUE
- The test occurs at the top of the loop

Flow control continued

- Alternatively the repeat construction with test at the bottom is used

```
> repeat{
+   x <- (x+y/x)/2
+   if (abs(x^2-y) < 1e-10) break
+ }
> x
[1] 111.1081
> x^2
[1] 12345
```

- The example also illustrates compound expressions (several expression between curly braces), the if construction for conditional execution, and the break expression causing exit of enclosing loop

Flow control continued

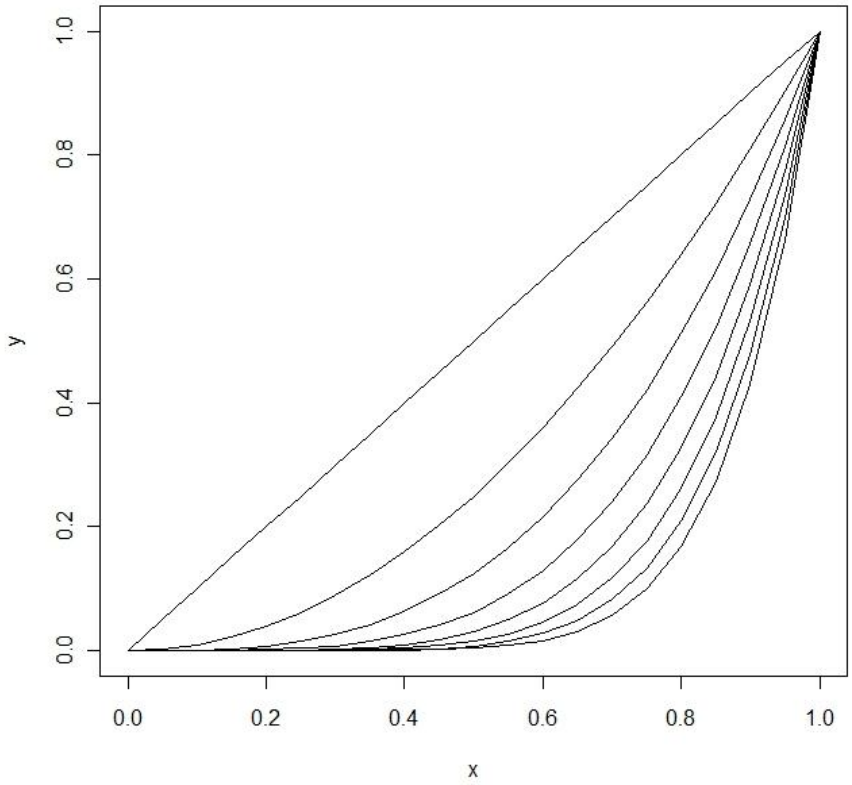
- An alternative execution might be specified after `else` keyword
- In flow control one uses `&&` and `||` for logical AND and OR and the expression on the right-hand side is only evaluated when needed
- The loop could now allow for y being a vector simply by changing the termination condition

```
if (all(abs(x^2-y) < 1e-10)) break
```

- The most frequently used looping construct is the `for` loop, which loops over a fixed set of values

```
> x <- seq(0, 1, 0.05)
> plot(x, x, ylab="y", type="l")
> for (j in 2:8) lines(x, x^j)
```

Flow control continued



Flow control continued

- Loops are not always needed

```
> x <- seq(1, 100, 3)
> y <- rep(0, length(x))
> for (i in 1:length(x)) {
+   y[i] <- x[i]^2
+ }
> y[1:5]
[1]   1  16  49 100 169
```

- The same can be done simply as `y <- x^2` as R works with vectors
- Loops can be slow so avoid them if possible
- Other ways to avoid loops are the `apply()` function and its derivates

Statistical distributions

- We already encountered rnorm(n) to generate random normal variables, and dnorm(x) to obtain the normal density function
- There exist further pnorm(x) giving the cumulative distribution function P(X≤x), and qnorm(p) giving the p-quantile, i.e. x : P(X≤x)=p
- Available distributions include pnorm(x,mean,sd) (normal), pt(x,df) (Student's t), pf(x,n1,n2) (F), pchisq(x,df) (chi-squared), pbinom(x,n,p) (binomial), ppois(x,λ) (Poisson), and punif(x,min,max) (uniform)
- E.g. to obtain the significance threshold at the 0.05 significance level for a chi-squared distribution with 2 df

```
> qchisq(0.05, df=2, lower.tail=FALSE)
[1] 5.991465
```

Installing R packages

- Suppose you want to install the genetics package

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

- Select a CRAN mirror in a location near you to download the package and the package will be installed automatically
- To assess the functions within the packages you need to type

```
> library(genetics)
```

- This needs to be repeated at the start of each new R session
- Bioconductor distributes several R packages targeted for the analysis of genomic data

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()
```

Getting help

- Suppose you want more information on the read.table() function

```
> help(read.table)
```

- The associated help file includes
  - A general description of the function and how to use it
  - Details on arguments to be entered and values returned
  - A simple example illustrating application of the function
- If you do not know the name of the function type

```
> help.search("read")
```

- This yields a list of functions and associated packages
  - A very useful online resource is Quick-R: http://www.statmethods.net/