



First steps in R

Florian Markowetz, Tim Beissbarth

Practical DNA Microarray Analysis

This tutorial refers to the practical session on day one of the course in Practical DNA Microarray Analysis: <http://compdiag.molgen.mpg.de/ngfn/>. We will guide you through your first steps in R. For further reading we recommend "An introduction to R" by Venables et al. which is available for free from <http://cran.r-project.org> or "Introductory Statistics with R" by Peter Dalgaard, Springer 2002, or "Bioinformatics and Computational Biology Solutions Using R and Bioconductor" by Gentleman, Carey, Huber, Springer 2005.

Preliminaries

From A to Ω. Invoking R depends on the platform you use. Under Windows it involves some clicking-around, under Linux/Unix you usually just type "R" at the Konsole. In our current installation at the MPI, you have to type "R-rel". You quit R with the command `q()`. Don't forget the parentheses!

Obtaining and Installing R. If you do not have R installed yet, you can obtain a copy via <http://www.r-project.org>. It is available for Windows, Mac and most Linux distributions, as well as in source code. You should also get a good text editor with syntax highlighting for R - for a list of recommendations look at the R web site under "Related Projects - R GUIs".

Installing R packages. Most packages are available either via CRAN (<http://cran.r-project.org>) and can be installed with the command `install.packages("packagename", dependencies=TRUE)` from within R or via Bioconductor (<http://www.bioconductor.org>) using the commands:

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

Other packages need can that sometimes can be obtained from the authors web sites can usually be installed using the command `R CMD INSTALL packagename.tar.gz` from the unix command line or from a zip file in the GUI menu of the windows version.

Getting help. There are many ways to get help from R. Find out what the function `library()` does by using the commands `help(library)` or `?library`. The command `library()` results in a list of R-packages that are already loaded and can be used by you. What happens if you type `library` without parentheses?

Online help. Running `help.start()` launches a web browser that allows the help pages to be browsed with hyperlinks. Spend some time getting used to it.

Vectors and assignments

Build a vector with entries 1 to 10 and call it x. Try different ways to do it:

```
> x <- 1:10
> assign("x",1:10)
> x <- seq(1,10,by=1)
> x <- seq(length=10,from=1,by=1)
> x <- c(1,2,3,4,5,6,7,8,9,10)    # c = concatenate
```

The result always is the same:

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

The operator “<-” assigns the value of an expression (e.g. “1:10”) to an object (e.g. “x”). If an expression is used as a complete command, the value is printed and lost.

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

Compare `a <- 1:10-1` to `b <- 1:(10-1)`.

With `ls()` or `objects()` you get an overview of the objects in your workspace. Single objects can be removed by `rm()`. To clear your whole workspace use `rm(list=ls())`. Remove *a* and *b*.

Let’s have a closer look at *x*. `summary()` gives you an overview of an objects properties. The output depends on what type of object it is. For vectors you get information on the distribution of values in it.

```
> summary(x)
> length(x)
> mode(x)
```

x is a numeric vector of length 10. The mode can be changed:

```
> y <- as.character(x)
> y
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
> mode(y)
[1] "character"
> x <- as.numeric(y)
```

Vector arithmetic

Build two vectors of length 5 and try some arithmetic operations like `+`, `-`, `*`, `/`, `sum`, `mean`, `^`, `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, `abs`, `max`, `min`, `range`, `prod`, `cumsum`. These operations work elementwise and can be nested.

```
> x <- 5:1
> x
[1] 5 4 3 2 1
> y <- c(4,7,3,7,6)
> y
[1] 4 7 3 7 6
> x+y
[1] 9 11 6 9 7
> x*y
[1] 20 28 9 14 6
> sin(x)+cos(y)
[1] -1.612567896 -0.002900241 -0.848872489 1.663199681 1.801641271
> sum(sin(x)+cos(y))
[1] 1.0005
> sqrt(x)
[1] 2.236068 2.000000 1.732051 1.414214 1.000000
> sqrt(x)^2
[1] 5 4 3 2 1
```

Let’s try to calculate a more complex formula. Imagine vector *x* containing your *n* measurements. The sample variance is defined by

$$\frac{1}{n-1} \sum_{i=1}^n \left(x_i - \frac{1}{n} \sum_{i=1}^n x_i \right)^2 .$$

```
> sum((x-mean(x))^2)/(length(x)-1)
> var(x) # gives the same result!
```

Vector indexing

Individual elements of a vector can be referenced by giving the name of the vector followed by the subscripts in square brackets. You can also use logical expressions for indexing. Some examples:

```
> x <- c(5.6,5.4,2,9,-3.9)
> x
[1] 5.6 5.4 2.0 9.0 -3.9

> x[4]
[1] 9
> x[2:3]
[1] 5.4 2.0
> x[c(1,3)]
[1] 5.6 2.0

> x[x > 4]
[1] 5.6 5.4 9.0
> x > 4
[1] T T F T F          # T = TRUE, F = FALSE
```

Negative indices exclude certain elements from the vector, e.g. `x[-3]` is the same as `x` with the third element missing.

```
> x[-3]
[1] 5.6 5.4 9.0 -3.9
> x
[1] 5.6 5.4 2.0 9.0 -3.9
```

Matrices

Matrices (or more generally *arrays*) are multi-dimensional generalizations of vectors. In fact, they *are* vectors that can be indexed by two or more indices.

```
> M <- 1:20
> dim(M) <- c(4,5)
> M
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

The second assignment gives the vector `M` a *dimension* attribute that allows it to be treated as a 4×5 -matrix (a matrix with 4 rows and 5 columns). We get the same result by

```
> M <- matrix(1:20,4,5)
```

We can index the elements of `M` in the same way we used for vectors. The only difference: now we need two indices in the square brackets, because `M` is two-dimensional. The first index corresponds to the rows, the second to the columns.

```

> M[1,2]      # first row, second column of M
[1] 5

> M[1:2,1:2] # the upper left corner of M
  [,1] [,2]
[1,]   1   5
[2,]   2   6

> M[1:2,-3]
  [,1] [,2] [,3] [,4] # the first two rows
[1,]   1   5  13  17 # with the third
[2,]   2   6  14  18 # column missing
> dim(M[1:2,-3])
[1] 2 4 # 2 rows and 4 columns

> M[2,]      # second row, all columns
[1] 2 6 10 14 18

```

Lists

A *list* is an object consisting of an ordered collection of objects known as its *components*. Components in lists can be of different modes and types (e.g. a character vector, a logical value and a matrix). Components are referred to by a number in double square brackets (in the form `listname[[number]]`) or by a name (in the form `listname$component`). Some easy examples:

```

> Data <- list(measurements = matrix(rnorm(50),10,5),
              tumor.type = factor(c("ER+", "ER-", "ER-", "ER-", "ER+")),
              differential.genes = c("xxY", "xYx", "Yxx")
            )
> summary(Data)

```

	Length	Class	Mode
measurements	50	-none-	numeric
tumor.type	5	factor	numeric
differential.genes	3	-none-	character

```

> Data[[3]]
[1] "xxY" "xYx" "Yxx"
> Data$differential.genes
[1] "xxY" "xYx" "Yxx"

```

What does `Data[[3]][2]` do? Give a command that shows the second row of the measurement matrix in your data without the last entry.

for-loops and apply

```

> M <- matrix(rnorm(50),10,5)

```

This results in a 10×5 matrix filled with normal-distributed random numbers. Imagine we need the sum over each row of this matrix. Idea: go through the matrix row by row and compute the sum in each step.

```

> for(i in 1:10){ print(sum(M[i,])) }

```

Ok, now collect the results of each step in a vector called `results`:

```
> results <- numeric(10)
> for(i in 1:10){ results[i] <- sum(M[i,]) }
```

In R you can do things like this even without using loops (which are a bit slow). The idea is to apply a certain operation (in our case `sum`) to each row of the matrix at the same time:

```
> results2 <- apply(M,1,sum)
|
1 = rows
2 = columns
```

The second argument of the function `apply()` corresponds to the dimensions of the matrix: 1 means rows and 2 means columns. Try `apply(M,2,sum)`.