Chapter 1

Brief Introduction into Using R

To get started, we will first introduce the R statistical programming language. In this chapter, our goal is to quickly get a new user up and running with R and Bioconductor. In particular, we will briefly explain how to: (1) install R and Bioconductor, (2) obtain help about using an R function, and (3) perform simple calculations in R. We also want to equip the new user with practical problem solving and debugging techniques that will make working in the R programming environment much more enjoyable.

We will also introduce how data is stored, accessed, and manipulated in the R programming language. In R, many computations are performed on data vectors and several illustrations of working with vectors are given. For example, in the case of gene expression data, the expression vectors are placed one beneath the other to form a data matrix with the genes as rows and the patients as columns. In general, the data matrix is arguably the most important data structure in R and is therefore extensively explained and illustrated by several examples. A data matrix which we will continually revisit consists of the classical Golub et al. (1999) data, which will be analyzed frequently to illustrate statistical procedures.

1.1 Getting R started on your computer

R is free for the public domain and can be freely downloaded from http://cran.r-project.org. Choose the appropriate operating system (*Windows*, *Linux* or *MacOS*) and simply follow the instructions. If installing on a computer that is running a *64-bit* OS, then it is recommended that the ap-

propriate *64-bit* version of R be installed. Once installed, the user can start R by simply launching the R icon on the desktop. The **RGui** application will popup and present the user with the prompt ">" and wait for input from the user. The *RGui* application that is installed with R is a rather simple graphical user interaface (GUI) that serves at the connecting bridge (interface) between the user and the R engine. Other more sophisticated 3rd party GUI applications exist that can make working with R much easier and more efficient. Feature-rich GUIs for software development are also often referred to as integrated development environments (IDEs). Two popular IDEs that interface with the R engine include *R-Studio* and *Rcmdr*. Fortunately, these powerful IDEs are also freely available and can be downloaded from `http://www.rstudio.com/` and `http://www.rcommander.com/`, respectively.

In this book, all input commands and resultant output from the R engine will be displayed as follows:

```
> input command
[1] output
```

In R, all useful commands (called *functions*) are contained in libraries called *packages*. The standard installation of R includes some basic packages such as *base* and *stats*. In addition, hundreds of more specialized packages are currently available for download from the `The Comprehensive R Archive Network (CRAN)` website at `cran.r-project.org`. These packages are written and made available by the growing R user community. Loaded with hundreds of packages and statistical procedures, the R programming environment has become the de facto standard for analyzing many different types of data.

The command to download and install a specific package is:

```
> install.packages(c("TeachingDemos"),repo="http://cran.r-project.org", dep=TRUE)
```

The above command installs the package *TeachingDemos*, developed by Greg Snow, from the repository at `http://cran.r-project.org`. By setting the option `dep` to `TRUE`, the packages on which the *TeachingDemos* depend are also installed. Setting this option when installing any package is strongly recommended! If omitted, missing dependencies in an R installation can cause erratic behavior. Alternatively, in many GUI front-ends to the R engine, the user can simply click on the appropriate *Packages* button and follow the instructions.[1] The installation of an R package needs to be done

[1] Another way to install a package is by downloading the zip file and placing it unpacked in the proper directory where R can read and load it.

only once. However, in order to use the functions in the package in an R session, the package must be *loaded* into the current R session using the `library()` command. For instance, to produce a nice plot of the outcome of throwing a die twelve times, the user can enter the following:

```
> install.packages("ctv")    # do the install only once!
> library("ctv")               # loads the "ctv" library into the current R session
> install.views("Genetics")
```

The # character is used to comment the R code. All text on a line after the comment character is ignored by the R engine. In general, liberal use of comments will make your code more understandable and maintainable. There is no such thing as over-commented code! In this book, we shall often use packages from *Bioconductor*, a very useful open source software project for the analysis and comprehension of genomic data.

To follow the book it is essential to install *Bioconductor* on your PC or network. Unfortunately, *Bioconductor* packages cannot be installed with the `install.packages()` function like most other packages. Instead, you have to use the `biocLite()` function that is defined in the `\tt http://www.bioconductor.org/biocLite.R` R script. Thus, *Bioconductor* is installed as follows:

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite()    # install or update your Bioconductor packages
```

Next, to (1) download the *Acute Lymphocytic Leukemia (ALL)* package from a repository to your system, (2) load it into the current R session, and (3) make the data available for subsequent analysis, the user should do the following:

```
> biocLite("ALL")    # do the install only once!
> library(ALL)      # load the "ALL" library into the current R session
> data(ALL)         # make the data stored in the libray available for outside access
```

These data will be analyzed extensively later on in Chapters 5 and 6. General help on loaded Bioconductor packages is available by executing `openVignette()`. For further information the reader is referred to `www.bioconductor.org` or to several other URLs[2] .

In this and the following chapters we will illustrate many statistical ideas by applying them to the Golub et al. (1999) data, see also Section 1.9. The `golub` data is made available for analysis by the following commands:

[2] http://mccammon.ucsd.edu/~bgrant/bio3d/user_guide/user_guide.html
http://rafalab.jhsph.edu/software.html
http://dir.gmane.org/gmane.science.biology.informatics.conductor

```
> biocLite("multtest")    # do the install only once!
> library(multtest)
> data(golub)
```

R is object-oriented in the sense that everything consists of objects belonging to certain classes. Type `class(golub)` to obtain the class of the object `golub` and `str(golub)` to obtain its structure or content. Type `objects()` or `ls()` to view the currently loaded objects. By default, references and objects in R are not automatically thrown away. In order to remove any unwanted object and reclaim its memory, the user can enter `rm(object)`. In order to completely remove all objects in memory, the user can enter `rm(list=ls())`. To prevent conflicting definitions, it is wise to remove all unneeded objects at the end of a session and before starting any new analysis. To quit a session, type `q()`, or simply click on the cross in the upper right corner of your screen.

1.2 Getting help

All functionality in R is organized into so-called *packages*. Use the `library()` function to see which packages are currently installed on your operating system. The packages *stats* and *base* are automatically installed since they contain all the basic R functions. To obtain an overview of the content of the package *stats* use `ls(package:stats)` or `library(help="stats")`. Similar information for other packages is possible by replacing the word `stats` with the appropriate package name. Help on a specific function can be obtained from the package manual by typing a question mark in front of a function. For instance, `?sum` gives details on the summation function. Also, when a user is seeking help on a function whose name contains a given substring like `diff`, executing `apropos("diff")` will return the names of all the functions loaded into the current session that contain that string:

```
> apropos("diff")
 [1] "*.difftime"             ".difftime"              "/.difftime"
 [4] "[.difftime"             "as.data.frame.difftime" "as.difftime"
 [7] "as.double.difftime"     "diff"                   "diff.Date"
[10] "diff.default"           "diff.POSIXt"            "diff.ts"
[13] "diffinv"                "diffmeanX"              "diffs.1.N"
[16] "difftime"               "format.difftime"        "is.numeric.difftime"
[19] "Math.difftime"          "mean.difftime"          "Ops.difftime"
[22] "print.difftime"         "setdiff"                "Summary.difftime"
[25] "units.difftime"         "units<-.difftime"       "xtfrm.difftime"
```

Also, when you are starting with a new function such as `boxplot()`, it is often convenient to have an example showing the API, the function output (in this case a plot), and working example code. This is where the `example()` function comes in very handy. The user can get all the above useful information by executing `example(boxplot)`. Lastly, the function `history()` can be useful for viewing previously executed commands.

Type `help.start()` to launch an HTML page that links to several well-written R manuals such as: "An Introduction to R", "The R Language Definition", "R Installation and Administration", and "R Data Import/Export". Further HTML help can be obtained from *http://cran.r-project.org*. The *CRAN* contributed page also contains links to well-written, freely available, on-line books[3] and useful reference charts[4]. Also on the *CRAN* website http://www.r-project.org are powerful search engines such as " *R site search*", and " *Rseek*". Lastly, there are a number of other useful URLs with information on R.[5]

1.3 Getting started with R

R can be used as a simple calculator. For instance, to add 2 and 3 we simply execute the following:

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

In many calculations, the natural base $e = 2.718282$ of exponential functions is used. These type of functions can be called as follows:

```
> exp(1)
[1] 2.718282
```

To compute $e^2 = e \cdot e$ we use `exp(2)`.[6] So, indeed, we have e^x =`exp(x)`, for any value of x. The sum $1 + 2 + 3 + 4 + 5$ can be computed by:

```
> sum(1:5)
[1] 15
```

and the product $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ by:

```
> prod(1:5)
[1] 120
```

[3] "R for Beginners" by Emmanuel Paradis or the "The R Guide" by Jason Owen

[4] "R reference card" by Tom Short or by Jonathan Baron

[5] We mention in particular:
http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/R_BioCondManual.html

[6] The argument of functions is always placed between parentheses `()`.

1.4 Generating a sequence and a factor

In order to compute quantiles of distributions (see Section 2.2.4) or plots of functions, we need to generate sequences of numbers. The easiest way to construct a sequence of numbers is by:

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

This sequence can also be produced by the function `seq()`, which allows for various sizes of steps to be chosen. For instance, in order to compute percentiles of a distribution we may want to generate numbers between zero and one with a step size equal to 0.1:

```
> seq(0,1,0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

For plotting and testing of hypotheses, we need to generate yet another type of a sequence, called a *factor*. It is designed to indicate categories or other qualititative data such as an experimental condition or a group to which a patient belongs.[7] For example, when for each of three experimental conditions there are measurements from five patients, the corresponding factor can be generated using the `gl()` (Generate factor Levels) function:

```
> factor <- gl(3,5)
> factor
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

The three conditions are often called *levels* of a factor. Each of these levels has five repeats corresponding to the number of observations (patients) within each level (type of disease). We shall later illustrate the idea of a factor because it is very useful for purposes of visualization.

Also, variables in R can be assigned in two different ways: either with a left arrow "<-" or with an equal sign "=":

```
> x <- 4 ; y = 3     # two ways to assign values
> x <- y <- 7        # multiple chained assigning
```

1.5 R is a vectorized language

The *vector* is the most basic datatype for data storage in R. It stores one or more values of the *same type*. The concatenate function `c()` can be used

[7] See Samuels & Witmer (2003, Chap. 8) for a full explanation of experiments and statistical principles of design.

to construct a vector from individual elements - again, of the same type. In addition, most functions in R are designed to work on vectors of length 1 or more:

```
> x = c(11, 12, 13, 14) ; x = 11:14
> x
[1] 11 12 13 14
> x+100                 # R functions are vectorized
[1] 111 112 113 114
> x^2                   # square function is vectorized too
[1] 121 144 169 196
```

Brackets are used for subsetting a vector. The returned value is another vector with potentially a subset of elements:

```
> x[4] # Accessing the fourth element
[1] 14
```

A common idiom in R is that "everything is a vector". This means that the most basic data structure in R is a vector or can be referenced as one (if possible):

```
> x = 22
> x
[1] 22           # x is a vector of length 1
> x[1]
[1] 22           # single subsetting still returns a vactor of length 1
> x[1][1]
[1] 22           # double subsetting also returns a vactor of length 1!
> x[1][1][1]
[1] 22           # triple subsetting also returns a vactor of length 1!
> x[1][1][1][1]
[1] 22           # it never ends....
```

Since vectors are the most basic data structure in R, all native functions in R are said to be *vectorized* - in that by default they accept and return vectors:

```
> x = 11:14
> x>12                      # produces a boolean vector
[1] FALSE FALSE  TRUE  TRUE
> x[x>12]                   # subset x based on the boolean vector
[1] 13 14
> x[1:3]                    # return thr 1st 3 elements of x
[1] 11 12 13
> is.na(x)                  # produces another boolean vector
[1] FALSE FALSE FALSE FALSE
> cumsum(x)                 # produces a numeric vector
[1] 11 23 36 50
```

Vectors must consist of elements of the same datatype. Below are 4 of the basic datatypes (classes) in R:

```
> x <- c("DNA", "RNA", "Protein"); class(x)
```

```
[1] "character"
> x <- c(1.2, 1.5); class(x)
[1] "numeric"
> x <- 1:4; class(x)
[1] "integer"
> x <- x>12; class(x)
[1] "logical"
```

1.6 Computing on a data vector

We will perform our first calculations on a data vector that is a collection of numbers obtained as outcomes from expression measurements of a gene. Suppose that the gene expression values $1, 1.5$, and 1.25 from the persons "Eric", "Peter", and "Anna" are available. To store these in a vector we use the concatenate function `c()` as follows:

```
> gene1 <- c(1.00,1.50,1.25)    # assign the variable gene1 to 3 expression values
> gene1
[1] 1.00 1.50 1.25
```

Now we have created the object `gene1` containing three gene expression values. To compute the sum, mean, and standard deviation of the gene expression values, we use the following functions:

```
> sum(gene1)
[1] 3.75
> mean(gene1)
[1] 1.25
> sum(gene1)/3
[1] 1.25
> sd(gene1)
[1] 0.25
> sqrt(sum((gene1-mean(gene1))^2)/2)
[1] 0.25
> summary(gene1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.125   1.250   1.250   1.375   1.500
```

By defining $x_1 = 1.00$, $x_2 = 1.50$, and $x_3 = 1.25$, the sum of the weights can be expressed as $\sum_{i=1}^{n} x_i = 3.75$. The mathematical summation symbol $\sum$ is calculated by the R function `sum()`. The mean is denoted by $\overline{x} = \sum_{i=1}^{3} x_i/3 = 1.25$ and the sample standard deviation as

$$s = \sqrt{\sum_{i=1}^{3}(x_i - \overline{x})^2/(3-1)} = 0.25.$$

Dataset are often incomplete with possibly many missing values. Often, these missing values are represented as NAs in the data. We can get rid of NAs in a data vector with the `is.na()` function.

```
> x <- c("DNA", "RNA", "Protein") # Vector of strings
> x
[1] "DNA"     "RNA"     "Protein"
> x[2] <- NA   # Replace the second element of the vector
> x
[1] "DNA"     NA        "Protein"
> x[!is.na(x)]
[1] "DNA"     "Protein"
```

1.7 Constructing a data matrix

In various types of spreadsheets it is customary to store data values in the form of a 2-dimensional matrix consisting of rows and columns. Likewise, in bioinformatics gene expression values (from several groups of patients) are stored as rows such that each row contains the expression values of all the patients for a particular gene and each column contains all the gene expression values for a particular person. Let's illustrate this by a small example. Suppose that we have the following expression values on three genes from Eric, Peter, and Anna.[8]

```
> gene2 <- c(1.35,1.55,1.00)
> gene3 <- c(-1.10,-1.50,-1.25)
> gene4 <- c(-1.20,-1.30,-1.00)
```

Before constructing the matrix it is convenient to add the names of the rows and the columns. We do so by constructing the following list:

```
> rowColNames <- list(c("gene1","gene2","gene3","gene4"),
+     c("Eric","Peter","Anna"))
```

After the last comma in the first line we entered a carriage return for R to create a new line starting with + in order to complete a command. Now we can construct a 2-dimensional matrix called `geneData` containing the expression values from our four genes by using the matrix function:

```
> geneData <- matrix(c(gene1,gene2,gene3,gene4), nrow=4, ncol=3,
+    byrow=TRUE, dimnames = rowColNames)
```

Always use variable names that are descriptive of the underlying data. Never use meaningless names like "variable1" and "variable2" - which would

[8] With the function `data.entry()` you can open and edit a screen with the values of a matrix.

only make the code difficult to understand, debug, and maintain. Here, `nrow` indicates the number of rows and `ncol` the number of columns. The gene vectors are placed in the matrix as rows. The names of the rows and columns are attached by the `dimnames` parameter. Note that internally a matrix is a vector of row or columns vectors with possibly named rows and columns. To see the content of our newly created object `geneData`, we print it to the screen:

```
> geneData
       Eric Peter  Anna
gene1  1.00  1.50  1.25
gene2  1.35  1.55  1.30
gene3 -1.10 -1.50 -1.25
gene4 -1.20 -1.30 -1.00
```

A matrix such as `geneData` has two indices `[i,j]`, the first of which refers to rows and the second to columns[9]. Thus, if we want to print the second element of the first row to the screen, we type `geneData[1,2]`. Also, not specifying a row or column at all results in grabbing all of them. For example, if we want to print the entire first row, we use `geneData[1, ]`. For the entire second column, we use `geneData[ ,2]`:

```
> geneData[1,2]
[1] 1.5
> geneData[1, ]
 Eric Peter  Anna
 1.00  1.50  1.25
> geneData[ ,2]
gene1 gene2 gene3 gene4
 1.50  1.55 -1.50 -1.30
```

It may be desirable to write the data to a file either for later use or to send them to a collaborator. The following commands illustrate how to save and load a data table in R:

```
> system("bash -c \"mkdir ~/biology664\"")   # create the destination directory, do only
    ↪ once
> write.table(geneData,file="~/biology664/geneData.Rdata")
> geneDataRead <- read.table("~/biology664/geneData.Rdata")
> geneDataRead
       Eric Peter  Anna
gene1  1.00  1.50  1.25
gene2  1.35  1.55  1.00
gene3 -1.10 -1.50 -1.25
gene4 -1.20 -1.30 -1.00
```

An alternative is to use the similar functions `write.csv()` and `read.csv()`.[10]

[9] Indices referring to rows, columns, or elements are always between square brackets `[]`.

[10] For more see the "R Data import/Export" manual, Chapter 3 of the book "R for Beginners", or search the internet by the key "r wiki matrix".

1.8 Computing on a data matrix

Means and standard deviations of rows or columns are often important for drawing biologically relevant conclusions. Such computations on a data matrix can be accomplished by using *for loops*. However, it is much more convenient to use the `apply()` function on a matrix. To do so we will: (1) specify the name of the matrix, (2) indicate either rows or columns (1 for rows and 2 for columns), and (3) supply the name of the function we wish to apply to either the rows or columns. To illustrate this we compute the mean of each person (column):

```
> apply(geneData,2,mean)
   Eric  Peter   Anna
0.0125 0.0625 0.0750
```

Similarly, the mean of each gene (row) can be computed:

```
> apply(geneData,1,mean)
    gene1     gene2     gene3     gene4
 1.250000  1.400000 -1.283333 -1.166667
```

Also, since R is a vectorized language, the `apply()` family of functions is highly optimized and will almost always generate faster executing code than writing custom *for loops*. We will cover the entire family of `apply()` functions later in the book.

It frequently occurs that we want to reorder the rows of a matrix according to a certain criterion, or, more specifically, the values in a certain column vector. For instance, to reorder the matrix `geneData` according to the row means, it is convenient to store these in a vector and then to use the `order()` function:

```
> meanExpressions <- apply(geneData,1,mean)
> o <- order(meanExpressions,decreasing=TRUE)
> o
[1] 2 1 4 3
```

Thus, `gene2` appears first because it has the largest mean of 1.4, then `gene1` with 1.25, followed by `gene4` with -1.16, and finally `gene3` with -1.28. Now that we have collected the order numbers in the vector `o`, we can reorder the whole matrix by specifying `o` as the row index:[11]

```
> geneData[o, ]
       Eric Peter  Anna
gene2  1.35  1.55  1.30
gene1  1.00  1.50  1.25
gene4 -1.20 -1.30 -1.00
```

[11] You can also use functions like `sort()` or `rank()`.

```
gene3 -1.10 -1.50 -1.25
```

Another frequently occurring problem is that of selecting genes with a certain property. We illustrate this by several methods to select genes with positive mean expression values. The first method starts with the observation that the first two rows have positive means and to use `c(1,2)` as a row index:

```
> geneData[c(1,2), ]
      Eric Peter Anna
gene1 1.00  1.50 1.25
gene2 1.35  1.55 1.30
```

A second way is to use the row names as an index:

```
> geneData[c("gene1","gene2"), ]
      Eric Peter Anna
gene1 1.00  1.50 1.25
gene2 1.35  1.55 1.30
```

A third and more advanced way is to construct a boolean evaluation in terms of TRUE or FALSE and place the results in a logical (boolean) vector. For instance, we may evaluate whether the row mean is positive:

```
> meanExpressions > 0
gene1 gene2 gene3 gene4
 TRUE  TRUE FALSE FALSE
```

Now we can use the evaluation of `meanExpressions > 0` in terms of the values `TRUE` or `FALSE` as a row index:

```
> geneData[meanExpressions > 0, ]
      Eric Peter Anna
gene1 1.00  1.50 1.25
gene2 1.35  1.55 1.30
```

Observe that this selects genes for which the evaluation equals `TRUE`. Above, we have illustrated that genes can be selected by (1) their row index, (2) row name, or (3) value on a logical variable.

1.9 Application to the Golub (1999) data

The gene expression data collected by Golub et al. (1999) are among the first classical datasets used in bioinformatics. A selection of the dataset called `golub` is contained in the `multtest` package, which is part of Bioconductor. The data consist of gene expression values for 3051 genes (rows) from 38 leukemia patients (columns)[12]. Twenty seven patients are diagnosed with

[12]The data are pre-processed by procedures described in Dudoit et al. (2002).

acute lymphoblastic leukemia (ALL) and eleven with acute myeloid leukemia (AML). The tumor class is given by the numeric vector `golub.cl`, where ALL is indicated by 0 and AML by 1. The gene names are collected in the matrix `golub.gnames` - of which the columns correspond to the gene index, ID, and Name, respectively.

We shall first focus on the expression values of a gene with manufacturer name *M92287_at*, which is known in biology as *CCND3 (Cyclin D3)*. The expression values of this gene are collected in row 1042 of the `golub` table. To load the data and obtain the relevant information from row 1042 of `golub.gnames` we can do the following:

```
> library(multtest); data(golub)
> golub.gnames[1042, ]
[1] "2354"            "CCND3 Cyclin D3" "M92287_at"
```

The data are stored in a matrix called `golub`. The number of rows and columns can be obtained by the functions `nrow()` and `ncol()`, respectively:

```
> nrow(golub)
[1] 3051
> ncol(golub)
[1] 38
```

Thus, the matrix has 3051 rows and 38 columns, see also `dim(golub)`. Each data element has a row and a column index. Recall that the first index refers to rows and the second to columns. Hence, the second value from row 1042 can be printed to the screen as follows:

```
> golub[1042,2]
[1] 1.52405
```

As a result, 1.52405 is the expression value of gene CCND3 (Cyclin D3) from patient number 2. All the gene expression values for the first patient (column) can be printed to the screen by the following:

```
> golub[ ,1]
```

To save space the output is not shown. We may now print the expression values of gene CCND3 (Cyclin D3) (row 1042) to the screen:

```
> golub[1042, ]
 [1]  2.10892  1.52405  1.96403  2.33597  1.85111  1.99391  2.06597  1.81649
 [9]  2.17622  1.80861  2.44562  1.90496  2.76610  1.32551  2.59385  1.92776
[17]  1.10546  1.27645  1.83051  1.78352  0.45827  2.18119  2.31428  1.99927
[25]  1.36844  2.37351  1.83485  0.88941  1.45014  0.42904  0.82667  0.63637
[33]  1.02250  0.12758 -0.74333  0.73784  0.49470  1.12058
```

To print the expression values of gene CCND3 (Cyclin D3) to the screen only for the ALL patients, we can to refer to the first twenty seven elements of row 1042. We can do so with the following command:

```
> golub[1042,1:27]
 [1] 2.10892 1.52405 1.96403 2.33597 1.85111 1.99391 2.06597 1.81649 2.17622
[10] 1.80861 2.44562 1.90496 2.76610 1.32551 2.59385 1.92776 1.10546 1.27645
[19] 1.83051 1.78352 0.45827 2.18119 2.31428 1.99927 1.36844 2.37351 1.83485
```

However, the method above requires that all the ALL patients are always placed in the first 27 columns of the matrix. If the order of the columns changes or if new patients (columns) are added, then the above R code will no longer return the correct results. For these reasons it is much more convenient to construct a *factor* indicating the tumor class of the patients. This will turn out to be very useful - e.g. for separating the tumor groups in various visualization procedures. The factor will be called `golubFactor` and is constructed from the vector `golub.cl` as follows:

```
> golubFactor <- factor(golub.cl, levels=0:1, labels = c("ALL","AML"))
```

In subsequent chapters, this factor will be used frequently. Obviously, the labels correspond to the two tumor classes: acute lymphoblastic leukemia (ALL) and acute myeloid leukemia (AML). The evaluation of `golubFactor == ''ALL''` returns `TRUE` for the first twenty seven values and `FALSE` for the remaining eleven. This boolean vector is useful as a column index for selecting the expression values of the ALL patients. The expression values of gene CCND3 (Cyclin D3) from the ALL patients can now be printed to the screen as follows:

```
> golub[1042,golubFactor=="ALL"]
 [1] 2.10892 1.52405 1.96403 2.33597 1.85111 1.99391 2.06597 1.81649 2.17622
[10] 1.80861 2.44562 1.90496 2.76610 1.32551 2.59385 1.92776 1.10546 1.27645
[19] 1.83051 1.78352 0.45827 2.18119 2.31428 1.99927 1.36844 2.37351 1.83485
```

For many types of computations it is very useful to combine a factor with the `apply()` function. For instance, to compute the mean gene expression across the ALL patients for each of the genes, we may do the following:

```
> meanALL <- apply(golub[ ,golubFactor=="ALL"], 1, mean)
```

The specification `golub[ ,golubFactor==''ALL'']` selects the matrix with gene expressions corresponding to the ALL patients. The 3051 means are assigned to the vector `meanALL`.

After reading the classical article by Golub et al. (1999), one becomes easily interested in the properties of certain genes. For instance, gene CD33 plays an important role in distinguishing lymphoid from myeloid lineage cells. To perform computations on the expressions of this gene we need to know its row index. This can be obtained with the `grep()` function:[13]

[13] Indeed, several functions in R are inspired by the Linux operating system.

```
> cd33 = grep("CD33",golub.gnames[ ,2], ignore.case = TRUE)
> cd33
[1] 808
```

Hence, the expression values of antigen CD33 are available at `golub[cd33, ]` and additional information is available at `golub.gnames[cd33, ]`:

```
> golub[cd33, ]
 [1] -0.57277 -1.38539 -0.47039 -0.41469 -0.15402 -1.21719 -1.37386 -0.52956
 [9] -1.10366 -0.74396 -0.97673 -0.00787 -0.99141 -1.05662 -1.39503 -0.73418
[17] -0.67921 -0.87388 -0.82569 -1.12953 -0.75991 -0.92231 -1.13505 -1.46474
[25] -0.59614 -1.04821 -1.23051 -0.38605  0.50814  0.70283  1.05902  0.38602
[33] -0.19413  1.10560  0.76630  0.48881 -0.13785 -0.40721
> golub.gnames[cd33, ]
[1] "1834"
[2] "CD33 CD33 antigen (differentiation antigen)"
[3] "M23197_at"
```

1.10 Constructing a data.frame

A data.frame is used for storing data tables (like a matrix) - but where different columns can contain different datatypes. As a result, data.frames are more flexible than matrices. A data.frame consists of a list of column vectors of equal length, where the data in each column must be of the same type. The column labels can be defined when creating the data.frame or afterwards using the `colnames()` function. Also, when creating a data.frame you can concatenate multiple matrices and/or columns together. For example, in the following examples we concatenate 3 column vectors together to create a data.frame. Lastly, the `head()` and `tail()` functions are very useful to view just the first or last few rows of a large matrix or data.frame, respectively:

```
> patients.df <- data.frame(    # Define the 3 column names when creating the data.frame
+   patientID = c("101", "102", "103", "104"),
+   treatment = c("drug", "placebo", "drug", "placebo"),
+   age = c(20, 30, 24, 22)
+ )
>
> patients.df <- data.frame(    # Define the 3 column names after creating the data.frame
+   c("101", "102", "103", "104"),
+   c("drug", "placebo", "drug", "placebo"),
+   c(20, 30, 24, 22)
+ )
> colnames(patients.df) <- c("patientID", "treatment", "age")
>
> patients.df
  patientID treatment age
1       101      drug  20
2       102   placebo  30
3       103      drug  24
```

```
4       104   placebo  22
> nrow(patients.df)
[1] 4
> ncol(patients.df)
[1] 3
> head(patients.df)
  patientID treatment age
1       101      drug  20
2       102   placebo  30
3       103      drug  24
4       104   placebo  22
```

1.11 Referencing column vectors in a data.frame

A column vector in a data.frame can be referenced in multiple ways - with *named indexing* (also called *named referencing*) always being the preferred method. Besides being more descriptive and thus more understandable, *named indexing* is also considered superior since adding new columns or rearranging existing ones will not affect the referencing. Due to these properties, *named indexing* creates more *reusable code* than *hardcoding* the index to some integer. If a new version of the data becomes available, then the existing R code that uses *named indexing* will be able to run on that data without modification and is thus more *reusable*:

```
> patients.df[[2]]           # a column vector using hardcoded member referencing (double
    ↪ brackets)
[1] drug    placebo drug    placebo
Levels: drug placebo
> patients.df[["treatment"]] # a named column vector using named referencing method 1 (
    ↪ double brackets)
[1] drug    placebo drug    placebo
Levels: drug placebo
> patients.df$treatment      # a named column vector using named referencing method 2 - THE
    ↪  PREFERRED METHOD
[1] drug    placebo drug    placebo
Levels: drug placebo
> patients.df[,2]            # a column vector specifying all rows and column 2 (hardcoded)
[1] drug    placebo drug    placebo
Levels: drug placebo
```

1.12 Row and column subsetting (slicing) on a data.frame

A row slice or subset of a data.frame returns a data.frame with a subset of the rows:

```
> patients.df[c(1,3),]                          # a hardcoded row slice or subset containing
    ↪  rows 1, 2, and 3
  patientID treatment age
1       101      drug  20
3       103      drug  24
> patients.df[patients.df$treatment=="drug",] # a named row slice or subset using logical
    ↪ indexing  - THE PREFERRED METHOD
  patientID treatment age
1       101      drug  20
3       103      drug  24
>
> treatmentIsDrug = patients.df$treatment=="drug"
> treatmentIsDrug
[1]  TRUE FALSE  TRUE FALSE
> patients.df[treatmentIsDrug,]                 # a row slice or subset using logical
    ↪ indexing
  patientID treatment age
1       101      drug  20
3       103      drug  24
>
> subset(patients.df, treatment=="drug")        # a named row slice or subset using the
    ↪ subset function
  patientID treatment age
1       101      drug  20
3       103      drug  24
>
> row.names(patients.df) = patients.df$patientID
> patients.df
    patientID treatment age
101       101      drug  20
102       102   placebo  30
103       103      drug  24
104       104   placebo  22
> patients.df[c("101", "103"),]                 # a row slice or subset using name indexing
    patientID treatment age
101       101      drug  20
103       103      drug  24
```

A column slice or subset of a data.frame also returns a data.frame with a subset of the columns:

```
> patients.df[2]                    # a hardcoded column 2 slice or subset
    treatment
101      drug
102   placebo
103      drug
104   placebo
> patients.df["treatment"]          # a named column slice or subset  - THE PREFERRED
    ↪ METHOD
    treatment
101      drug
102   placebo
103      drug
104   placebo
> patients.df[c("treatment", "age")] # a named columns slice or subset - THE PREFERRED
    ↪ METHOD
    treatment age
```

```
101      drug  20
102   placebo  30
103      drug  24
104   placebo  22
```

1.13 Adding columns and rows to a data.frame

A new column vector can be added to an existing data.frame using the `cbind()` function. Note that the length of the column vector must be the same length as the existing columns in the data.frame (which is the same as the number of rows):

```
> patients.df2 = cbind(patients.df, weight=c(160, 114, 210, 102))
> patients.df2
  patientID treatment age weight
1       101      drug  20    160
2       102   placebo  30    114
3       103      drug  24    210
4       104   placebo  22    102
>
> patients.df3 = cbind(patients.df2, gender=c("male", "female", "male", "female"))
> patients.df3
  patientID treatment age weight gender
1       101      drug  20    160   male
2       102   placebo  30    114 female
3       103      drug  24    210   male
4       104   placebo  22    102 female
```

Also, you can concatenate the rows of two data.frames together using the `rbind()` function. Note that the number of columns and the datatypes of each column must match in order to concatenate the two data.frames together:

```
> patients.df1 <- data.frame(    # Define the 3 column names when creating the data.frame
+   patientID = c("101", "102", "103", "104"),
+   treatment = c("drug", "placebo", "drug", "placebo"),
+   age = c(20, 30, 24, 22)
+ )
> patients.df2 <- data.frame(    # Define the 3 column names when creating the data.frame
+   patientID = c("97", "98", "99"),
+   treatment = c("drug", "placebo", "drug"),
+   age = c(24, 31, 42)
+ )
> pateints.df1and2 = rbind(patients.df1, patients.df2)
> pateints.df1and2
  patientID treatment age
1       101      drug  20
2       102   placebo  30
3       103      drug  24
4       104   placebo  22
5        97      drug  24
```

```
6        98   placebo  31
7        99      drug  42
```

1.14 Merging two data.frames

Multiple data.frames can also be merged (joined) together into a larger data.frame if they share at least one common column. For example, by using the `merge()` function, two data.frames can be joined *by* their shared column(s). The four main different types of joins between two data.frames are *inner join*, *left outer join*, *right outer join*, and *(full) outer join*. Table 1.1 summarizes these four different types of joins for two data.frames `A` and `B`, with the corresponding option for the `merge()` function.

Table 1.1: The four main types of joins using the merge() function.

Join type	description	set notation	merge() option
Inner	keep only rows that match between A and B	$A \cap B$	`all = FALSE`
Left Outer	all the rows of A and only those from B that match	A	`all.x = TRUE`
Right Outer	all the rows of B and only those from A that match	B	`all.y = TRUE`
(full) Outer	keep all rows from both data frames A and B	$A \cup B$	`all = TRUE`

```
> A <- data.frame(    # Define the 3 column names when creating the data.frame
+   patientID = c("101", "102", "103", "104"),
+   treatment = c("drug", "placebo", "drug", "placebo"),
+   age = c(20, 30, 24, 22)
+ )
> B <- data.frame(    # Define the 3 column names when creating the data.frame
+   patientID = c("101", "102", "105", "106"),
+   gender = c("male", "female", "male", "female"),
+   weight = c(160, 114, 224, 130)
+ )
>
> innerJoin = merge(x=A, y=B, by="patientID", all=FALSE)
> innerJoin
  patientID treatment age gender weight
1       101      drug  20   male    160
2       102   placebo  30 female    114
>
> leftOuterJoin = merge(x=A, y=B, by="patientID", all.x=TRUE)
> leftOuterJoin
  patientID treatment age gender weight
1       101      drug  20   male    160
2       102   placebo  30 female    114
3       103      drug  24   <NA>     NA
4       104   placebo  22   <NA>     NA
>
> outerJoin = merge(x=A, y=B, by="patientID", all=TRUE)
> outerJoin
  patientID treatment age gender weight
```

```
1       101      drug  20   male    160
2       102   placebo  30 female    114
3       103      drug  24   <NA>     NA
4       104   placebo  22   <NA>     NA
5       105      <NA>  NA   male    224
6       106      <NA>  NA female    130
```

Notice that any missing data from a *left outer join*, *right outer join*, or *(full) outer join* is filled in with *NA*s.

1.15 Constructing a list

A list is a generic vector containing objects of potentially more than one datatype (class). A list is the most generic (and confusing) datatype in R. A list can contain any data structure within it, including other lists:

```
> list1 <- list(
+   c("p53", "p63", "p73"),
+   matrix(1:10, nrow=2),
+   c(TRUE, FALSE, TRUE, FALSE, FALSE)
+ )
> list1 # without named elements
[[1]]
[1] "p53" "p63" "p73"

[[2]]
     [ ,1] [ ,2] [ ,3] [ ,4] [ ,5]
[1, ]    1    3    5    7    9
[2, ]    2    4    6    8   10

[[3]]
[1]  TRUE FALSE  TRUE FALSE FALSE

>
> # The better way is to define the elements
> p53FamilyGenes = c("p53", "p63", "p73")
> list2 <- list(
+   genes = p53FamilyGenes,
+   matrix1 = matrix(1:10, nrow=2),
+   remission = c(TRUE, FALSE, TRUE, FALSE, FALSE)
+ )
> list2 # with named columns
$genes
[1] "p53" "p63" "p73"

$matrix1
     [ ,1] [ ,2] [ ,3] [ ,4] [ ,5]
[1, ]    1    3    5    7    9
[2, ]    2    4    6    8   10

$remission
[1]  TRUE FALSE  TRUE FALSE FALSE
```

1.16 Subsetting (slicing) of a list

List slicing or subsetting is accomplished by using single brackets. The returned value is another list with a subset of the elements:

```
> list1[1]                          # hardcoded sublist containing element 1
[[1]]
[1] "p53" "p63" "p73"

> list1[1:2]                        # hardcoded sublist containing elements 1 and 2
[[1]]
[1] "p53" "p63" "p73"

[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> list2["genes"]                    # sublist containing the element named "genes" - THE
    ↪ PREFERRED METHOD
$genes
[1] "p53" "p63" "p73"

> list2[c("genes", "remission")]    # sublist containing the elements named "genes" and "
    ↪ remission" - THE PREFERRED METHOD
$genes
[1] "p53" "p63" "p73"

$remission
[1]  TRUE FALSE  TRUE FALSE FALSE
```

However, list member referencing is accomplished by using double brackets. Note that list member referencing returns the actual referenced element whereas single-bracket slicing returns the element in a list.

```
> list2[[1]]             # Retrieving element 1 (hardcoded)
[1] "p53" "p63" "p73"
> list2[[1]][1] = "her2" # re-assigning within element 1 (harcoded)
> list2[[1]]             # Retrieving element 1 (hardcoded)
[1] "her2" "p63"  "p73"
> p53FamilyGenes         # the original variable p53FamilyGenes is unaffected!
[1] "p53" "p63" "p73"
>
> list2[["genes"]]       # named referencing method 1
[1] "her2" "p63"  "p73"
> list2$genes            # named referencing method 2 - THE PREFERRED METHOD
[1] "her2" "p63"  "p73"
```

1.17 Search path attachment

It's also possible to attach to a data.frame through "search path attachment" using the `attach()` function. Attaching to *data.frames* is convenient when

we have more than one data.frame with the same internal structure and we wish to use the same commands on each of them by using “attach" and “detach" respectively:

```
> attach(list2)
> genes
[1] "her2" "p63"  "p73"
> detach(list2)
>
> attach(patients.df)
> treatment
[1] drug    placebo drug    placebo
Levels: drug placebo
> detach(patients.df)
```

However, be careful when using `attach()` and `detach()`. Unintentional masking can lead to unintentional results:

```
> genes = c("p53", "p63", "p73")
> attach(list2)
The following object(s) are masked _by_ '.GlobalEnv':

    genes
> remission
[1]  TRUE FALSE  TRUE FALSE FALSE
> genes # PROBLEM!!: remission is what we expect, but not genes!! genes has been "masked"
    ↪ by the global value
[1] "p53" "p63" "p73"
> detach(list2) # always detach at the end of your R script!
> gene1 <- c(1.00,1.50,1.25)
> summary(gene1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.125   1.250   1.250   1.375   1.500
```

1.18 Pass-by-value

The example code below illustrates that R is a *pass-by-value* programming language. In the command `y = x`, y is not assigned directly to x, but rather to a copy of the values referenced by the variable x. Therefore, changing the values of x after the `y = x` command does not change the values of y:

```
> x = 10:20     # x is assigned to a vector of integers from 10 to 20
> x
 [1] 10 11 12 13 14 15 16 17 18 19 20
> x * 100       # functions applied to x do not chaange the values of x
 [1] 1000 1100 1200 1300 1400 1500 1600 1700 1800 1900 2000
> x
 [1] 10 11 12 13 14 15 16 17 18 19 20   # the values in x have not changed
> y = x         # assign y to a copy of the vector referenced by x
> x = x * 100   # re-assign x to x * 100
> y             # however, re-assiging x has not changed the variable y
 [1] 10 11 12 13 14 15 16 17 18 19 20
```

1.19 Looping in R

Sometimes it is necessary to "loop" over a block of code and execute the commands contained in the loop repeatedly - over-and-over again until some condition is met. When looping over code in R , always use braces to denote the block of code that is being looped over! Using braces to denote your code blocks makes your code more readable and less bug prone. A *for loop* has the following structure in R:

```
for (variable in sequence) {
    statements
}
```

For example, here is a *for loop* that defines a variable called `index` that takes on values `1` through `5`. The *body* of the *for loop* prints the value of the index to the terminal:

```
> for (index in 1:5) {
+   print(index)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Here is a similar *for loop* that now has an extra *stop condition* that ends the looping prematurely when the variable stopCondition equals 4:

```
> stopCondition = 4
> for (index in 1:10) {
+   if (index < stopCondition) {
+     print(index)
+   }
+   else {
+     break  # break-out of the "for loop"
+   }
+ }
[1] 1
[1] 2
[1] 3
```

A *while loop* is similar to a *for loop* except that it is more general. A *while loop* loops forever until the condition statement is false (or a stop command is encountered):

```
while(condition is true) {
  statements
}
```

For example, the *while loop* below will loop forever printing out values of `z` until the condition statement `z < 5` is false:

```
> z <- 0
> while(z < 5) {
+   z <- z + 2
+   print(z)
+ }
[1] 2
[1] 4
[1] 6
```

1.20 Running scripts

It is very convenient to use a text editor like Notepad, Kate, Emacs, or WinEdt for the formulation of several consecutive R commands on multiple lines within separated files - which are known as *R scripts*. In addition, most IDEs for R have text editors built into them that are specifically designed for the creation of R scripts (e.g. R-Studio and Rcmdr). Short R scripts can be executed by simply copy-and-pasting them to the R command line. Another option is to execute a script from a file using the `source()` function. To illustrate the latter consider the following:

```
> library(multtest); data(golub)
> golubFactor <- factor(golub.cl,levels=0:1, labels= c("ALL","AML"))
> meanALL <- apply(golub[ ,golubFactor=="ALL"], 1, mean)
> meanAML <- apply(golub[ ,golubFactor=="AML"], 1, mean)
> o <- order(abs(meanALL-meanAML), decreasing=TRUE)
> print(golub.gnames[o[1:5],2])
[1] "CST3 Cystatin C (amyloid angiopathy and cerebral hemorrhage)"
[2] "INTERLEUKIN-8 PRECURSOR"
[3] "Interleukin 8 (IL8) gene"
[4] "DF D component of complement (adipsin)"
[5] "MPO Myeloperoxidase"
```

The row means of the expression values per patient group are computed and stored in the object `meanALL` and `meanAML`, respectively. The absolute values of the differences in means are computed and their order numbers (from large to small) are stored in the vector `o`. Next, the names of the five genes with the largest differences between means are printed to the screen.

After saving the script under the name `meanDiff.R` in the directory `C:\\Rscripts` on a PC, it can be executed by using `source("C:\\Rscripts\\meanDiff.R")`. Note that on UNIX and Mac OSs, the pathnames contain single forward slashes `"/"` instead of a drive letter and backslashes `"\\"` like this: `source("~/Rscripts/meanDiff.R")`.

```
> source("~/Rscripts/meanDiff.R")  # forward slashes used on UNIX and Mac OSs
[1] "CST3 Cystatin C (amyloid angiopathy and cerebral hemorrhage)"
[2] "INTERLEUKIN-8 PRECURSOR"
[3] "Interleukin 8 (IL8) gene"
[4] "DF D component of complement (adipsin)"
[5] "MPO Myeloperoxidase"
```

Once the script is saved, any text editor can be used to easily modify and re-run it.

Readers are strongly recommended to experiment by trial-and-error with respect to writing programming scripts. Learning any new programming language requires hands-on experimentation and R is no exception. We also recommend using your favorite text editor to write and save all your R code and to either *source* your saved R scripts using the `source()` function or copy-and-paste the code into the R terminal.

1.21 Overview and concluding remarks

It is easy to install R and Bioconductor. R has many convenient built-in-functions for statistical programming. In addition, help and illustrations on many analysis topics and R functions are available from various sources. With the reference charts, R manuals, on-line books, and R Wiki at hand, the user has various sources of information to help him or her in the process of becoming proficient in R programming. Although there are several good R IDEs available, we shall remain IDE-neutral and instead focus on the default command line editor.

The above introduction is of course very condensed. A more extensive introduction into R that assumes some background on biomedical statistics is given by Dalgaard (2002). There are also book-length publications that combine R with statistics (Venables, & Ripley, 2002; Everitt & Hothorn, 2006). Additional publications that delve much deeper into the programming aspects include Becker, Chambers, & Wilks, 1988; Venables & Ripley, 2000; and Gentleman, 2008.

For the sake of illustration, we shall work frequently with the data kindly provided by Golub et al. (1999) and Chiaretti et al. (2004). The corresponding scientific articles are freely available from the web. Having these articles available may further motivate readers for future computations.

1.22 Exercises

1. **Some questions to orientate yourself.**

   (a) Use the function `class()` to find the class to which the following objects belong: `golub`, `golub[1,1]`,`golub.cl`, `golub.gnames`, `apply()`, `exp()`, `golubFactor`, `plot()`, `ALL`.

   (b) Define the meaning of the following abbreviations: `rm()`, `sum()`, `prod()`, `seq()`, `sd()`, `nrow()`.

   (c) For what purpose are the following functions useful: `grep()`, `agrep()`, `apply()`, `gl()`, `library()`, `source()`, `setwd()`, `history()`, `str()`.

2. **Standard deviations.** Consider the data in the matrix `geneData` that is constructed in Section 1.7. Its small size enables you to check your computations even with a pocket calculator. [14]

   (a) Use `apply()` to compute the standard deviation of the persons.

   (b) Use `apply()` to compute the standard deviation of the genes.

   (c) Order the matrix according to the gene standard deviations.

   (d) Which gene has the largest standard deviation?

3. **Computations of gene expression means in the Golub data.**

   (a) Use `apply()` to compute the mean gene expression value.

   (b) Order the data matrix according to the gene means.

   (c) Give the biological names of the three genes with the largest mean expression value.

4. **Computations of gene expression standard deviations in the Golub data.**

   (a) Use `apply()` to compute the standard deviation per gene.

   (b) Select the expression values of the genes with standard deviation larger than 0.5.

[14] Obtaining some routine knowledge with the apply() functionality is quite helpful for what follows.

(c) How many genes have this property?

5. **Oncogenes in Golub data.**

   (a) How many oncogenes are there in the dataset? Hint: Use `grep()`.

   (b) Find the biological names of the three oncogenes with the largest mean expression value for the ALL patients.

   (c) Do the same for the AML patients.

   (d) Write the gene probe ID and the gene names of the ten genes with largest mean expression value to a `csv` file.

6. **Constructing a factor.** Construct factors that correspond to the following setting.

   (a) An experiment with two conditions each with four measurements.

   (b) Five conditions each with three measurements.

   (c) Three conditions each with five measurements.

7. **Gene means for B1 patients.** Load the `ALL` data from the `ALL` library and use `str()` and `openVignette()` for further information about this dataset.

   (a) Use `exprs(ALL[ ,ALL$BT==‘‘B1’’])` to extract the gene expressions from the patients in disease stage B1. Compute the mean gene expressions across these patients.

   (b) Give the gene identifiers of the three genes with the largest mean expression.