DNA microarray data analysis using Bioconductor

DNA microarray data analysis using Bioconductor

Jarno Tuimala

CSC, the Finnish IT center for Science

CSC – IT Center for Science Ltd. is a non-profit organization for high-performance computing and networking in Finland. CSC is owned by the Ministry of Education. CSC runs a national large-scale facility for computational science and engineering and supports the university and research community. CSC is also responsible for the operations of the Finnish University and Research Network (Funet).

ISBN 978-9525520-34-7

http://www.csc.fi/oppaat/R/

Preface

In this book we concentrate on analysing gene expression data generated using DNA microarrays. Microarrays generate large amounts of numeric data that should be analyzed effectively. R statistical software and its expansion packages from Bioconductor project provide flexible means to manage and analyze these data. Hence, it was selected as the analysis tool for this book.

This book is intended for researches who are involved in DNA microarray data analysis. It can also be used for teaching the basics of microarray data analysis. Much of the material has indeed been adapted from the R and Bioconductor courses we have given at CSC. However, theoretical background on the data analysis and statistics are not extensively covered here, because a separate book published by CSC - DNA microarray data analysis - covers those topics more thoroughly. Readers interested in getting a theoretical background about the analysis are directed to read that book first.

Structure of the book

This books is divided into several parts. *Introduction* covers R installation and basics of R language and graphics. *Preprocessing* discusses topics of preanalysis preparations, such as information assembling, normalization of different data types, and filtering. *Analysis* part emphasizes data analysis techniques, such as statistical testing, and clustering. *Extras* contains material that is not probably interesting to everybody, such as how to use R on CSC's servers.

The data analysis part of the book starts by going through the preprocessing of Affymetrix, Agilent and Illumina data. After preprocessing the data is stored in a uniform format, so that the following chapters are not specific to any of the data types. The order of the chapters is the possible order of the data analysis steps: preprocessing, quality control, filtering, statistical analyses, annotation and finally visualization.

How to read this book

The main emphasis of this book is on extensive R code examples.

The code examples are laid out in the `teletype` font either inside the main text or on separate lines:

```
# Saves numbers in a vector
a<-c(1:10)
```

Lines starting with the hash-sign (#) are comments, and are not executed in R. They are written inside the chunks of code simply for code documentation purposes. If you're testing the code examples on your own, it is not necessary to type these comment lines in R.

Output from R is laid out with a small `teletype` font:

```
>a

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

R commands inside text are typeset with teletype, and are always associated with brackets: `q()`. The names of the objects in R memory and command arguments are also typeset with teletype, but without brackets: `genes`.

Throughout this book we use the following convension for saving the data from the various analyses. Object `dat` always holds the raw data. Once the data is normalized, it is saved into an object `dat2`. The normalized expression values are stored in an object `dat.m`. When the normalized data is filtered or analyzed using some statistical test, the resulting data is saved into object `dat.f` and `dat.s`, respectively.

I suggest readers to try out the code examples themselves. While testing the examples, it is especially helpful to break down the nested commands into individual commands, and run them one by one. Using this technique it is possible to get a better insight into how the commands work, and what kind of input they require. It is also a good idea to consult help pages for all new commands one comes across while reading this book. Keeping these simple directions in mind, we hope that this book could also successfully work as a self-learning material.

Acknowledgements

I am in debt to all students who have attended the R and Bioconductor courses arranged at CSC and Finnish universities during the last few years. Without the pressure to prepare the course material this book might never have come to existence. I also want to thank my colleagues who have teached

on the mentioned courses.

I am very interested in receiving feedback about this publication. Especially, if you feel that some essential technique has not been included, let me know. Please send your comments to the e-mail address Jarno.Tuimala@csc.fi.

Espoo, 22nd October 2008

Jarno Tuimala

Contents

Part I

Introduction

1 Installation of R and Bioconductor

1.1 What are R and Bioconductor?

R is a programming environment specifically tailored to suit statisticians. R is modular, which means that users can easily write extensions and distribute the codes to other users. Most of the functionality in R is in the extension packages written, not by the core developers, but by other users. Bioconductor is a development project that aims to offer tools for genomic data analyses. The emphasis of the project is on DNA microarray data analysis.

1.2 Downloading R

R home page is located at `http://www.r-project.org`. The main site is mirrowed in several locations around the world, but one of the closest sites to Finland is located in Sweden at `http://ftp.sunet.se/pub/lang/CRAN/`.

The precompiled binary distributions for Linux, Mac OS X and Windows are available. In addition, source codes for compilation to other systems are available. The setup program is located under the base folder. This setup program should be downloaded to the computer. It installs the R base, and a few recommended packages. Other packages are available for downloading in the contrib folder. Consult the following sections for detailed installation instruction for different systems.

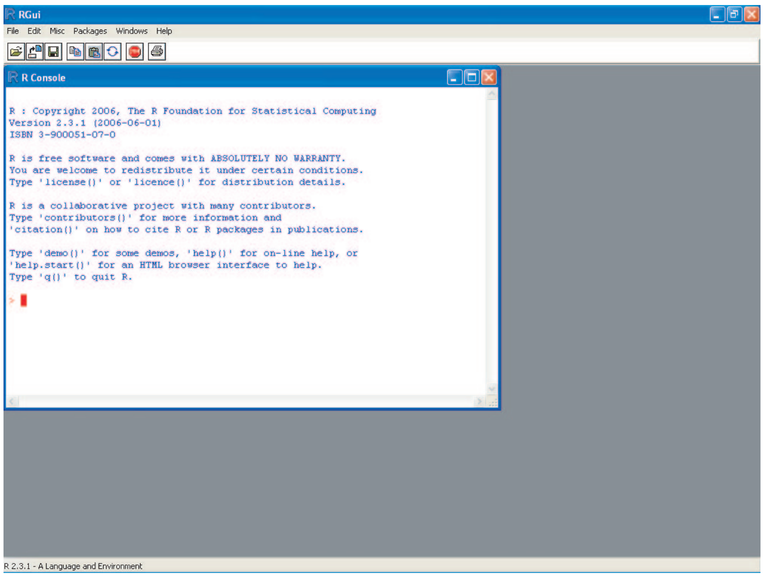
1.2.1 R installation instructions for Windows

Setup program is best run with administrator privileges. License should be accepted, and a suitable installation folder selected. R shortcut can be placed on the Desktop - this would make starting the program easier.

If there is more than 1Gb of memory available on the computer with R, it is possible to configure R to use it. After right-clicking with mouse over the R icon (on the Desktop), properties should be selected from the pop-up window. A new window with detailed information should open. Target field should be edited: Adding `--max-mem-size` to the end of the field enables R to use more memory. For example, the target field might become something like (if there's 2 GBs of memory on the computer):

```
"C:\Program Files\R\R-2.7.2\bin\Rgui.exe" --max-mem-size=1.5Gb
```

R can now be started by double-clicking on its icon on Desktop. If installation is successful, the user interface should start:


RGui
File Edit Misc Packages Windows Help
R Console
R 2.3.1 - A Language and Environment

```
R : Copyright 2006, The R Foundation for Statistical Computing
Version 2.3.1 (2006-06-01)
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

1.2.2 R installation instructions for UNIX and Linux

The downloaded source code distribution should be moved to the correct folder before starting the installation. To install R, type the following UNIX commands:

```
    gzip -d R-2.7.2.tar.gz
    tar xvf R-2.7.2.tar
    cd R-2.7.2
    ./configure
    make
    make check
```

Note that the first three commands are specific to the R version. Here, the commands apply for the version 2.7.2, but if you're using a different version, you need to change the file names.

First, the R source code distribution is unpacked. This creates a new folder. The configuration script is run in this folder, and R is installed. Last, the installation is checked. After a successful checking, Bioconductor and possibly other extra packages can be installed.

The installation above creates a 32-bit R, but if a 64-bit machine is available, it might be better to install a 64-bit version of R. It would allow users to allocate more memory for their jobs. The UNIX commands are just slightly tweaked (more options are available in the R administration manual):

```
gzip -d R-2.7.2.tar.gz
tar xvf R-2.7.2.tar
cd R-2.7.2
./configure 'CC=gcc -m64' 'F77=g77 -m64' 'CXX=g++ -m64'
make
make check
```

1.2.3 Installation instructions for Bioconductor

After a successful R installation, Bioconductor should be installed. Bioconductor can most easily be installed using a ready-made installation script. First, download the script. Give the following command in R:

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

Next, install the basic packages for Bioconductor:

```
> biocLite()
```

Test whether the installation was successful by typing:

```
> library(affy)
```

If the installation was successful, the following message should appear:

```
Loading required package: Biobase
Loading required package: tools

Welcome to Bioconductor

    Vignettes contain introductory material. To view, type
    'openVignette()' or start with 'help(Biobase)'. For details
    on reading vignettes, see the openVignette help page.

Loading required package: affyio
```

If you can't access Internet directly from R, you can download the packages from the Bioconductor site at `http://www.bioconductor.org`. Note the default packages from the download page, and download all of those. Then consult the section Installing extra packages from ZIP files below.

1.2.4 Installing extra packages on any system

All R and Bioconductor packages are stored in the Internet in a place called a repository. CRAN is the central repository for general R packages, but Bioconductor packages are stored in a repository of their own. Packages from CRAN or Bioconductor repositories can be installed directly from R.

First the repository should be selected in R. Type:

```
> setRepositories()
```

This gives a list of available repositories. At least one has to be selected.

Next, the packages to be installed should be specified:

```
> install.packages()
```

One mirror (e.g., Austria) should be selected from the list, and all the desired packages from the second list. Typically, install.packages() opens a pop-up window where this selection can be made.

Bioconductor also contains many annotation packages, such as KEGG pathways and GO ontology. Available package names can be checked from the Bioconductor site at `http://www.bioconductor.org/download/metadata/` . The latest release should be selected from the list of releases. The names of the packages are exactly as they appear in the list on the webpage. For example, to download and install KEGG, GO and cMAP pathway packages, type in R:

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("KEGG", "GO", "cMAP"))
```

An unlimited number of packages can be installed at the same time using this syntax. Just add the names of the new packages enclosed with quotation marks and separated by comma (,).

Annotation for all Affymetrix chips are available from the Bioconductor site. These can be downloaded and installed as specified above. For example, to install all yeast ygs98 chip related files, type:

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite(c("ygs98", "ygs98cdf", "ygs98probe"))
```

These packages contain the annotations for the probe sets (ygs98), mappings between probes and probe sets (ygs98cdf), and probe sequences (ygs98probe). Packages for other Affymetrix chip types are similarly named, but the naming scheme for other technologies differs from this.

1.2.5 Installing extra packages from ZIP files

If you can't connect to Internet directly from R, packages can be downloaded to the local computer, and be installed from these files. For windows packages come as ZIP-files, and for UNIX systems as .tar.gz-files. These should *not* be expanded before installation.

In windows a ZIP-package can be installed from the user interface menu *Packages* using the selection *Install package(s) from local zip files*. For UNIX systems the installation is slightly more demanding. Command `install.packages()` is used for this. For example, to install a package called amap, the command would be:

```
> install.packages(repos=NULL, pkgs="amap_0.7-3.tar.gz)
```

Before running the installation script, you need to change the R path to point to the folder where the package can be found. This is accomplished with the command `setwd()`, where the path goes inside the brackets surrounded by quotation marks, e.g., `setwd("/etc/home")`.

1.3 Graphical user interfaces

There are several graphical user interfaces to R. These range from text editors integrated with R to graphical point and click interfaces. Only TinnR, a code editor, is introduced here.

1.3.1 TinnR

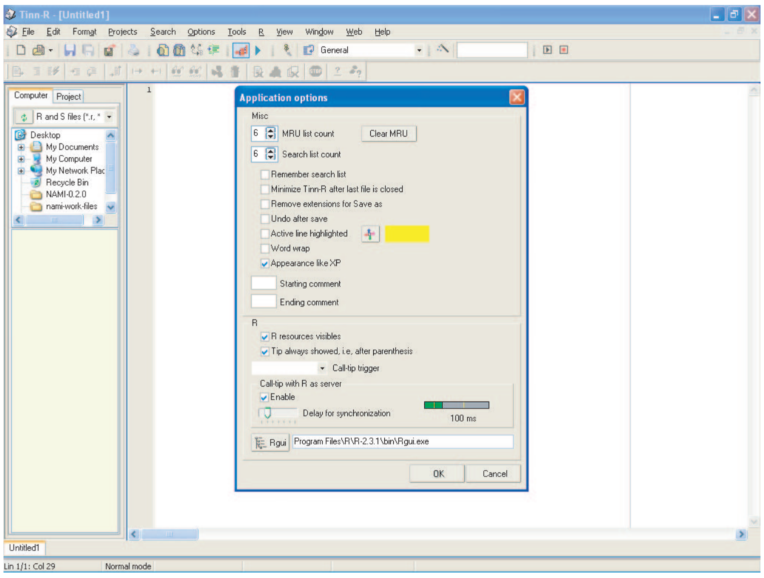
TinnR is a text editor that has a capability to link directly to R. It has been written with Windows systems in mind. There are other text editors for UNIX and Linux system, such as Emacs (and it's extension for communicating with R, ESS).

TinnR can be download from `http://www.sciviews.org/Tinn-R/index.html`. The current version of TinnR is compatible with R run in SDI (single document interface) mode only. After downloading and installing R and TinnR, right click with mouse over the R icon (on Desktop). Select properties from the pop-up window. A new window opens. Add `--sdi` to the end of the target field so that the field becomes something like

```
"C:\Program Files\R\R-2.7.2\bin\Rgui.exe" --sdi
```

Now R is configured to run in SDI mode.

TinnR needs to know where R is located on the computer. The path to R can be specified from the *Options*-menu, selection *Main* -> *Application*. In the bottom of the opening setting window is a field called Rgui, where the path to R should be typed. After typing in the path, select OK:



Misc
6 MRU list count
6 Search list count
☐ Remember search list
☐ Minimize Tinn-R after last file is closed
☐ Remove extensions for Save as
☐ Undo after save
☐ Active line highlighted
☐ Word wrap
☑ Appearance like XP
Starting comment
Ending comment

R
☑ R resources visibles
☑ Tip always showed, i.e, after parenthesis
Call-tip trigger
Call-tip with R as server
☑ Enable
Delay for synchronization 100 ms
Rgui Program Files\R\R-2.3.1\bin\Rgui.exe

If you're not sure how to find the correct path, open some file browser (e.g., Windows Explorer), and browse to the R installation directory. Then coyp and paste the path from the file browser to Tinn-R settings. Youn can check that you specified the correct path by testing the connection between TinnR and R. Selection *Initiate/Close Rgui* -> *Initiate preferred Rgui* should start R.

TinnR can color the written code. The color scheme can be changed to R from menu *Options* with selection *Syntax* -> *Set* -> *R*. Now Tinn R is ready for use.

R code can be written directly to the TinnR editor. Written code can be run in R directly from TinnR. The part of the code to be run should first be colored. The code can then be run in R by selecting *Send to R* -> *Selection* from the menu *R*.

2 Introduction to R language

2.1 Basics of R language and environment

Starting the work with R without any previous knowledge of possible commands is rather demanding. This chapter offers an overview of basics of R language and some of the most common R commands.

2.1.1 Starting and closing R

R can be started either by clicking its icon (Windows and Mac OS X) or by typing its name on the command prompt (Linux and UNIX).

R is closed with the command `q()`. When R is closed, the user has an option to save everything done in the same session. R asks whether to "Save workspace image". If answered yes, all the objects in R memory are saved in a file called .RData, and the command history is saved in a file called .Rhistory. This is very helpful, since these files can later on be loaded into R memory again, and the analysis can be continued. In windows these files can be loaded from menu *File* -> *Load workspace* and *File* -> *Load history*. In UNIX, Linux and Mac OS X the files can be loaded using the command `load`, for example:

```
> load(".RData")
```

2.1.2 Prompt

When R starts user gets to the prompt. In the R environment user can type commands on the line starting with larger than sign (>). This sign is called a prompt, because it prompts or urges user to write something. Sometimes lines start with a plus (+) sign. This means that the input on previous lines was not complete, and R expects to get more input. This can happen when the given commands were incomplete, possibly missing some elements of the command, and when the command is typed on several lines, as is the case with loops and conditional execution of commands (see more below).

2.1.3 Help!

R ships with comprehensive help files. These can be accessed directly from R or using a web browser. The web browser is especially suitable for searching new possibly usable commands. Direct access from R is faster, but the name of the command is needed in order to access the help files this way.

Web browser help can be invoked with the command:

```
> help.start()
```

This opens a new browser window. A basic book on R language can be accessed there through the link *An introduction to R*. Other basic manuals are also included as web-pages. Packages-link gets to a detailed description of functions and datasets in each package. Search Engine can be used for searching commands and datasets by quory words.

Help can be accessed directly from R using the command `help()`. The name of the command is written in parenthesis:

```
> help(mean)
```

The command opens the help page in R. Typically in a Windows system a new window is opened for the help page. In UNIX the help page is displayed in the editor, and pressing `q` exits the help page.

The general layout of the help page is as follows. The first line gives the name of the command, and the package it can be found from. Then comes the title of the function or the command. Command `mean()` calculates arithmetic mean. After the title comes a short description and the general usage of the command. Command `mean()` takes several arguments (`x`, `trim` and `na.rm`) that are described in more details under the arguments part of the help page. After arguments comes a description of the values generated by the command. There are typically some examples of correct usage in the end of the help page.

```
mean                  package:base                  R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:
       x: An R object.  Currently there are methods for numeric data
          frames, numeric vectors and dates.  A complex vector is
          allowed for 'trim = 0', only.

    trim: the fraction (0 to 0.5) of observations to be trimmed from
          each end of 'x' before the mean is computed.

   na.rm: a logical value indicating whether 'NA' values should be
          stripped before the computation proceeds.

     ...: further arguments passed to or from other methods.

Value:

     For a data frame, a named vector with the appropriate method being
     applied column by column.

     If 'trim' is zero (the default), the arithmetic mean of the values
     in 'x' is computed, as a numeric or complex vector of length one.
     If any argument is not logical (coerced to numeric), integer,
     numeric or complex, 'NA' is returned, with a warning.

     If 'trim' is non-zero, a symmetrically trimmed mean is computed
     with a fraction of 'trim' observations deleted from each end
     before the mean is computed.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.

See Also:

     'weighted.mean', 'mean.POSIXct'

Examples:

     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))

     mean(USArrests, trim = 0.2)
```

2.1.4 Commands

R commands are always case sensitive and associated with parenthesis immediately following the command. As we saw in the Help-section above, command `mean()` calculates an arithmetic mean of an R object (`x`) that has been given to the command. For example, age of ten individuals can be stored in a variable called age, and a mean age for the group can be calculated with function `mean()`:

```
> mean(age)
```

The function `mean()` always expects to get a name of a variable containing the values for calculation of the arithmetic mean. Thus, each command consists of three parts:

- command name
- parenthesis
- argument(s) within parenthesis

In other words, age is a value assigned to to the argument x for the command `mean()`. As described in the help page for `mean()` an argument `na.rm` can also be specified. This argument is a logical value (either TRUE or FALSE) indicating whether to remove missing values from the variable before calculating the mean. If the argument gets a value of TRUE, missing values are removed before calculations. If the value is FALSE, missing values are not removed. For example, mean age with missing values removed can be calculated as:

```
> mean(age, na.rm=TRUE)
```

As already mentioned, R commands are case sensitive. Thus, the following would not work, because function `MEAN()` is not recognized as a valid R command:

```
> MEAN(age, na.rm=TRUE)

Error: could not find function "MEAN"
```

It is not always easy to guess what arguments each command takes. In such cases, it is best to consult the help page for the function, and read the arguments part of the help page meticulously. In addition, it is often helpful to read and test the examples from help page.

In general, for an argument that take numerical or logical values, the arguments are given without quotation marks. For arguments taking strings, values need to be given within quotation marks. For example, command `mean()` does not need any string input, and all its arguments are specified without quotation marks:

```
> mean(age, na.rm=T, trim=0.1)
```

Commands can also be tiled, i.e., commands can be written one after another on the same line as long as the number of parenthesis is balanced. For example, missing values can first be removed, and then the mean of the values can be calculated:

```
> # Removes missing values from age
> age2<-na.omit(age)
> # Calculates mean of age2 (no missing values)
> mean(age2)
> # The same procedure but written
> # on the same line
> mean(na.omit(age))
```

2.1.5 Environment

R environment has some helpful functionality. Previous commands can be leafed through using the up and down arrow keys. This functionality is especially handy when there is a long command that was mistyped. Pressing once the up arrow key gets back to the last command, and it can be edited without a need to retype the whole command.

It is also possible to get a list of all the previous commands executed in R using the command `history()`. By default it displayes only of a few dozen lastest commands, but the number of commands displayed can be modified using argument `max.show`. For example, the following command shows the last 500 commands:

```
> history(max.show=500)
```

2.1.6 Packages

Additional packages can be installed in R as described in the Installation chapter of this book. The installed packages are not by default loaded into memory everytime R is started. Instead the user should load them into memory using the command `library()`. For example, package for classification and regression tree analysis (`rpart`) can be loaded in the memory:

```
> library(rpart)
```

To get a list of all the packages loaded into memory, use function `sessionInfo()`:

```
> sessionInfo()

Version 2.3.1 (2006-06-01)
i386-pc-mingw32

attached base packages:
[1] "methods"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "base"

other attached packages:
   rpart
"3.1-29"
```

If the package is not loaded into memory the commands or datasets contained in the package cannot be used, and help functions in R do not find documentation for the functions. For example, `help(rpart)` would give the following error message if the package rpart is not loaded into memory:

```
No documentation for 'rpart' in specified packages and libraries:
you could try 'help.search("rpart")'
```

Sometimes two packages contain functions with exactly the same name. This can trigger a warning:

```
Attaching package: 'rpart'


        The following object(s) are masked _by_ .GlobalEnv :

         rpart
```

This warning simply means that the command `rpart()` in package `rpart` cannot be used, because there is already another command with the same name in memory. This can fixed by removing the conflicting package from the memory using the command `detach()`:

```
> detach(package:rpart)
```

Or, better still, restaring R.

2.1.7 Changing the working directory

It is a good practice to dedicate a single folder for a single dataset. Whenever starting to work with a new dataset, it should be copied to a new folder. Before data importing, R should be told where the data resides, and this can be accomplished in Windows using the menu *File* -> *Change Dir*. This opens a new windows where the user can browse to the correct folder. In UNIX and Linux the user needs to specify the path to the data folder using the command `setwd()`. For example, to set the path to the timeseries data folder on CSC server Corona, user could issue a command:

```
> setwd("/home/csc/jtuimala/timeseries")
```

Sometimes it is useful to check the current working directory. The path to the working directory can be checked using the command `getwd()`:

```
> getwd()
[1] "/home/csc/jtuimala/timeseries"
```

Files in the current folder can be listed using the command `dir()`:

```
> dir()
[1] combined.txt
```

2.2 R is an expanded calculator

R is an excellent calculator. It has a very comprehensive collection of mathematical and arithmetic functions. The standard order of precedence is used in R. Powers and roots are calculated first, followed by multiplication and division, and last addition and subtraction. Some of the most commonly used arithmetic and mathematical features of R are introduced here.

2.2.1 Arithmetic

Addition and subtraction are done using operators + and -:

```
> 9+5
[1] 14
> 9-5
[1] 4
```

Multiplication is done using asterisk (*):

```
> 4*6
[1] 24
```

Division uses slash (/):

```
> 16/4
[1] 4
```

Exponentiation uses the hat ($\wedge$):

```
2^2
[1] 4
```

For square root, there is a command `sqrt()`:

```
> sqrt(4)
[1] 2
```

2.2.2 Mathematical functions

In addition to arithmetic functions, all sorts of other mathematical functions (commands) are available, most importantly logarithms and exponentiation.

Logarithm are calculated using the command `log()`:

```
log(4)
[1] 1.386294
```

The `log()` -command gives natural logarithms (base 2.718282). Logarithms to base 10 can be calculated using the function `log10()`, and logarithm to base 2 with function `log2()`. Command `exp()` calculates an exponential (antilog) function (2.718282^x):

```
> exp(2)
[1] [1] 7.389056
```

2.2.3 Logical arithmetic

Testing for equality of two objects can be done using the operator ==, and the result is a logical expression (either true or false):

```
> 2==2
[1] TRUE
```

Other possible operators are >, <, >=, <=, and !=. These test whether the first objects is larger than, smaller than, larger or equal to, smaller or equal to or unequal to the last objects. For example:

```
> 2<3
TRUE
> 2>3
FALSE
```

2.2.4 Number of decimals

R makes the calculations using more decimals than are reported in the output. Therefore, the number of decimals in the output can limited. This can be accomplished by rounding the already acquired numbers to a certain number of decimals (command `round()`).

```
> round(exp(2), digits=2)
[1] 7.39
```

2.3 Data input and output

Data can be read into R in several ways. The simplest possibility is to input data using functions provided by R. Another, probably less laborous way for larger datasets, is to read the data from a table. The table can be created in any spreadsheet editor, such as Microsoft Excel, and saved in a tab-delimited text format. Similar tables are typically produced by microarray image analysis software. Both data input types are introduced here.

R assumes by default that the decimal delimiter for numeric values is dot (.). Comma (,) is a delimiter for lists. Therefore, all the numerical values that are imported into R should be checked to contain only numbers and dots. Otherwise the values are read in as text. However, there is way around this using the command `read.table()` (see below).

2.3.1 Allocation

Allocation is acquired using the operator <- that is made up from the smaller than sign coupled with the minus sign. Allocation is used for storing information into some named object in R memory. Allocation destroys the existing object of the same name.

Saving a single number to an object called `number` is acquired as follows:

```
> # This works, but a better way would be
> # to use the command c(), see below
> number<-2
```

Names of the objects are regulated. They cannot start with a number or contain special letters, such as å, ä or ö. If the variable name is long, it is better to separate the parts with dot (.) not with blanks ( ), minus (-) or underscore (_). It is also best to avoid using command names as object names.

2.3.2 Typing in the data

R typically thinks in vectors. A vector is an ordered list of numbers or strings. Vector is the simplest type of data structure available in R.

A vector is created using the command `c()`. The numbers that form the vector are typed inside the parenthesis and separated by comma (,). Usually the vector is saved in an object with assignment operator <-:

```
> a<-c(1,2,3,4,5)
> a
[1] 1 2 3 4 5
```

Note that objects can be viewed by typing their name on the command prompt.

2.3.3 Reading tabular data

Tabular data, such as tab-delimited text files, can be easily read into R using the command `read.table()`. It should be provided with the name of the file (inside quotation marks), a logical value indicating whether the table columns have titles, the separator used in the file, and optionally the number of the column having the row names:

```
> dat<-read.table("ab.txt", header=TRUE, sep="\t", row.names=1)
> dat
  a  b
a 1  6
b 2  7
c 3  8
d 4  9
e 5 10
```

Here, a tab-delimited text file containing two columns named `a` and `b` was read in an R object `dat`.

If the decimal separator is something else than dot (.), it can be specified in the command `read.table()` using the argument `dec`. For, example, the following command would read in a rather rare pipe-separated (|) file:

```
> dat<-read.table("ab.txt", header=TRUE, sep="\t", row.names=1,
+ dec="|")
```

2.3.4 Writing tabular data

Tabular data can be written into a file using the command `write.table()`. The command takes as arguments the name of the data object to be written to the file, file name inside quotation marks, and the type of the file separator (here, tab). For example:

```
> write.table(ab, "ab.txt", sep="\t")
```

2.3.5 Saving output to a file

Output generated by R commands can be saved to a file. Before any commands are run, command `sink()` is issued. It takes as an argument a file name where the output is saved. After the commands are executed, `sink()` is issued again, and any further output is not saved to a file anymore. For example:

```
> sink("result.txt")
> print(ab)
> print(summary(ab))
> sink()
```

2.4 Calculations with vectors

All the functions introduced above for numbers are applicable for vectors, also. When applied to a vector, the function is applied on every number of the vector individually. In addition, there are several commands that can be applied meaningfully only to vectors.

2.4.1 Arithmetic on vectors

Basic arithmetic operators can be applied for vectors:

```
> x; y
[1] 0 1 0 0 1 0 0 1 0
[1] 0 0 1 0 0 1 0 0 1
> x+y
[1] 0 1 1 0 1 1 0 1 1
```

2.4.2 Mathematical operators for vectors

The most commonly used commands for vectors might be `sum()` and `mean()` that calculate a sum or an arithmetic average of the numbers of the vector:

```
> sum(x)
[1] 3
mean(x)
[1] 0.33
```

Commands `min()`, `max()` and `range()` give the minimum, maximum and range of the numbers in a vector:

```
> min(x)
[1] 0
> max(x)
[1] 1
> range(x)
[1] 0 1
```

Basic statistics for a vector are generated by the command `summary()`:

```
> summary(x)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   0.000   0.000   0.333   1.000   1.000
```

Standard deviation, variance and correlation are calculated as:

```
> sd(x)
[1] 0.5
> var(x)
[1] 0.25
> cor(x,y)
[1] -0.5
```

2.5 Object types

The object types introduced below are the basic object types that are available in R. The object type can be found out using the command `class()`. This is important, since not all commands or functions are available for all object types. Some statistical commands generate large objects. To get a list of all elements contained in the object, command `str()` should be used. To demonstrate these:

```
> class(x)
[1] "numeric"
> class(dat3)
[1] "matrix"
> str(dat3)
 num [1:2, 1:9] 0 0 1 0 0 1 0 0 1 0 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:2] "x" "y"
  ..$ : NULL
```

Here, command `str()` gives a detailed description of a matrix that contains 2 columns and 9 rows. That information is given on the first line of output from `str()`. The second line says that there are names for the dimentions of the matrix, and those names are then listed. Columns have names x and y, but rows do not contains names, and the names are listed as NULL (no data).

2.5.1 Vector

A vector is an ordered list of numbers or strings. Vectors have been discussed above in details, and the same information is not duplicated here.

A function not yet introduced, but that might be of use is `length()`. It calculates the lenght (number of values) of a vector:

```
> length(a)
[1] 9
```

2.5.2 Factor

A factor is a vector that is used for grouping components of other vectors. Factors are often used for storing categorical data. Therefore, they are typically used in statistical models, such as linear regression models.

Vectors can easily be converted into factors using the command `as.factor()`:

```
> xf<-as.factor(x)
> xf
[1] 0 1 0 0 1 0 0 1 0
Levels: 0 1
```

Note that by default, `as.vector()` converts the vector to a factor with the same levels that were present in the vector. If argument `mode="numeric"` is used, the factor levels in ascending order are recoded with numbers starting from 1.

2.5.3 Matrix

A matrix is a table of *n* rows times *m* colums. All columns of a matrix should contain similar information. For example, all the columns of a matrix could contain numbers, but numbers and strings can not be mixed in the same matrix. For example:

```
> dat2
      x y
 [1,] 0 0
 [2,] 1 0
 [3,] 0 1
 [4,] 0 0
 [5,] 1 0
 [6,] 0 1
 [7,] 0 0
 [8,] 1 0
 [9,] 0 1
> class(dat2)
[1] "matrix"
```

Every value in a matrix can be accessed using a subscript. Subscript indicates the row and column of the observation to be accessed:

```
> # The first row and the first column
> dat2[1,1]
[1] 0
> # The first row only
> dat2[1,]
x y
0 0
> # The first column only
> dat2[,1]
[1] 0 1 0 0 1 0 0 1 0
> # The rows from 1 to 3
> dat2[1:3,]
     x y
[1,] 0 0
[2,] 1 0
[3,] 0 1
```

Subscripts can be used for getting a subset of a dataset. There are several ways to do this, but only a couple are introduced here.

```
> a
[1]  0  1 11  0  1 11  0  1 11
> # Gives only the rows of dat2 for which
> # a is equal to 11
> dat2[a==11,]
     x y
[1,] 0 1
[2,] 0 1
[3,] 0 1
> # Command which gives a list of index numbers
> # indicating the rows of the matrix.
> # This can be used for getting a subset
> # of the data.
> which(a==11)
[1] 3 6 9
> dat2[which(a==11),]
     x y
[1,] 0 1
[2,] 0 1
[3,] 0 1
```

The dimensions of a matrix can be checked using the command `dim()` that prints two numbers. The first number is the number of rows, the seconds number being the number of columns:

```
> dim(dat2)
[1] 9 2
```

2.5.4 Data frame

A data frame is a table, where different columns can contain different kinds of information. In contrast to matrices, data frames can contain a mix of columns having numbers and strings.

Data frames can be subscripted as described above for matrices. In addition data frames can contain names for columns and rows. These can be found out using the commands `colnames()` and `rownames()`, respectively.

A data frame can be created, e.g., from two vectors:

```
x<-c(0,1,0,0,1,0,0,1,0)
y<-c(0,0,1,0,0,1,0,0,1)
dat2<-data.frame(x, y)
```

Names of a data frame can be easily changed. For example:

```
> dat2
      x y
 [1,] 0 0
 [2,] 1 0
 [3,] 0 1
 [4,] 0 0
 [5,] 1 0
 [6,] 0 1
 [7,] 0 0
 [8,] 1 0
 [9,] 0 1
> names(dat2)
[1] "x" "y"
> row.names(dat2)
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
> # Setting the names for rows
> # Naming each row using a letter
> l<-letters[1:9]
> l
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
> row.names(dat2)<-l
> row.names(dat2)
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

In addition to subscripting, each column of a data frame can be accessed using its name. The name is written after the data frame name followed by a dollar sign ($):

```
> dat2$x
[1] 0 1 0 0 1 0 0 1 0
```

An existing matrix can be converted to a data frame with a specific conversion command:

```
dat2<-as.data.frame(dat2)
```

2.5.5 S3/S4 class

Some objects, such as AffyBatch objects created using certain functions from affy-package developed by Bioconductor project create S3 or S4 class structures. These class structures store data in slots. Names of the slots can be checked using the command `str()`. Data in slots can be accessed using the operator `@`. For example `dat@cdfName` would show the data in the slot `cdfName` of object `dat`.

2.6 Data manipulation

2.6.1 Generating sequences of numbers

The simplest way to generate a sequence of numbers is by using the colon operator (:). When colon separates two numbers, a regularly spaced sequence of numbers is generated:

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

More complicated sequences are generated with command `seq()`. It takes three arguments. The first indicates the start of the sequence, the second the end of the sequence, and the third the increment or decrement to use. For example:

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(5, 1, -0.5)
[1] 5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

2.6.2 Generating repeats

Repeats can be generated easily with the command `rep()`. Repeats are an easy way to generate new variables for statistical models. Command `rep()` takes two argument, what to repeat and how many times:

```
> # Repeat number 1 five times
> rep(1, 5)
[1] 1 1 1 1 1
> rep("Hello!", 5)
[1] "Hello!" "Hello!" "Hello!" "Hello!" "Hello!"
> # Repeat number 1-3 two time
> rep(1:3, 2)
[1] 1 2 3 1 2 3
> # Repeat numbers 1-3 each two times before
> # proceeding to the next number
> rep(1:3, each=2)
[1] 1 1 2 2 3 3
```

2.6.3 Searching and replacing

Searching and replacing is most easily done using the command `ifelse()`. It takes three arguments, a logical comparison, the value to return if the comparison is true, and the value to return if the comparison is false. Command `ifelse()` can be used, for example, for recoding variables. Here, the original variable is recoded into two dummy contrast variables:

```
> a<-rep(c(0,1,11), 3)
> a
[1]  0  1 11  0  1 11  0  1 11
> # If a is equal to 1, return 1,
> # otherwise return 0
> x<-ifelse(a==1, 1, 0)
> # If a is equal to 11, return 1,
> # otherwise return 0
> y<-ifelse(a==11, 1, 0)
> # Prints first x, then y
> x; y
[1] 0 1 0 0 1 0 0 1 0
[1] 0 0 1 0 0 1 0 0 1
```

It is a good practise to check that the recoding works as it was supposed to work. This can be done using the command `table()` that calculates a cross-tabulation of two vectors provided for it:

```
> table(a, x)
    x
a    0 1
  0  3 0
  1  0 3
  11 3 0
> table(a, y)
    y
a    0 1
  0  3 0
  1  3 0
  11 0 3
```

2.6.4 Merging tables

Merging two tables is done with command `merge()`. Command takes two obligatory arguments, names of the tables. It is often necessary to specify the column by which matching of elements is done. Columns are specified using options `by.x` and `by.y`. For example, merging two tables might proceed as follows:

```
> g<-data.frame(y=1:5)
> # Naming rows with some letters
> row.names(g)<-c(letters[1:2], letters[7:9])
> g
  y
a 1
b 2
g 3
h 4
i 5
> h<-data.frame(x=6:10)
> # Naming rows with some letters
> row.names(h)<-letters[1:5]
> h
   x
a  6
b  7
c  8
d  9
e 10
> # Matching is done using the row names
> merge(g, h, by.x="row.names", by.y="row.names")
  Row.names y x
1         a 1 6
2         b 2 7
```

Note that only rows with a name a or b are combined, because they exist in both tables `g` and `h`. Other non-matching rows are discarded.

2.6.5 Transposition

Sometimes the data matrix needs to be flipped around so that columns become rows and rows become columns. This is especially useful when certain statistical models are applied to microarray data. Classic statistical tools assume that columns contain variables, and rows individual observations for the variables. In microarray data this is not usually the case, because individual chips constitute columns. Transposition is accomplished using the command `t()`:

```
> # Making a simple table of two vectors
> dat2<-cbind(x,y)
> dat2
     x y
[1,] 0 0
[2,] 1 0
[3,] 0 1
[4,] 0 0
```

```
 [5,] 1 0
 [6,] 0 1
 [7,] 0 0
 [8,] 1 0
 [9,] 0 1
> dat3<-t(dat2)
> dat3
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
x    0    1    0    0    1    0    0    1    0
y    0    0    1    0    0    1    0    0    1
```

2.6.6 Sorting and ordering

Sorting and ordering vectors are basic manipulations that are often very useful. Sorting a vector either in ascending or descending order in done with command `sort()`. Command `order()` returns a permutation vector that will sort the original vector in ascending order. To demonstrate this:

```
> # Generating five integers and a sequence
> # and binding them to a data frame
> i<-data.frame(x=round(abs(c(rnorm(5)*10))), y=c(1:5))
> # Naming rows with letters
> row.names(i)<-letters[1:5]
> i
   x y
a  4 1
b  5 2
c  4 3
d  2 4
e 11 5
> # Returns the values in ascending order
> sort(i$x)
[1]  2  4  4  5 11
> # Returns a vector with which the values
> # can be sorted
> order(i$x)
[1] 4 1 3 2 5
```

The output from `order()` is slightly more demanding to comprehend. The output vector lists the ordering in which the values should taken from the original vector in order to get them into ascending order. In the example above, the fourth value (2) of the original vector would be taken first. The first value would be next in order, and so forth until all the values have been ordered.

Command `order()` can be used for sorting whole tables. Sorting tables uses subscripts as follows:

```
> # Sorting table i according to x
> i[order(i$x),]
   x y
d  2 4
a  4 1
c  4 3
b  5 2
e 11 5
```

Sorting table proceeds in a row wise fashion. As command `order()` gave a permutation vector, the original table's rows are sampled in the order given by the permutation vector, and a new table is returned. This new table is sorted in ascending order according to the values of the vector specified in the command `order()`.

2.6.7 Missing values

Biological data often contains missing values. Those are indicated in R with string NA. Missing values complicate many analyses, such as hierarchical clustering or even calculation of arithmetic average of a vector.

Two simple solutions to the missing value problem are removal and imputation. If missing values are removed from the data, all rows (observations) that contain at least one missing value for at least one variable are removed from the data set. Removal is accomplished using the command `na.omit()`:

```
> # Creating a vector with one missing value
> vwm<-c(1,2,3,NA,5)
> # Removing missing values
> vwom<-na.omit(vwm)
> vwom
[1] 1 2 3 5
attr(,"na.action")
[1] 4
attr(,"class")
[1] "omit"
> # Calculating a mean of the vector
> # If missing values exist, this gives NA as a result
> mean(vwm)
[1] NA
> # Without missing values this succeeds
> mean(vwom)
[1] 2.75
```

Command `na.omit()` works similarly for matrices and data frames, but instead of single values, whole rows are removed.

Missing values can also be replaced with imputing a value for all missing observations. Function `impute()` is available in the library e1071:

```
> library(e1071)
```

There are two possibilities for imputation. Missing values can be replaced with either mean or median of the other observations in the same variable. Note that imputation needs a matrix or a dataframe as an input.

```
> # Creating a matrix
> v<-matrix(1:10, ncol = 2)
> v[1,1]<-NAv[1,]<-NA
> # Mean imputation
> v2<-impute(v, what=c("mean"))
> v2
> v2
     [,1] [,2]
[1,]  3.5    6
[2,]  2.0    7
[3,]  3.0    8
[4,]  4.0    9
[5,]  5.0   10
> # Median imputation
> v2<-impute(v, what=c("median"))
> v2
     [,1] [,2]
[1,]  3.5    6
[2,]  2.0    7
[3,]  3.0    8
[4,]  4.0    9
[5,]  5.0   10
```

2.7 Loops and conditional execution

Loops are control structures that enable execution of the same command or commands several times. For-loop is probably the most general one, and it is introduced below. Conditional execution makes it possible to execute some command only if a certain rule in fulfilled. Conditional execution in R is programmed using commands `if()` and `if()...else`.

2.7.1 for-loop

For-loop requires a name of an index and range it takes on. The name of the index can be any R legitimate object name, although it is best to avoid names of already existing objects and commands. The lines belonging to the loop are typed within curly brackets.

For example, a loop that calculates 2^x for values of x between 1-5, would be writtes as:

```
> # Here i is the index name
> # Values of i range from 1 to 5
> for(i in 1:5) {
+     print(2^i)
+ }
[1] 2
[1] 4
[1] 8
[1] 16
[1] 32
```

2.7.2 if

If evaluates a logical comparison, and if the comparison is true, the commands within curly brackets are executed. For example, to calculate 2^x for values of x between 1-5, only if variable k equals 1, would be programmed as:

```
> k<-c(1)
> if(k==1) {
+     for(i in 1:5) {
+         print(2^i)
+ }
+ }
```

You can test the conditional execution in the example above by changing the value of k to, e.g., 2.

2.7.3 if...else

Sometimes it is desirable to do something else if the condition in the if-clause is not met. For example, 2^2 is calculated only if k equals 1 otherwise 3^2 is calculated:

```
> k<-c(1)
> if(k==1) {
+     print(2^2)
+ } else {
+     print(3^2)
+ }
```

The same procedure can also be implemented using two if-clauses:

```
> if(k==1) {
+     print(2^2)
+ }
> if(k!=1) {
```

```
+    print(3^2)
+ }
```

The last example uses an operator `!=` that means unequal in R language. Sometimes several comparisons need to be made. These can be implemented using operators | and &. The former operator means or, and the latter is and. These are logical operators, and can be used in any commands. For example, the loop above can be modified to print the square of two if k is equal to one or two, and print square of three if k is larger than 2:

```
> if(k==1 | k==2) {
+    print(2^2)
+ }
> if(k>2) {
+    print(3^2)
+ }
```

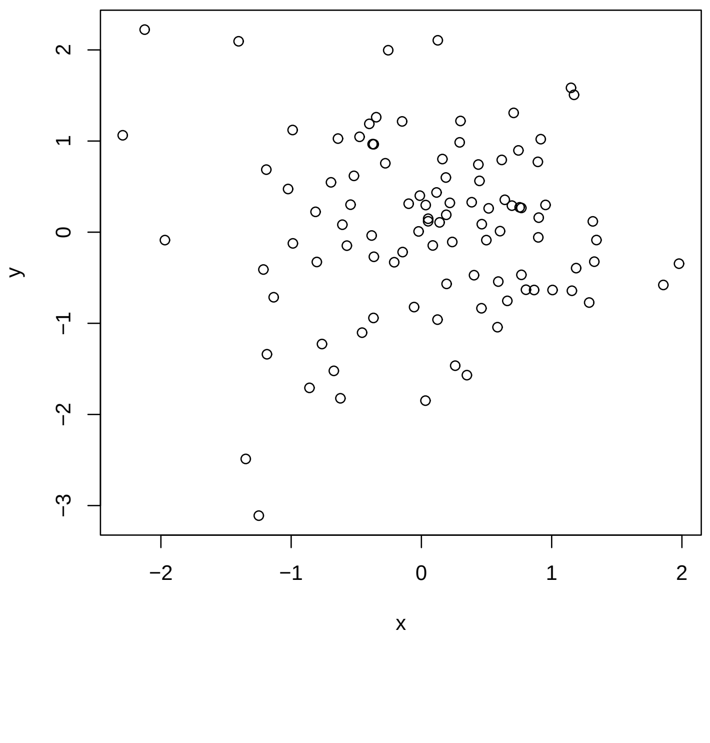
2.8 Graphics

R has very good graphics capabilities. It is only possibly to scratch the surfase of its capabilities here, and the emphasis is on basic consepts of R graphics.

2.8.1 Plot, a general command

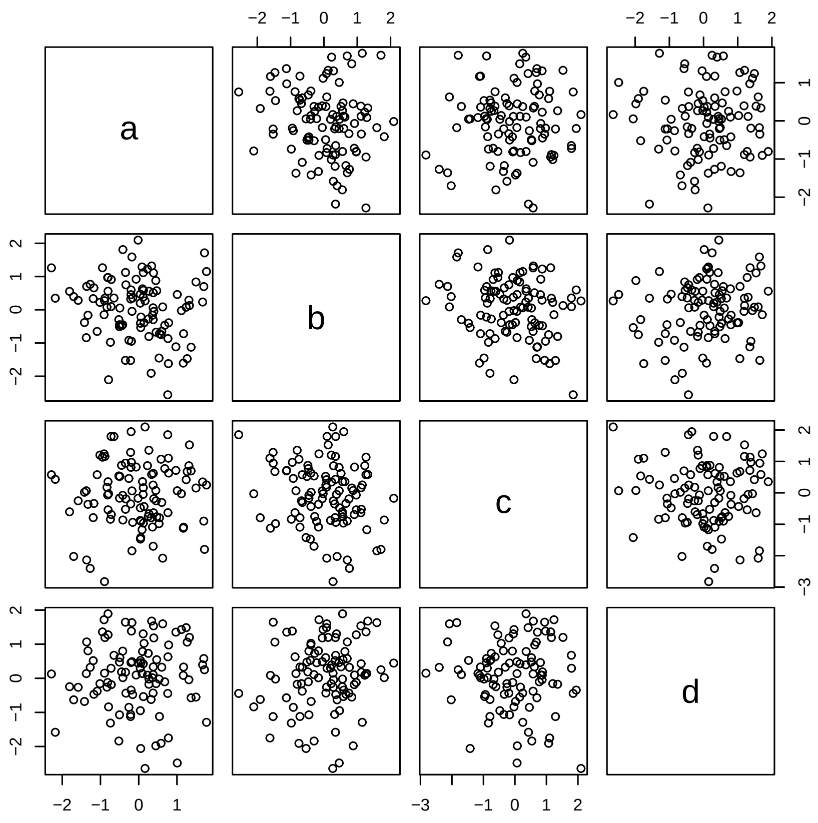
The elementary command of graphics is `plot()`. It can be used for visualizing many different kinds of objects. The image produced by `plot()` is typically a scatter plot. Two vectors are needed for producing a scatter plot:

```
> # Generate two random vectors containing
> # a hundred values from a normal distribution
> x<-rnorm(100)
> y<-rnorm(100)
> # Plot the vectors in a scatter plot
> plot(x, y)
```

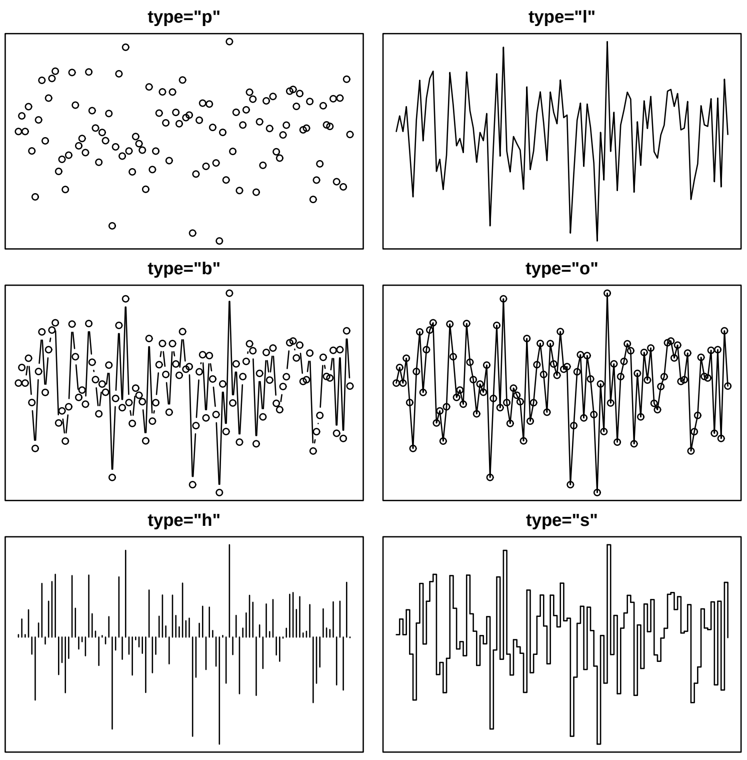


If `plot()` is provided with a table (matrix or data frame), the command defaults to another command `pairs()` that plots all columns aginst each other with scatter plots:

```
> # Makes a data frame with four columns
> # called a, b, c, and d
> dat<-data.frame(a=rnorm(100), b=rnorm(100), c=rnorm(100),
+ d=rnorm(100))
> plot(dat)
```



Command `plot()` takes other arguments, also. The type of the plot can be specified using the argument `type`. The possible options are points (p), lines (l), both points and lines (b), overplotted (o), histogram like bars (h), and steps. These possibilities are illustrated below.



2.8.2 Changing colors and symbols

In addition to the command specific arguments, graphics commands usually accept graphics parameter arguments, and title related arguments. A title can be generated for a plot adding the title arguments inside the graphics' command's parenthesis. For example:

```
> # A scatter plot with main title and x
> # and y axis labels
> plot(x, y, main="Scatter plot", xlab="X variable",
+ ylab="Y variable")
```

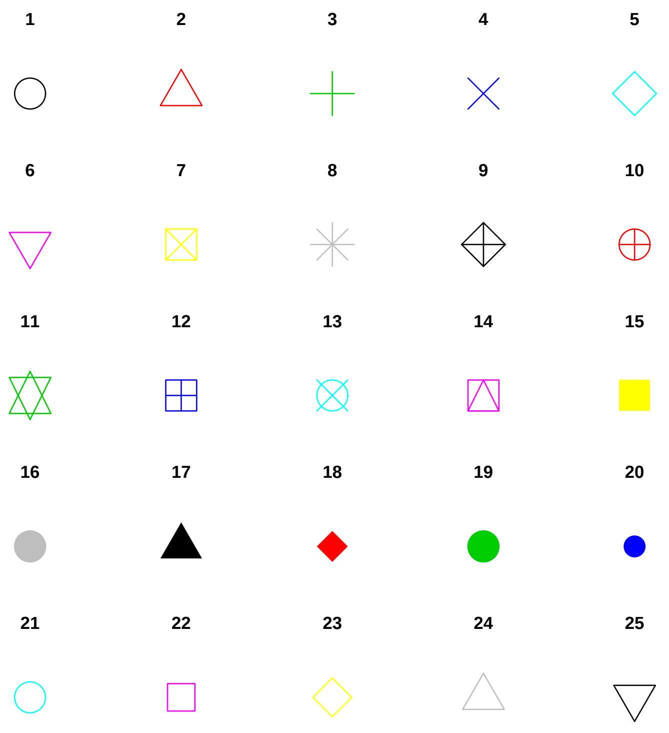
General graphics parameters, such as color for titles and axis labels, and plotting color can be specified before plotting using the command `par()` that is used for setting also many other graphics specific options. It is also possible to give the same information within the graphics' command's parenthesis. Plotting symbol can also be specified as a graphics parameter in the plotting command. For example:

```
> # Generating a vector for colors
> # of the dots
> color<-c(rep(1, 50), rep(2, 50))
> # Argument col specifies the colors
> plot(x, y, col=color)
> # Generating a vector for point
> # types of the dots
> pnt<-c(rep(1, 50), rep(2, 50))
> # Argument pch specifies the point types
> plot(x, y, col=color, pch=pnt)
```

There is one value in `color` and `pnt` vectors for each observation in the data. However, it is not obligatory to give the colors and point types this way. Vector giving the color and point type can be of lenght 1, i.e., contain only one value. Then all the points are colored and plotted with the same point type. These settings (`col` and `pch`) are actually passed on to the command `par` that makes the changes for this specific plot only. For example:

```
> plot(x, y, col=2, pch=3)
```

All possible point types are shown below.



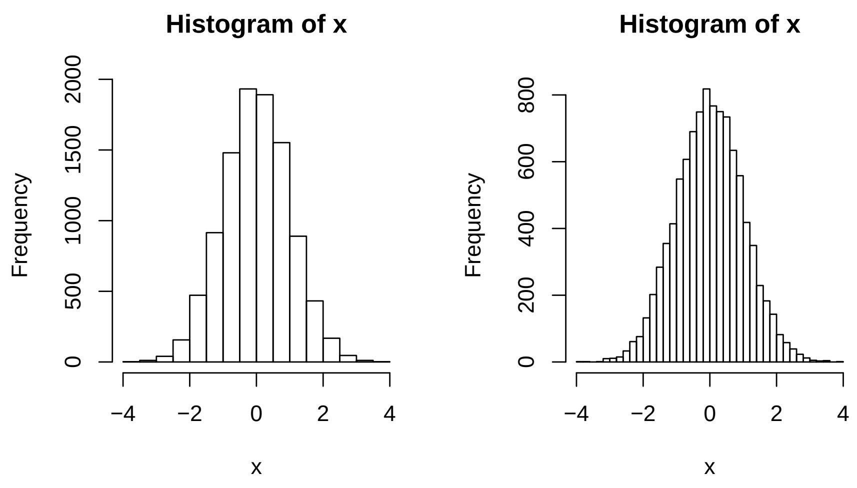
Colors can also called with their names instead of numbers. Valid color names are such as black, white, yellow, lightgreen. In addition to the listed point types, letters and other marks can be utilized. For example, specifying pch="H" would result into a plot where each observation is depicted with the letter H.

2.8.3 Histogram

Vectors can be visualized as histograms. The command for histogram is hist(). There are 10 bars in the histogram by default, but the number can be changed using the argument breaks. For example, the code:

```
> hist(x)
> hist(x, breaks=40)
```

produces the following two images:

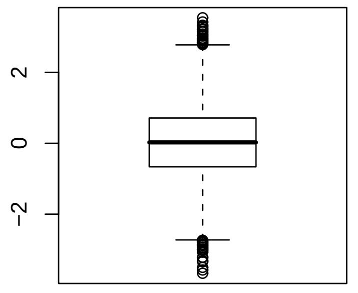


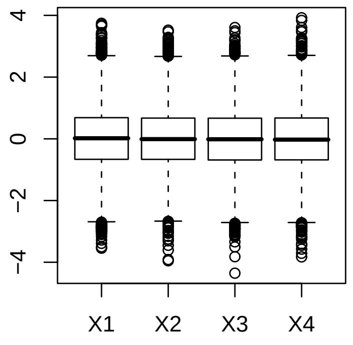
2.8.4 Boxplot

Boxplots can be produced using the command boxplot(). It can be used for plotting a single vector as a single boxplot, or a data frame as several boxplots in the same figure. If a matrix is plotted using boxplot(), it is treated as a vector, and only a single boxplot is produced. For example, the code:

```
> # Plotting a vector
> boxplot(x)
> # Plotting a matrix
> boxplot(dat)
```

creates the two images below:





There are several arguments available for boxplot, one of witch is `notch`. It highlights the median of the distribution using a notch. If the notches of two boxplots do not overlap it is strong evidence that the two medians differ.

2.8.5 Scatter plot

Scatter plot is produced by plotting two vectors using the command `plot()` as described above.

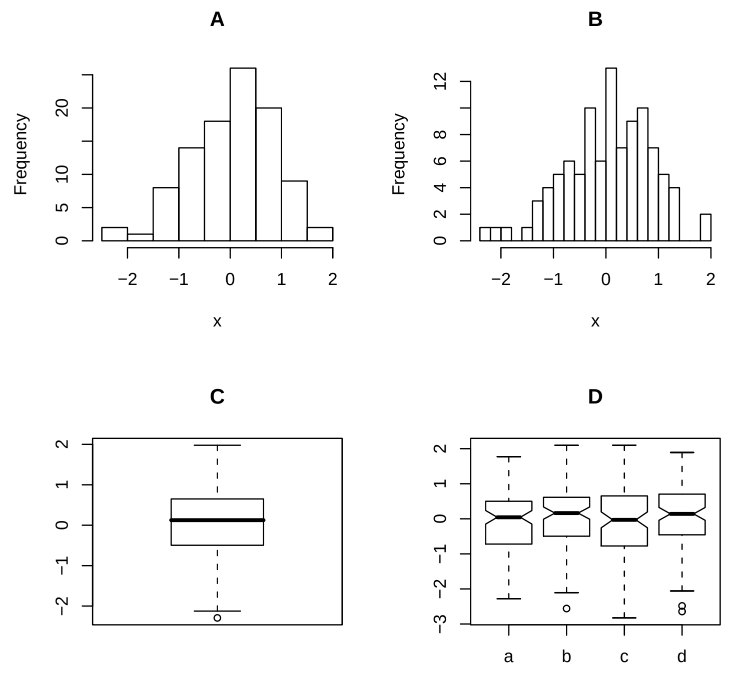
2.8.6 Panel plots

Panel plots are figures where several individual plots are visualized in the same figure. Typically all the plots are of the same type, but not necessarily. For example, the command `pairs()` generates a panel of scatter plots for a data frame.

There is a graphical parameter `mfrow` that can be used for setting the number of rows and columns in the panel. After setting the dimensions, an equal number of plotting commands are issued. These plots are plotted in the panel in the order from the top left to the bottom right corner of the panel.

For example, the figure below was produced using the following commands:

```
> # The dimentions of the panel:
> # 2 rows and 2 columns of plots
> par(mfrow=c(2,2))
> hist(x, main="A")
> hist(x, breaks=20, main="B")
> boxplot(x, main="C")
> boxplot(dat, notch=T, main="D")
```



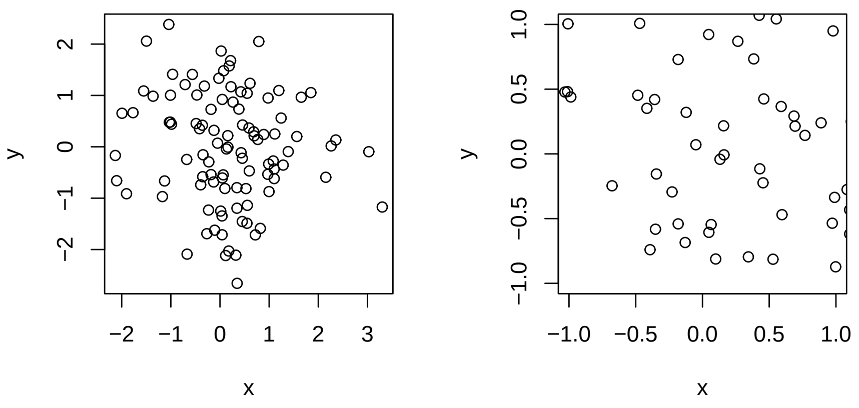
Note that the figure was set to be composed of four individuals plots (2 rows and 2 columns = 4 "cells"). Therefore, the par command dividing the plot region in four is followed by exactly four plotting command.

2.8.7 Other graphical settings

As already discussed, the graphical settings can be changed using the command `par()`. Next, a few of the most commonly used settings are introduced. If the settings are applied using `par()`, they apply for all of the plots produced in the same R session. If the options are applied plotwise, the settings apply only for the current plot.

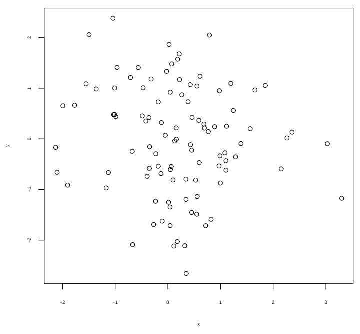
One of the most common tasks is changing the range of axes. In R this is done using the arguments `xlim` and `ylim`. Both arguments take a vector that holds two values, the lower end and the upper end of the axis range. For example, the two images in the figure below were produces as:

```
> # Default axis range
> plot(x, y)
> # User defined axis range
> plot(x, y, xlim=c(-1, 1), ylim=c(-1, 1))
```



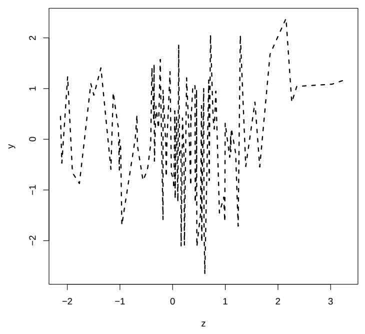
Another common task is changing the symbol and font size. This is accomplished with cex-arguments. Argument cex changes the plotting symbol size. Arguments cex.axis, cex.label and cex.main change the size of the axis annotation, x and y axis labels and plot main title. For example, this produces the image below:

```
> plot(x, y, cex=0.5, cex.axis=0.5, cex.lab=0.5)
```



Changing the plot type using command `plot()` was covered above. If the plot type is lines (`type="l"`), the type and width of the line can be changed using the arguments `lty` and `lwd`, respectively. For example:

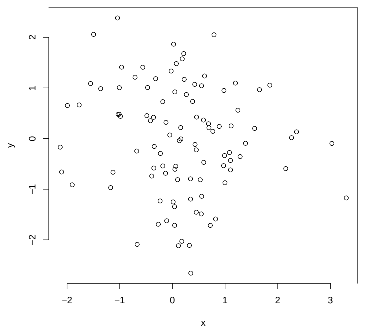
```
> # Plots dashed lines with width 2
> plot(z, x, type="l", lty=2, lwd=2)
```



Legitimate settings for the line type are numbers 0-6 ((0=blank, 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash). Line width can be any positive number, even a decimal number, although not all plot types support line widths less than 1 (the default).

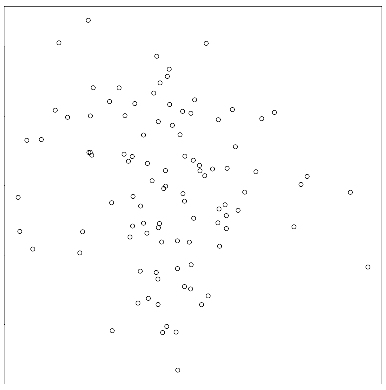
The argument `bty` specifies the type of the box drawn around the plot. Legitimate values are "o", "l", "7", "c", "u", and "]". The shape of the box resembles the capital letter of the argument value. For example:

```
> plot(x, y, bty="7")
```



Setting margins correctly is probably the most complicated issue of graphical settings. The argument `mar` specifies the number of margins on each side of the plot. The argument requires a vector of four values in the order bottom, left, top, right. The default is c(5, 4, 4, 2) + 0.1. To remove all the blank areas around the plot, the following settings can be used (bottom-left image):

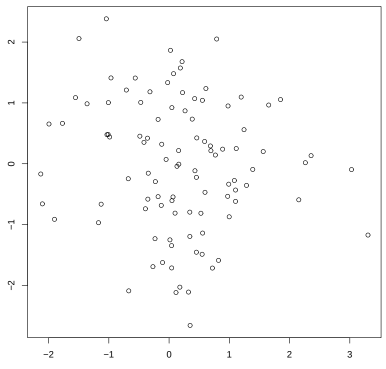
```
> par(mar=c(0,0,0,0))
> plot(x, y)
```



However, this effectively removes the areas reserved for titles, tick marks of axes, etc. A better result is often acquired by leaving some space for these around the plot. For example (bottom-right image):

```
> par(mar=c(2,2,2,0)+0.1)
> plot(x, y)
```

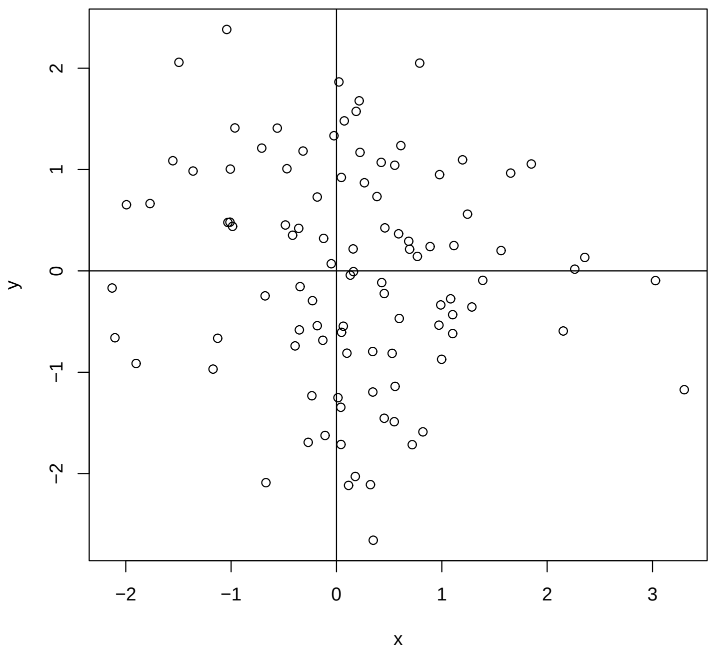
The small addition (+0.1) in the end of the margins settings leaves a small space between the plotted axes labels and titles and the outer limit of the plot area.



2.8.8 Adding new objects to the plots

New objects can be added to an already plotted image. Command `abline()` adds a horizontal or vertical line to a specified position in the existing plot:

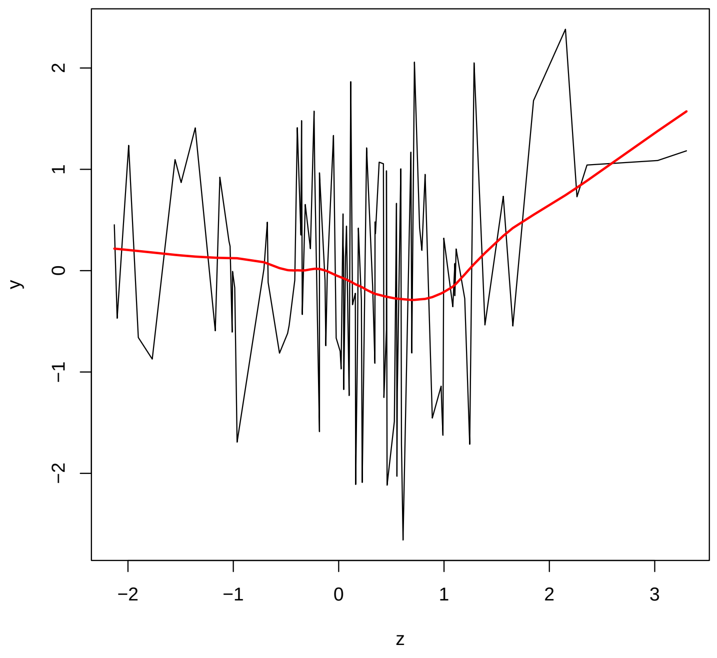
```
> plot(x, y)
> abline(h=0)
> abline(v=0)
```



Arguments `h` and `v` specify the horizontal or vertical position of the line. The position is given in the space of the plot. In the top-left image above, values of *x* and *y* range from -2 to +2, and the argument should be given values in this range. Otherwise the line is drawn but is does not appear in the plot, because it falls out the plotting area.

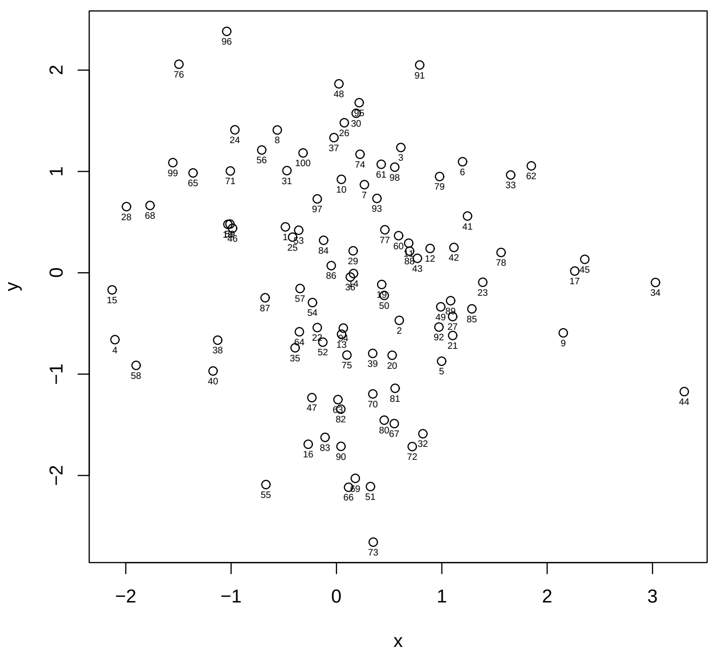
Lines can be added to an existing plot using the command `lines()`, also. It is a more general command than `abline()`, and even non-linear lines can be added to the plot. For example, a red lowess smoother line can be added:

```
> plot(z, y, type="l")
> # Calculates the lowess smoother
> l<-lowess(z, y)
> # Plot the lowess smother in the
> # existing line graph with red
> lines(l, col="red", lwd=2)
```



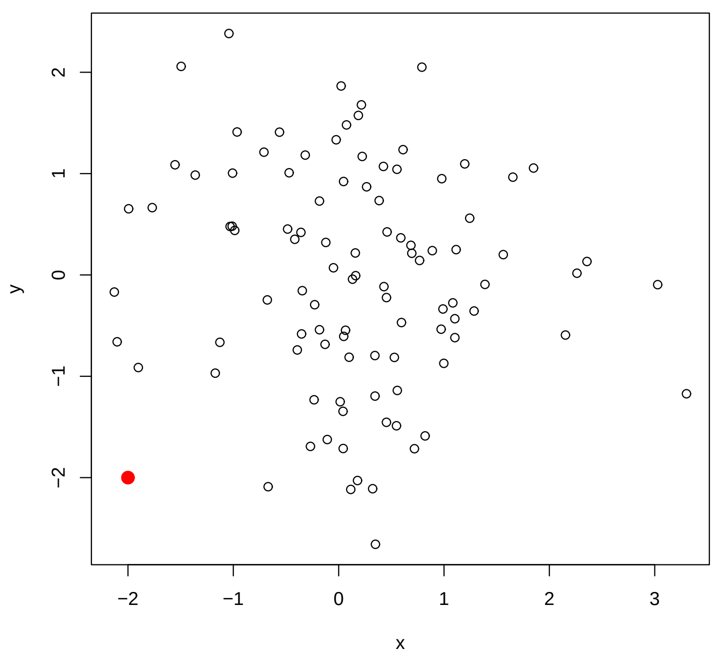
Sometimes it is of interest to know the labels of the observations plotted in the scatter plot. Labels can be added using the command `text()`. It requires three arguments, x coordinate of the text, y coordinate of the text and the text to plot. Additionally, it is often helpful to specify the size of the text to be plotted using graphical parameter `cex` as an argument:

```
> plot(x, y)
> # 0.1 is subtracted from the
> # y coordinate to make the
> # label not to overlap the
> # observations
> text(x, y-0.1, z, cex=0.5)
```



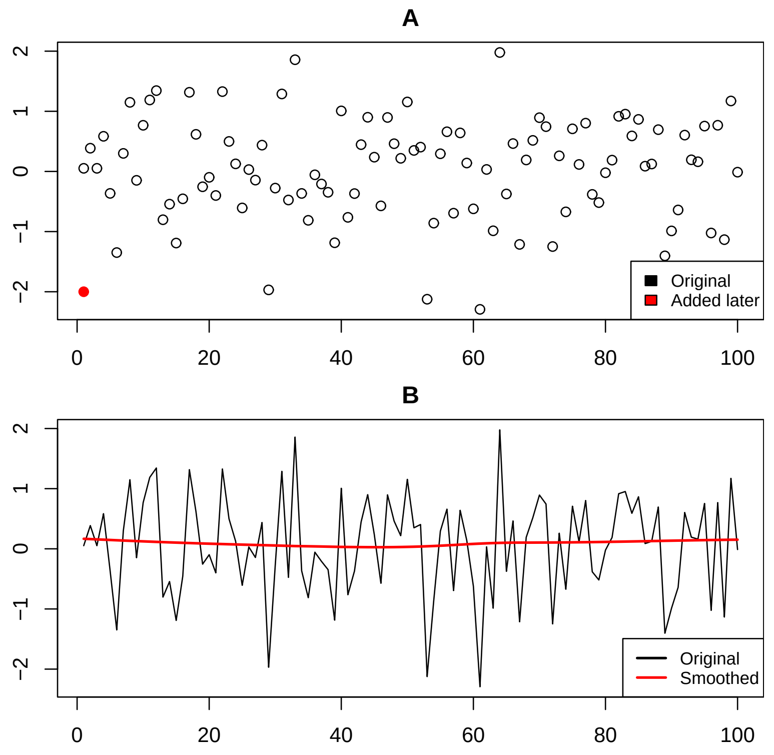
If there is a need to add new observations to the plot, it is possible to do this using the command `points()`. Points takes two arguments, x and y coordinates of the new observation. Color(s) of the new observation(s) can be specified using the graphical parameter `col`, and the size and type of the symbol can be changed simultaneously using arguments `cex` and `pch`:

```
> plot(x, y)
> points(-2, -2, col="red", cex=1.5, pch=19)
```



Legends are always added after the actual plotting of an image. Command `legend()` adds a legend. The arguments it takes vary with the plot type, but the two obligatory arguments are the position of the legend (`x`) and the text to be written in the legend (`legend`). For scatter plots, an extra argument specifying the colors of the filled boxes to appear next to the texts can be specified (`fill`). For line graphs, colors of the lines (`col`) and type (`lty`) and width (`lwd`) of the lines can be setup:

```
> # Scatter plot with legend
> plot(z, x)
> points(1, -2, col="red", pch=19)
> legend(x="bottomright", legend=c("Original", "Added later"),
+ fill=c("black", "red"))
> # Line graph with legend
> plot(z, x, type="l")
> lines(lowess(z, x), col="red", lwd=2)
> legend(x="bottomright", legend=c("Original", "Smoothed"),
+ col=c("black", "red"), lty=1, lwd=2)
```



2.8.9 Saving images

There are several ways to save images from R. In windows it is possible to activate the image windows (it is active just after plotting), and to save the image using the menu choise *File* -> *Save as*. It is possible to save the image as Windows metafile, postscript, PDF, PNG, bitmap or JPG. Windows metafile is not available in other systems. A general way to save the image to a file is to use the commands for directing the output for a graphical device.

The graphics device is first activated, the plot is generated and the graphical device is closed. The graphical devices are opened with one of the commands `postscript()`, `pdf()`, `png()`, `bmp()`, `jpeg()`. The plot is then generated as usual, and the devices is closed with the command `dev.off()`. All graphical device commands require slightly different options, so consulting the help page for the command is recommended. However, the obligatory argument for all the commands is the name of the file to plot to. For example:

```
jpeg("scatterplot.jpg")
plot(x, y)
dev.off()
```

The plot generated with the command `plot(x, y)` is now saved in the file scatterplot.jpg.

2.9 More information

R manuals can be accessed from web help pages (`help.start()`). An Introduction to R provides more details on the R language and an introduction to basic statistical and graphical commands. The R Language Definition gives extensive description of the language *per se*. For example, it lists all the possible operators.

The web page at `http://cran.r-project.org/other-docs.html` contains a list of documentation for the R language and environment written by people not directly associated with R development. A Finnish introduction to R has been written by Jari Oksanen from University of Oulu, and it can be accessed from `http://cc.oulu.fi/∼jarioksa/opetus/rekola/Rekola.pdf`.

A reference card for R commands is available from `http://cran.r-project.org/doc/contrib/Short-refcard.pdf`. It lists many R commands that can be used for data inport, manipulation, analysis and graphics.

Part II

Preprocessing

3 Importing DNA microarray data

This chapter describes how Affymetrix, Agilent, and Illumina data can be imported into R.

You need to have R and Bioconductor installed on your computer in order to be able to work through these instructions. See Chapter 1 for installation instructions if you don't have the tools available yet.

Before importing the data you should first change to the working directory where the data for analyses the resides. All data files are assumed to be located in the same folder. If you don't know how to change the working directory, please consult Chapter 2 for further instruction.

3.1 Affymetrix data

Traditional 3'-expression arrays are designed to include several probes per transcript. Affymetrix CEL-files contain slightly processed raw data of these probe intensities. The probe intensities are acquired from a scanned array image after background correction.

Data for other Affymetrix array formats, such as exon arrays or tiling arrays, can also be stored in CEL-files, but importing these data are not covered in this Chapter.

3.1.1 Reading CEL-files

Functions for reading Affymetrix data are available in the package affy. The function `ReadAffy()` reads in the raw data files, and stores the data as an AffyBatch object. By default, all CEL-files in the same directory are read. Alternatively, the files to be read can be specified using an additional argument.

In order to read the CEL-files you need to give the following commands:

```
> library(affy)
> dat<-ReadAffy()
```

The raw data is now stored in an object dat. You can check that everything was read correctly just by typing the name of the object to the prompt:

```
> dat

AffyBatch object
size of arrays=712x712 features (11 kb)
cdf=HG-U133A (22283 affyids)
number of samples=17
number of genes=22283
annotation=hgu133a
notes=
```

It seems that the data stored in dat is from human HG-U133A arrays, and those interrogate 22283 transcripts. There were 17 arrays (samples), so everything we needed was read in, and the array type was correctly recognized. As everything is in order, you can move on to Chapter 4, and normalize your data.

3.2 Agilent data

Agilent data files contain data from image analysis of scanned arrays. In addition to spot and background intensities they include several dozens of columns of quality control information.

3.2.1 Reading two-color data files

Functions for reading two-color data are in the package limma. Function read.maimages() takes care of the actual data import step. You need to specify the files you want to import by giving their names in the argument files to the function read.maimages(). If all the files in the same directory should be imported, a rapid way to generate a vector of the file names is to call the function dir(), which lists the files in the directory. In addition, you need to specify the datatype, which is given in the argument source. For example:

```
> library(limma)
> dat<-read.maimages(files=dir(), source="agilent")

Read GSM48254_wt.txt
Read GSM48255_mut.txt
Read GSM48501_mut.txt
Read GSM48503_mut.txt
Read GSM48507_wt.txt
Read GSM48510_wt.txt
Read GSM48539_mut.txt
Read GSM48543_mut.txt
Read GSM48546_wt.txt
Read GSM48584_wt.txt
Read GSM48585_wt.txt
Read GSM48586_wt.txt
Read GSM48587_mut.txt
Read GSM48588_mut.txt
```

Command `read.maimages()` reads all the files from GSM48254_wt.txt to GSM48488_mut.txt in an object called `dat`. This object is now of a type RGList. The type of an object can be checked using the function `class()`, e.g., `class(dat)`. You can check what it contains by typing its name to the prompt.

You can also quickly check whether all required fields were generated:

```
> names(dat)

[1] "G"       "Gb"      "R"       "Rb"      "targets" "genes"   "source"
```

3.2.2 Reading one-color data files

In addition to the traditional two-color format, Agilent data can also come in a one-color variaty. This data is generated by hybrizing just one sample to every array. Hence, the data files contain data for only one channel. Reading these data files is not as straight-forward as reading the two-color files, since function `read.maimages()` expects to get a file with two channels. It can be used for reading one-color data also, but for this to work, we need to trick the funtion to think we have two-color data (even if we actually don't).

First you need to find out whether the data is reported for the green channel or for the red channel. You can find this out just by opening the data file in a spreadsheet such as Excel. If it contains the column gMeanSignal, then the data is reported for the green channel. If the column is called rMeanSignal, then you know it's reported for the red channel. How the data is reported depends on how you did the scanning of the array.

Function `read.maimages()` can take an argument `columns` that we can use for specifying which columns in the data files we want to treat as data columns. Argument `columns` should be given a list of column names to be used for red (`R`) and green (`G`) spot intensities and red (`Rb`) and green (`Rg`) background intensities. As we only have one channel, we give exactly the same spot and background intensities for both red and green channels. For example:

```
> dat<-read.maimages(files=dir(), columns=list(
+ G = "gMeanSignal", Gb = "gBGMedianSignal",
+ R = "gMeanSignal", Rb = "gBGMedianSignal"))
```

Now the data is read into R so that both red and green channels contain data from the column gMeanSignal and both red and green channel backgrounds contain data from the column gBGMedianSignal.

In addition, we might read in the probe annotations from the data files. This is accomplished by giving a vector of column names that should be read as annotation during the data import. These column names are specified using the argument `annotation`. For example:

```
> dat<-read.maimages(files=dir(), columns=list(
+ G = "gMeanSignal", Gb = "gBGMedianSignal",
+ R = "gMeanSignal", Rb = "gBGMedianSignal"),
+ annotation=c("Row", "Col", "Start", "Sequence",
+ "SwissProt", "GenBank", "Primate", "GenPept",
+ "ProbeUID", "ControlType", "ProbeName", "GeneName",
+ "SystematicName", "Description"))
```

Again, the resulting object `dat` is of type RGList.

3.3 Illumina data

Illumina data can come in very many variaties, but here we discuss importing BeadSummaryData that is output from the image analysis of Illumina arrays. BeadSummaryData can come in several formats, and the format is customizable, so some of the columns might be missing from certain BeadSummary-Data files. We will cover two formats, one from the BeadStudio version 1 that uses TargetIDs as identifiers, and another one from BeadStudio version 3 that reports expression for every ProbeID. There can be several ProbeIDs per gene. TargetID is a summary of these ProbeIDs, and can be thought to represent a single gene.

Regardless of the BeadStudio version, each BeadSummaryData file might contain data for more than 1 array. The functions that are covered in the following sections read in the whole file, and it should not be modified before importing it.

3.3.1 Reading BeadStudio v1 data

The functions for reading Illumina data are in two different packages, beadarray and lumi. Here, we will use beadarray for reading in the data. Function `readBeadSummaryData()` reads the data in. Usually the data are stored in a tab-delimited file. To be able to read it in, a file containing the BeadSummaryData should be specified (here, Testi.txt). In addition, our BeadSummaryData file identifies the genes using TargetIDs, which we indicate using the argument `ProbeID`. The file is tab-delimited, and the argument `sep` is used for indicating this. Last, there is a header with seven lines that we don't want to read, so we skip those lines (argument `skip=7`).

```
> dat<-readBeadSummaryData("Testi.txt",
+ ProbeID="TargetID", sep="\t", skip=7)
```

Data are now stored in an object `dat` that is an ExpressionSetIllumina. You can check that everything went fine just by typing dat to the prompt:

```
> dat

ExpressionSetIllumina (storageMode: list)
assayData: 24350 features, 8 samples
  element names: exprs, se.exprs, NoBeads, Detection
phenoData
  rowNames: sample1, sample2, ..., sample8  (8 total)
  varLabels and varMetadata description:
    sampleID: NA
featureData
  featureNames: GI_10047089-S, ..., GI_9998947-A  (24350 total)
  fvarLabels and fvarMetadata description:
    ProbeID: NA
experimentData: use 'experimentData(object)'
Annotation:
QC Information
 Available Slots:  exprs se.exprs NoBeads controlType
  featureNames:
  sampleNames:
```

3.3.2 Reading BeadStudio v3 data

BeadStudio version 3 files can be imported using the same `readBeadSummaryData` -function as the older data files, but the function calls needs modifications. First, BeadStudio version 3 files don't include any header by default, so we don't need to skip any lines from the beginning of the file (`skip=0`). Second, the unique identifier is now in the column ProbeID that contains identifiers for all probes (not genes!) on the array:

```
> dat<-readBeadSummaryData("ProbeProfile.txt",
+ ProbeID="ProbeID", sep="\t", skip=0)
```

Again, data are now in an object `dat` that is an ExpressionSetIllumina. You can check that everything went fine just by typing `dat` to the prompt.

4 Normalizing DNA microarray data

Normalization is a broad term for methods that are used for removing systematic variation from DNA microarray data. In other words, normalization makes the measurements from different arrays inter-comparable. The methods are largely dissimilar for different DNA microarray technologies. For example, robust multiarray average (RMA) is a commonly used method for preprocessing and normalizing Affymetrix data, but it can't be applied to any other data types. However, one part of the RMA method is quantile normalization that is applicable to all data types.

Typically log2-transformed data is used for further analysis. Most of the normalization functions produce data in this format by default. If this is not the case, it is indicated below after the normalization. After normalization and possible log-transformation, the data is saved in a tabular format for further analysis.

Some of the most typical normalization methods are covered in this chapter. These methods apply to whole genome chips, where we can assume that most of the genes are not changing. Normalization is applied to the imported data that was stored in an object `dat` in the previous chapter.

4.1 Normalizing Affymetrix data

Normalization is just one part of Affymetrix data processing before estimates of gene expression are ready for further analyses. Typically preprocessing methods, such as RMA, consist of several steps: background correction, normalization of probes, and summarization where individual probes are combined into a probeset.

Functions for Affymetrix normalization are distributed over several packages. The MAS5 method developed by Affymetrix is available in the package affy, command `mas5()`. A newer method Plier, also developed by Affymetrix is available in package plier, command `plier()`. The RMA method is implemented in package affy (command `rma()`, but it's adaption for taking into account the differences in probe's GC% (GCRMA), is available in a separate package gcrma (command `gcrma()`.

Here, we will apply RMA preprocessing to the data. The reason why RMA was chosen is based on observations that it gives highly precise estimates of expression (which is desirable), although it might not give as accurate results as MAS5. In other words, RMA seems systematically to underestimate gene expression.

RMA preprocessing can be applied to the imported data, if the affy library has been loaded into memory. The preprocessed data will be stored in an object `dat2`:

```
dat2<-rma(dat)
```

Function `rma()` assumes that all raw data are first read into R. When the dataset is larger than about a dozen or few a dozen arrays, the data might not fit into the computer's memory. In such cases RMA preprocessing should be applied without reading the data into memory. Function `justRMA()` accomplishes this. It assumes that the working directory is pointing to the directory where the data resides:

```
dat2<-justRMA()
```

Note that the preprocessed Affymetrix data is now stored as an ExpressionSet:

```
> dat2

ExpressionSet (storageMode: lockedEnvironment)
assayData: 22283 features, 17 samples
  element names: exprs
phenoData
  sampleNames: GSM11805_normal.CEL, ..., GSM12444_normal.CEL  (17 total)
  varLabels and varMetadata description:
    sample: arbitrary numbering
featureData
  featureNames: 1007_s_at, ..., AFFX-TrpnX-M_at  (22283 total)
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
Annotation: hgu133a
```

4.2 Normalizing Agilent data

Agilent data normalization typically consists of two phases, background correction and normalization. Normalization is slightly different for one-color and two-color Agilent data so these are presented separately.

4.2.1 Two-color data

For background correction, several methods exist. The most simple is subtraction where the background intensities are simply subtracted from the spot (foreground) intensities. The downside of subtraction is that it typically generates plenty of negative estimates of expression. Therefore, other methods that guarantee that the estimates of expression are positive have been developed. According to a published comparison, the best of these alternative methods is normexp with an offset 50. Offset is the number added to the spot intensities to assist in the background correction step.

Dye-bias is a common phenomenan in any two-color platform. It is acaused by unequal labeling by the two dyes. Dye-bias can be somewhat corrected using a lowess (or loess) normalization, which fits a curve to the data, and uses this curve for normalizing the expression values. Therefore, lowess normalization is probably the most commonly used method for any two-color data, and it will be covered here.

Functions for normalizing two-color Agilent data are available in package limma. Function `normalizeWithinArrays()` does both background correction and loess normalization for arrays. It can be used in conjunction with normexp background correction with offset of 50:

```
> library(limma)
> dat2<-normalizeWithinArrays(dat, method="loess",
> "normexp", offset=50)
```

The resulting normalized data is stored in an object `dat2` that is of type MAList.

In addition to the array normalization, one can also want to normalize the genes. This is not necessary to make the arrays comparable, but is sometimes used. However, note that the gene-wise normalization might affect the results of filtering and statistical analyses. Most typically the gene-wise normalization is used with visualization methods, such as hierarchical clustering. Normalization of genes is carried out using the command `normalizeBetweenArrays()`. It can be applied to the array-normalized data the same way as it is applied to the one-color Agilent data (see the next section), with a possible exception that the argument `method=scale` is used instead of `method="quantile"`.

4.2.2 One-color data

One-color data is normalized in two steps. First the background correction is applied to the raw data (using normexp + offset 50 method), and then the corrected values are normalized. As the last step the data is log2-transformed. This requires running three separate functions. Here, quantile normalization is used:

```
dat2<-backgroundCorrect(dat, "normexp", offset=50)
dat2<-normalizeBetweenArrays(dat2$R, method="quantile")
dat2<-log(dat2)
```

4.3 Normalizing Illumina data

Illumina data can come either background corrected or not corrected. Illumina's BeadStudio software does the background correction, and we are not going to touch it here. As a consequence, normalization of Illumina data consists of just one step, normalization. Illumina suggests using rank invariant normalization that is based on genes that don't change their rank very much on different arrays. Another commonly used method is quantile normalization. The function `normaliseIllumina()`, available in package beadarray, is able to carry out both of these possibilites, but we will use quantile normalization here. It is the default option in the normalization function:

```
> dat2<-normaliseIllumina(dat)
```

Note that the data type of `dat2` is ExpressionSetIllumina.

4.4 Getting the raw data

The previous sections have explained how to get from the raw data to the normalized values. Sometimes it is useful to be able to get the raw data also. It is rather typical to compare the raw data to the normalized values in the quality control phase.

For Affymetrix data there is no such concept as raw data, if one wants to see the values already grouped into probesets. There are just different preprocessing methods.

For Agilent data, the raw data can be produced by using the normalization method none. In other words, instead of using the arguments `method="loess"` or `method="quantile"` in the calls to commands `normalizeithinArrays()` or `normalizeBetweenArrays()`, one can change the argument to `method="none"`. This produces an object that holds the un-normalized data. In addition, it is possible to drop the background correction phase to get to the really untransformed data.

For Illumina data, an argument `method="none"` can be provided in the call to the command `normaliseIllumina()`. Similarly to Agilent data, it produces an object that holds the untransformed raw data read from the arrays, and put into a suitably formatted R object.

4.5 Saving the expression values

After normalization the data is stored in a format that is internal to certain Bioconductor packages, such as affy or beadarray. Most of the functions in R don't know how to handle these types, so it might be worth while to write the expression values to disk in a tab-delimited text file.

For Affymetrix and Illumina data the expression values (already log2-transformed) can be extracted using the command `exprs()`:

```
> dat.m<-exprs(dat2)
```

In addition, Illumina data should be log2-transformed

```
> dat.m<-log2(dat.m)
```

For Agilent data the procedure is different depending on the data type. For two-color data, we should first extract the expression values (the M-values), and then assigning the genes their correct names:

```
> dat.m<-dat2$M
> rownames(dat.m)<-dat2$genes$ProbeName
```

In case you're wondering where one can find out what values to save, first check the help file for the command `normalizeBetweenArrays()`. It lists all the values it produces in the Value-field. Second, you can check what fields there are in the normalized data object `dat2` by typing the command `str(dat2)`. Command `str()` is a shorthand for "structure", and it prints on the screen the structure of the data object.

For one-color Agilent data no such tricks are required, since the normalized data is already in a tabular format. However, we can save it using the same name (`dat.m`) as for other data type by:

```
> dat.m<-dat2
```

Object dat.m containing the expression values in a tabular format can be written to disk using the command `write.table()`. It takes several argument. In the order of appearance these are the name of object to be written, name of the file the data should be written to, separator (here tabulator), whether to write row names, whether to write column names, and whether to quote the character values, such as probe IDs. So the following command writes row and column names and data to a file called affymetrix.txt, and nothing gets quoted.

```
> write.table(dat.m, "affymetrix.txt", sep="\t",
+ row.names=T, col.names=T, quote=F)
```

5 Quality control

This chapter introduces some simple graphical exploration methods for checking the quality of the data both before and after normalization.

Arrays are often thrown out just by looking at the quality control information. It might be better to base the decision on several sources of data. If several quality control plots suggest that there is something terribly wrong, then it might make sense to exclude that particular sample from further analysis. Or, if there is strong evidence from the lab that the sample was mishandled, then it might be better it remove it, also.

Rather often ordination methods, such as principal component analysis or non-metric multidimensional scaling are used for checking whether the biological replicates go together. It is rather typical that there is some mixing of biological groups in these images, so excluding samples from the analysis on the basis of ordination plots might not always make sense.

5.1 Checking Affymetrix data

Quality control of Affymetrix arrays is performed for raw data, i.e., imported CEL files. In Chapter 3 Affymetrix CEL files were read in an object `dat`. Quality checks can be performed using that raw data.

Basic quality control for Affymetrix consists of checking for RNA degradation and examining the expression for control genes, scaling factors, percentage of present genes and the average background. Functions for performing these analyses are devided between two packages, affy and simpleaffy. In addition, boxplot, hierarchical clustering and non-metric multidimensional scaling can be used to complement these basic tools. Thse work with the normalized data matrix (`dat.m`) as specified for the Agilent one-color and Illumina data in their respective sections.

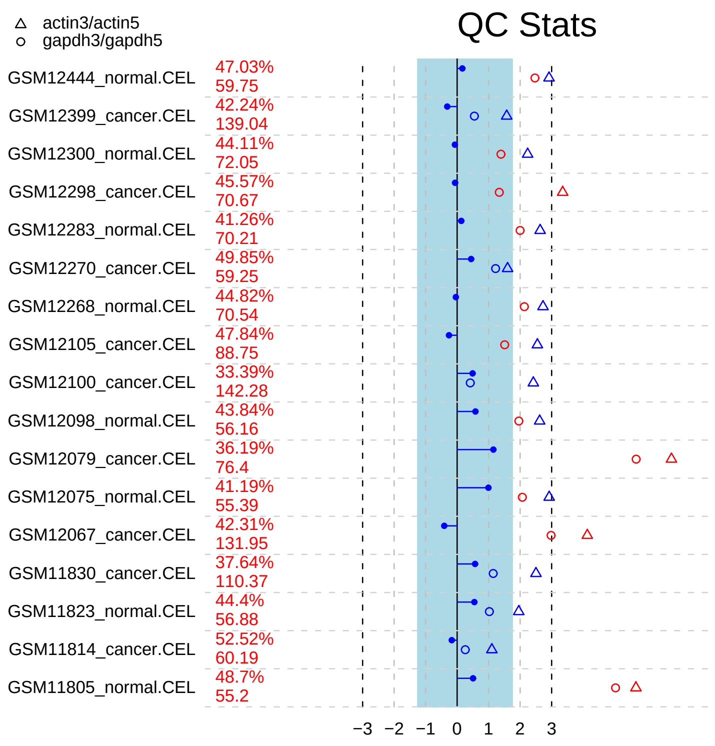
First we need to calculate the quality control information. RNA degradation is assessed using the function `AffyRNAdeg()` from the affy package, and the other descriptives can be calculated using the function `qc()` from the package simpleaffy. Note that the input for both of these quality control measures is the AffyBatch object holding the raw data:

```
> library(affy)
> library(simpleaffy)
> aqc<-qc(dat)
> deg<-AffyRNAdeg(dat)
```

It might be interesting to see what kind of information objects `deg` and `aqc` contain, but typically we want to examine a visualization of the result. Plotting the general quality control statistics is simple. Command:

```
> plot(agc)
```

will produce the following image:

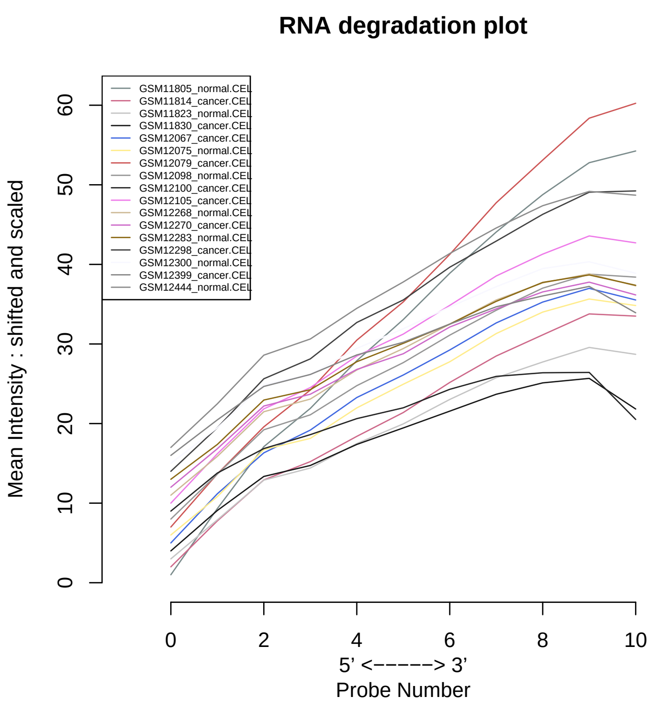


QC stats plot reports quality control parameters for the chips. Different chips are separated by vertical grey lines in the plot. The red numbers on the left report the number of probesets with present flag, and the average background on the chip. The blue region in the middle denotes the area where scaling factors are less than 3-fold of the mean scale factors of all chips. Bars that end with a point denote scaling factors for the chips. The triangles denote beta-actin 3':5' ratio, and open circles are GADPH 3':5' ratios. If the scaling factors or ratios fall within the 3-fold region (1.25-fold for GADPH), they are colored blue, otherwise red. The deviant chips are therefore easy to pick of by their red coloring.

Here, all scaling factors are within the acceptable range, but some of the housekeeping genes fall outside the acceptable range. However, most of the chips having very deviant control gene expression are samples from cancerous tissue, and the quality control genes might represent the large gene expression changes typically displayed by the cancer tissue. So, there is nothing too worrying in the quality control images.

RNA degradation plot is slightly more demanding to plot, since getting a good image requires tinkering with some graphical parameters. First we sample a number of colors from a set of all available colors, and save these to an object `cols`. This object then contains as many distinct colors as there are samples in the dataset (number of rows in the experimental description of the AffyBatch object). Then we plot the image (command `plotAffyRNAdeg()`) using the colors so that every line in the image is colored individually. Adding the argument `col=cols` in the plotting command accomplishes that. Last we add a title (command `legend()`) that labels the arrays in the plot using the names of the original datafiles (again read from the experimental description part of the affyBatch object). The legend goes to the top left corner of the image (argument `x="topleft"`), and for every array it holds one thin line (argument `lty=1`) that is colored according to the color we earlier generated for the arrays (argument `col=cols`) and labeled with small text (argument `cex`).

```
> cols<-sample(colors(), nrow(pData(dat)))
> plotAffyRNAdeg(deg, col=cols)
> legend(legend=sampleNames(dat), x="topleft",
+ lty=1, cex=0.5, col=cols)
```



In the resulting image every single array is represented by one line. The idea is to check whether the slopes and profiles of the lines are similar for all the arrays. It is easy to spot the two lines that deviate from the others by having a steeper slope. Even if they seem to be dissimilar to others, it would probably not be too worrisome a phenomenan, and it is acceptable to retain them in the analysis.

5.2 Checking Agilent data

For Agilent two-color arrays there are two very commonly used quality control tools, namely MA plot, and density plot. MA plot visualizes the modified red and green intensities against each other. Density plot creates a smoothed histogram (or more correctly kernel density estimate) of expression values. After normalization MA plots should not contain any visible non-linearities and all the chips should display about the same smoothed histogram in the density plot. In addition, boxplot, hierarchical clustering and non-metric multidimensional scaling can be used to complement these basic tools. These work with the normalized data matrix (`dat.m`) as specified for the Agilent one-color and Illumina data in their respective sections.

It might be a good idea to produce the quality control plots for both normalized and un-normalized data to check how the data has changed. See the chapter Normalizing DNA microarray data for more details on how to produce both normalized and un-normalized values for Agilent data.

5.2.1 Two-color data

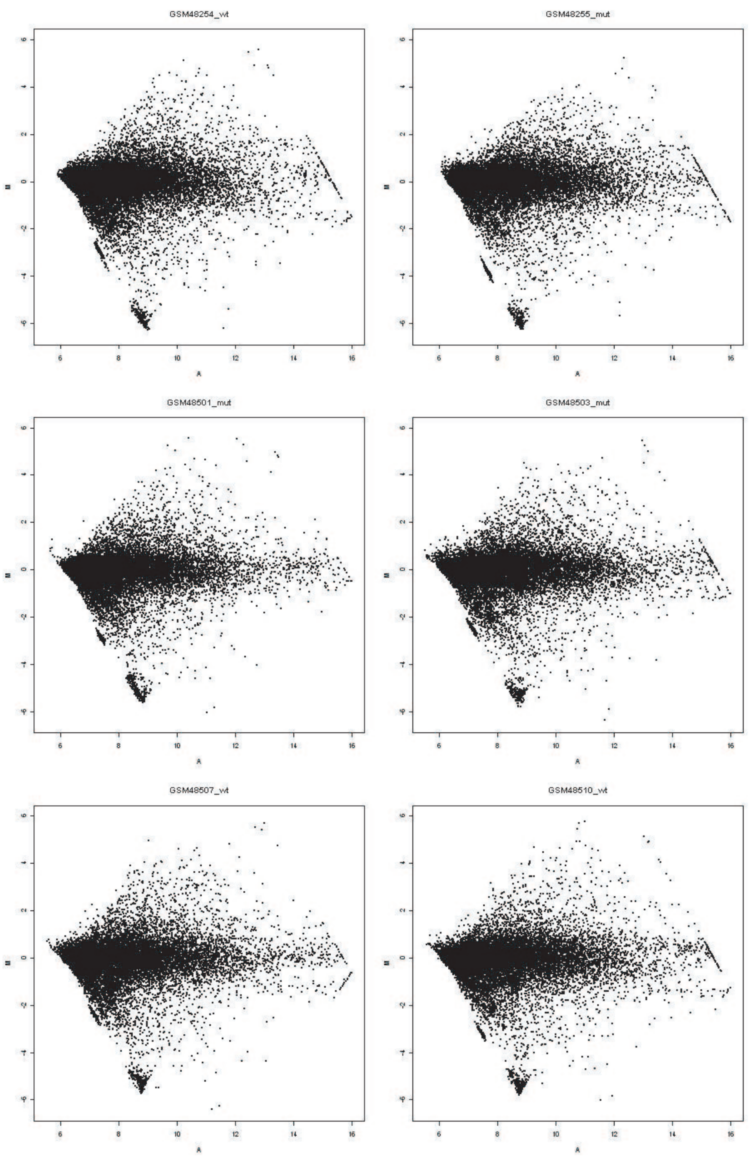
Quality control for Agilent data is typically carried out on the normalized data. In Chapter 4 we normalized the raw data, and saved the normalized values in an object `dat2`. We will use this object for running the quality control analyses.

MA plots can be generated one array at a time, but it is easier to plot six arrays at a time. There is function `plotMA3by2()` in limma package that does exactly that. This function actually automatically creates image files, each of which contains MA plots for six arrays. To generate the plot, you can use the command:

```
plotMA3by2(dat2, device="pdf")
```

The command saves the plots in PDF files (argument `device="pdf"`)that can be opened (outside R) with some PDF reader, such as Adobe Reader. Here we had 14 arrays, so three separate files were created.

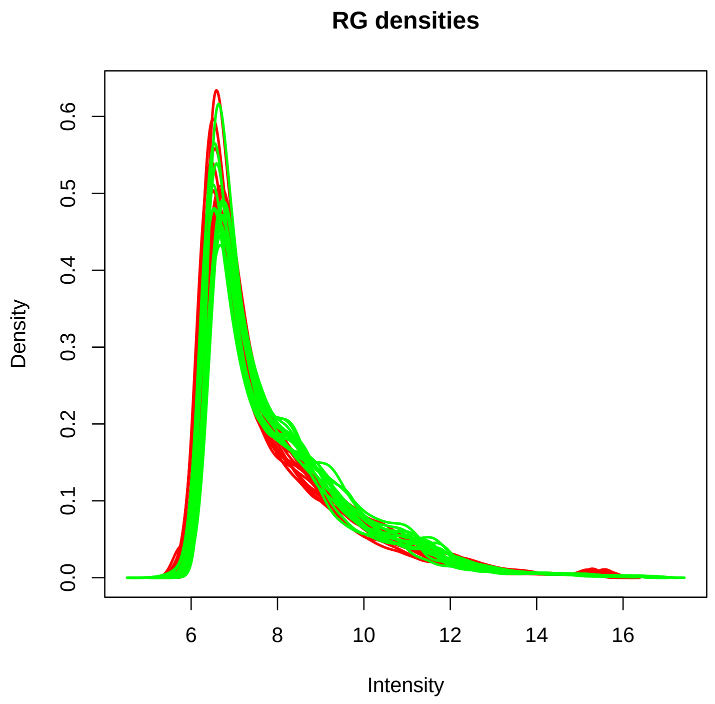
The plots in the first file (see below) show show some saturation in the higher end of the scale. Saturation can be spotted by looking for lines in the lower of higher end of the expression value scale. For example, in the first plot on top-left, saturated spots form a very distinct line extending from top-left to lower-right in the plot. There are often some saturated spots, but their frequency should not be too high. If there are very many saturated spots, then the scanner settings used while scanning the slides could have been wrong.



The other quality control image containing the smoothed histograms for both channels and all arrays of the normalized data, can be produced using the command:

```
> plotDensities(dat2)
```

The resulting image looks something like:



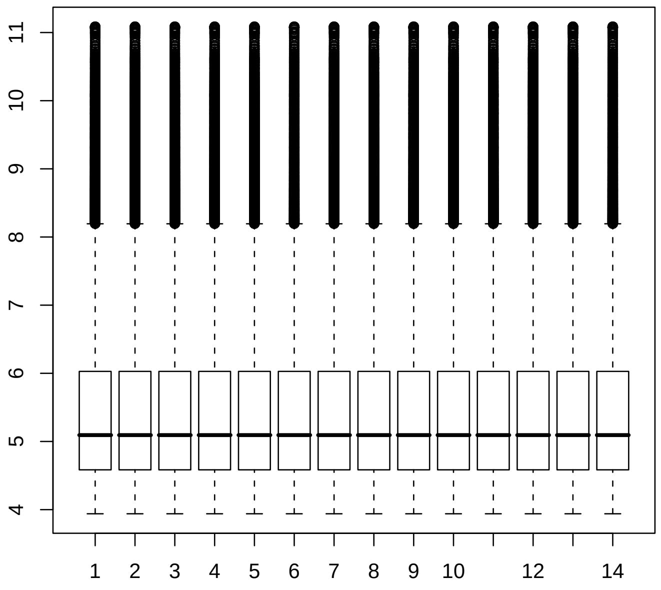
Since the smoothed histograms for all arrays and both channels look very similar, there are no systematic biases in the data anymore, and normalization has done a good job in removing them (if some was present in the beginning).

5.2.2 One-color data

Agilent one-color data can be checked using similar tools to Agilent two-color data. One-color data can not be automatically visualized using the function `plotDensities()`, since it doesn't work with one-color data. The same information can be derived from a boxplot. Normalized data is stored as a matrix in an object `dat.m`. Before generating the boxplot, object needs to be converted into a data frame:

```
> boxplot(data.frame(dat.m))
```

The resulting image should look something like the one below. This is a standard boxplot where every single array is represented by one box. One should check that the median are all on the same level. Medians are marked with the horizontal bars inside the boxes. In this case all arrays seem to have exactly the same distribution of expression values, and this is a result of applying quantile normalization that specifically makes the distribution exactly the same. Thus, for quantile normalized data the boxplot is not a very informative quantile control tool.



In addition, hierarchical clustering and non-metric multidimensional scaling can be used to complement these basic tools. These work with the normalized data matrix (`dat.m`) as specified for the Illumina data in the section below.

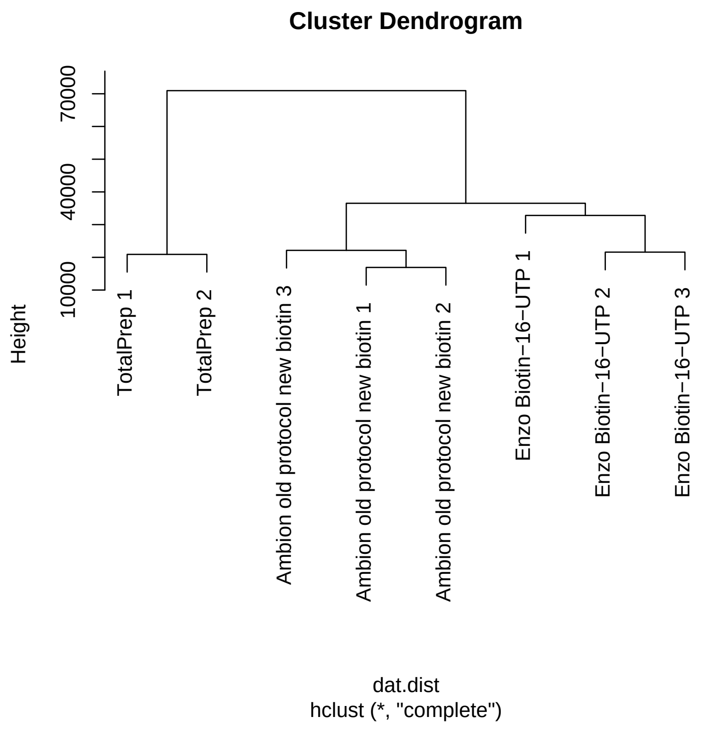
5.3 Checking Illumina data

The same methods that were used for the Agilent data are in principle applicable to Illumina data. One caveat is that quantile normalization is typically used for Illumina data, and that renders boxplots, and smoothed histograms practically useless. But there are other means available for checking, e.g., replication. The two widely used methods for checking the quality of replication is to produce a dendrogram and see if the samples from the same group are clustered together, and to produce an ordination plot using non-metrix multidimensional scaling (NMDS).

Let's start with hierarchical clustering in order to produce a dendrogram. First we need to calculate all pairwise distances between the samples using the command `dist()`. Note that the expression matrix where columns represent samples need to be transposed so that samples become rows before calculating distance. This is important, since the `dist()`-function calculates distances between rows, and calculating distances between tens of thousands of rows would take a long time. Transposition is taken care of by the command `t()`. Then these distances between samples are turned into a dendrogram using the unweighted pair group method with arithmetic mean (UPGMA or average linkage) method. UPGMA is the default tree construction method in the command `hclust()`. Last the tree can be visualized using a call to the command `plot()`.

```
> dat.dist<-dist(t(dat.m))
> plot(hclust(dat.dist))
```

The resulting plot looks something like the one below. Now, all the samples group very nicely according to their biological grouping, which indicates that the data is at least in that sense clean and ready for further analyses.

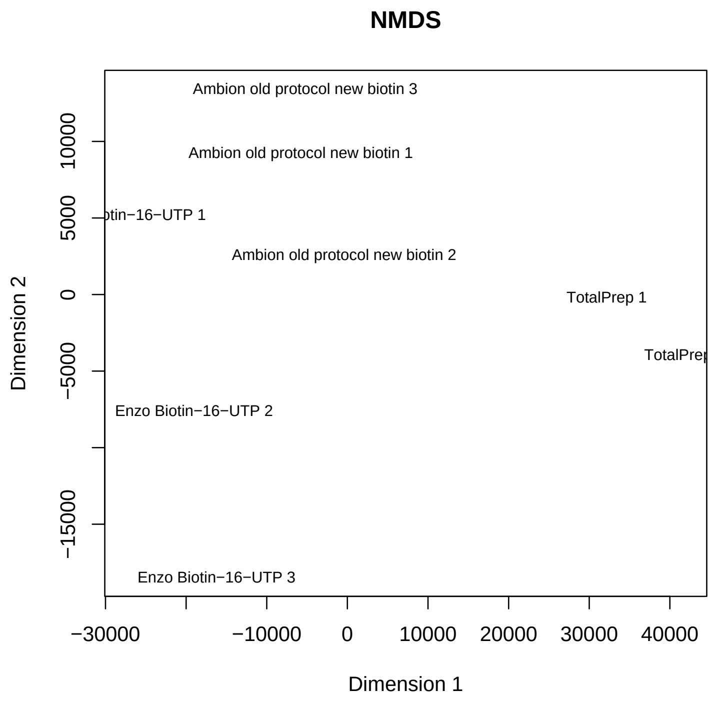


The same distances can be used for producing an ordination plot. The function for calculating an NMDS solution is in the package MASS. The NMDS can be produced by the command `isoMDS()` and the resulting ordination is then plotted using the two first axes of the NMDS solution.

In the command `plot()` we plot the first axis (`mds$points[,1]`) of the NMDS on the x-axis (it's mentioned first in the plotting command), and the second axis (`mds$points[,2]`) to the y-axis. We label the plot as NMDS (argument `main="NMDS"`), and the axes as Dimension 1 (argument `xlab="Dimension 1"`) and Dimension two (argument `ylab="Dimension 2"`), but we do not mark the samples in the plot with any symbols (argument `type="n"`). After the plot has been created, we add the sample names as labels in the plot using the command `text()`.

```
> library(MASS)
> mds<-isoMDS(dat.dist)
> plot(mds$points[,1], mds$points[,2], main="NMDS",
+ xlab="Dimension 1", ylab="Dimension 2",
+ type="n")
> text(mds$points[,1], mds$points[,2],
+ rownames(mds$points)), cex=0.75)
```

The resulting plot resembles the one below.



The samples that belong to the same group can be easily distinguished from the other groups using these two axes, so the NMDS plot supports our view that the quality of the replicates is good.

6 Filtering and differential expression

6.1 Why filtering?

Filtering is often used to describe both unspecific filtering, a topic covered in this chpater, and specific filtering, a topic covered in the next chapter. Unspecific filtering refers to methods for excluding a certain part of the data without any knowledge of the grouping of the samples. Specific filtering is used is situations when the filtering is affected by the known grouping of the samples. For example, in a case-control study genes that are expressed on a very low level across all samples might be removed in an unspecific filtering process. Genes could also be removed from the data using some statistical test or some other method that requires group knowledge. These latter approaches are specific filtering methods.

Unspecific filtering is typically used for excluding any uninteresting genes from the dataset. Genes that are not changing at all during the experiment or are expressed on a very low level so that their measurements are unreliable, are usually excluded from further analyses. There is an on-going discussion whether this is actually a good or bad habit, since filtering often somewhat alters the results of the subsequent statistical tests (making the adjusted p-values lower than they would be without filtering first). If the filtering in truly unspecific, then no bias has been introduced to the statistical testing, and it's results should be valid. If in doubt whether to filter or not, one can alway first run a statistical test, and after that use unspecific filtering.

This chapter introduces two unspecific filtering methods, filtering by standard deviation and filtering by expression. These are just two examples of a wide range of possible filters, but they have been selected due to their popularity in the published papers of the field.

6.2 Filtering tools

After normalization we saved the normalized expression values in a matrix called `dat.m`. This object is used as input for the filtering tools introduced next.

6.2.1 Standard deviation filter

Library genefilter contains ready-made functions for filtering. For the standard deviation filter, we first calculate a standard deviation for every single gene. Function `rowSds()` does the calculation fast. After calculating these row-wise statistics, they are used for excluding the genes; we only retain those rows (genes) of the expression matrix that have a standard deviation of at least 2. The following code does the calculations:

```
> library(genefilter)
> rsd<-rowSds(dat.m)
```

Now that we have the standard deviation saved in a vector `rsd`, we can use it for filtering the matrix. If you don't recall how matrices (or data frames) were subsetted using subscripts, please see the chapter Introduction to R. The following code does the actual filtering. The filtered dataset is saved in a new matrix called `dat.f`.

```
> i<-rsd>=2
> dat.f<-dat.m[i,]
```

On the first line of the code above a vector of logical (TRUE or FALSE) values is created. If the value in rsd is larger than or equal to 2, then the logical value returned is TRUE, otherwise FALSE gets returned. On the second line of the code this vector is used for filtering the data matrix. If the value in the vector `i` is TRUE, the corresponding row from the data matrix is selected and saved in the new object `dat.f`.

6.2.2 Expression filter

When filtering by expression, it is not a sensible assumption that all arrays would behave similarly. On some arrays the gene might be expressed, but for some reason on some other arrays, it does not seem to be expressed at all or is expressed at a very low level. Therefore, the filter needs to take into account this possible discrepancy. This is implemented by letting the gene pass the filter (and to be included in the dataset), if the gene is expressed at the set level in at least some proportion of the samples. This kind of filters can be easily created using the functions `kOverA()` and `pOverA()`. The former function uses the absolute number of samples during filtering, whereas the latter function uses the proportion. We will use the proportion method here.

Filtering proceeds in several steps. First, a filtering function is created using the function `pOverA()`. Then this function is applied to all rows of the matrix using an accessory function `genefilter()`. Here we assume that we want to find 2-fold over-expression (`A=1`), and that the gene has to be over-expressed in at least 50% (`p=0.5`) of the arrays. The result is a vector of TRUE and FALSE values indicating whether the gene passes the filter or not. This vector is then used for subtracting the passed genes from the whole dataset using subscripts. The complete filter is as follows:

```
> ff<-pOverA(A=1, p=0.5)
> i<-genefilter(dat.m, ff)
> dat.fo<-dat.m[i,]
```

To be precise, we now have a set of over-expressed genes. If we also want ot get the under-expressed genes, we can invert the matrix, i.e, make under-expressed values over-expressed and vice versa. The procedure is exactly the same as the one outlined above, but we add a minus sign in front of the name of the matrix. This inverts it's values.

```
> ff<-pOverA(A=1, p=0.5)
> i<-genefilter(-dat.m, ff)
> dat.fu<-dat.m[i,]
```

We can combine these two matrices into one matrix, if we want to retain both under- and over-expressed genes in the same dataset. This can be accomplished using the funtion `cbind()` that combines two matrices row-wise, i.e., it adds the rows in the second matrix after the rows in the first matrix. This is explained in the chapter Introduction to R in more details. The procedure is as follows:

```
dat.f<-rbind(dat.fo, dat.fu)
```

6.3 Filtering after statistical testing

The tools presented above can be applied either before or after normalization. Here, the tools have been used for the normalized data, but you wish to apply them for the data that contains only the statistically significant genes, change the name of the object from `dat.m` to `dat.s`. If you have followed through the analysis in the statistical testing chapter, `dat.s` should contain the data for the differentially expressed genes only.

Part III

Analysis

7 Statistical analyses

Statistical analysis of DNA microarray experiments is still under heavy development. There are no concensus, no strict guidelines or real rules of thumb when to apply some tests and when never to apply certain other tests. One of the widely used tools for the statistical analysis is limma, which implements linear models. One of the assumptions of the limma's method is that the data is normally distributed (otherwise the significance tests give wrong results), but the real world data is not always normally distributed. From a typical Affymetrix experiment, maybe only about 20% of the expression values are normally distributed (inferred from several chips, of course). Other are non-normally distributed, and one should probably use non-parametric methods for the analysis. However, usually the same method is used for all genes, and the results are therefore only approximate. One can probably rank the genes according to the p-values, but assuming that the p-values are un-biased in the traditional statistical sense is an illusion.

Here, we will present the statistical analysis using limma package from the Bioconductor project.

7.1 Key concepts

Linear models are very versatile tools that can be used for analyzing even very complicated experimental setups. However, in order to be able to use tools in limma package, one must be able to build a model matrix that describes the experiment. Fortunately, the limma manual presents examples of many different experimental setups, both for one- and two-color data. We will introduce the concepts below using an example from a comparsion of control and treatment groups, but we do not cover any of the more complicated designs.

7.1.1 Model matrix for a two-group comparison

R comes with a command `model.matrix()` that makes building the model matrix a bit easier. Let's take a concrete example where we want to compare control samples and treatment samples. The experiment has been conducted using Affymetrix chips, with an equal number of samples in both groups (3 in each). The data has been normalized using RMA as oulined in the chapter Normalizing DNA microarray data. In the resulting matrix, the first three columns are control samples, and the last three columns are the treatment samples. Now, we want to code this information in R somehow. The easiest way is to a create a vector that contains one entry for every column in the normalized data matrix. We can code the samples with "C"and "T", so that the control samples are coded as "C" and treatment samples as "T". The R code that does that is as follows:

```
> groups<-c("C", "C", "C", "T", "T", "T")
```

Now the vector groups holds the information about the groups. Three columns coded as "C" and three columns coded as "T".

In order to be able to analyze these data, next we need to generate the model matrix. The vector groups helps as here a bit. Now, it is extremely important to understand that the same data can be analyzed in several ways, all of which give exactly the same answer. We will present here only one of these, in order to avoid introducing too much confusion.

If you've ever read any statistics, you might recall that in linear regression there usually is a constant term that tells us where the regression line crosses the Y-axis (if the expression is on the Y-axis, and the groups-variable on the X-axis). You might also recall that linear regression gives an estimate for the effect of every variable put into the model. Now we only have one variable we want to use (groups), and the only decision we need to make is whether to put an intercept in the model or not. Usually we want to keep the intercept in the model, since we do not want to assume that gene expression would be zero when the group is control.

To create such a model matrix, one needs to give the following R commands. First we convert the vector groups into a factor groups using the command `as.factor()`. This is not essential here, but if you used numerical codes for the groups in the first place, and especially if you have several groups to compare, the vector needs first to be converted into a factor in order to build the correct model matrix (for details, see the next section). The factor is then subsequently used to create the model matrix using the command `model.matrix()`

```
> groups<-as.factor(groups)
> design<-model.matrix(~groups)
> design
```

```
  (Intercept) groups2
1           1       0
2           1       0
3           1       0
4           1       1
5           1       1
6           1       1
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$groups
[1] "contr.treatment"
```

Object `design` now harbors the model matrix. There is one row in the model matrix for every sample in the dataset. You can see from the names of the `design` that there is an intercept, which is just a column filled with 1s, and another column groups2, which specifies that we want to compare treatment (coded with 1s in the model matrix column groups2) with the control group (coded with 0s in the model matrix column groups2). This is a general feature of the model matrix. The baseline group is coded with zeros, and the group that is compared to it is coded with ones.

What you need to keep in mind is that there is practically always one column filled with ones (the intercept) and one or more columns with, usually, zeros and ones.

7.1.2 Model matrix for a three-group comparison

In the previous section we covered the basics of the model matrix for a two-group case. The same principle is very easily expanded to a three-groups case. Let's assume that we have one control group and two treatment groups. Let's further assume that every group consists of two samples. This principle actually applies also when a short time series is analyzed. Different time points can be treated as seperate groups.

To create a model matrix, we first need to code the samples of the data matrix as we did in the two case:

```
> groups<-c("C", "C", "T1", "T1", "T2", "T2")
> groups<-as.factor(groups)
> design<-model.matrix(~groups)
> design
```

```
  (Intercept) groupsT1 groupsT2
1           1        0        0
2           1        0        0
3           1        1        0
4           1        1        0
5           1        0        1
6           1        0        1
attr(,"assign")
[1] 0 1 1
attr(,"contrasts")
attr(,"contrasts")$groups
[1] "contr.treatment
```

Now there are two columns in addition to the intercept in the model matrix. The first column compares the treatment 1 to the control and the second compared treatment 2 to the control.

It is worthwhile to name the groups wisely, since construction of the model matrix is then easier. By default R sorts the names of the groups alphabetically when the command `as.factor()` is called. In the example, the control group (C) is alphabetically the first, treatment 1 (T1) is the next, and the treatment 2 (T2) is the last. So, by naming the groups conveniently we can construct a model matrix with very little hassle.

7.2 Analysis using a linear model

Once the design matrix is ready we can move on to the actual analysis. The method in limma is called empirical Bayes, since it uses a method where certain parameter are inferred from the data (hence, empirical), and Bayes is term used to describe certain approaches in statistics. Empirical Bayes is a better analysis method than, say, traditional t-test for DNA microarray data, since it gives us more precise estimates of the statistical significance of the genes.

Empirical bayes analysis is simple in practise. The analysis is carried out by using the command `lmFit()` followed by `eBayes()`. The `lmFit()` wants to get the data matrix, and a design matrix. The analysis can be carried out using the filtered data or with the original un-filtered data as long as the data is in a matrix format.

For the unfiltered data:

```
> fit<-lmFit(dat.m, design)
> fit<-eBayes(fit)
```

For filtered data (only the name of the data matrix has been changed):

```
> fit<-lmFit(dat.f, design)
> fit<-eBayes(fit)
```

The object `fit` contains the results of the analysis. Results can be extracted using the command `toptable()`. By default it gives output for the first column of the design matrix, in other words, the intercept. However, this coefficient is seldom of interest, but the coefficient to report can be changed in the function call using the argument `coef`. If the experiment was a comparison between two groups, the following command would extract the genes that are statistically significantly differentially expressed between the control and treatment groups:

```
> toptable(fit, coef=2)
```

```
       logFC         t      P.Value adj.P.Val           B
1437  -5.080 -27.43783 3.371796e-05 0.4762233  0.13339585
527   -2.165 -20.40133 9.424038e-05 0.4762233  0.01676968
4134   3.695  17.77480 1.518048e-04 0.4762233 -0.06198460
9455  -1.860 -17.56082 1.582920e-04 0.4762233 -0.06982439
8879  -1.870 -16.15531 2.111136e-04 0.4762233 -0.12829384
4527  -2.945 -15.79118 2.283778e-04 0.4762233 -0.14570050
21232 -4.385 -15.20258 2.603142e-04 0.4762233 -0.17614837
5487   3.345  14.43324 3.112599e-04 0.4762233 -0.22082107
1851  -1.565 -14.35081 3.174521e-04 0.4762233 -0.22597133
514   -1.570 -14.31917 3.198711e-04 0.4762233 -0.22796837
```

How to read the output? The first column (without a name) is the row number in the original data. So, in this case, the most significant gene was the one on the row 1437 of the data matrix we gave to `lmFit()`. The next (logFC) reports a log2-based fold change between the groups. If the value is negative, it indicates down-regulation in the treatment group, and a positive value would indicate up-regulation in the treatment group. There are two statistics, t and B. The t is the moderated t-statistics from the empirical Bayes method, and the B is the log-odds that the gene is differentially expressed. The two p-value columns contain the raw p-value (P.Value) and the p-value corrected for multiple comparisons (adj.P.Val) using the Benjamini and Hochberg's false discovery rate.

7.2.1 Differential expression and p-values

The results above indicate that after correcting for multiple comparisons there were no significantly differentially expressed genes. However, this is probably not true, since the differences in expression among the top 10 genes vary from about 12X over-expression to 32X under-expression (the valuses are log2-transformed, hence -5^2 = -32X). The insignificant p-values just indicate that the sample size was too small to draw statistically reliable conclusions. The situation may change if more samples are gathered.

On the other hand, if the aim of the study is to show differential expression using the microarray data only, it is important to draw conclusions based on the adjusted p-values. If the aim is to find, say, 20 top genes that are to be verified using some other method, such as RT-PCR, then it does not matter which p-values you're using for selecting those top genes. The rank of the genes remains the same after the multiple testing correction.

7.2.2 Extracting the genes from the original data

One is often interested in seeing the original values of the data after the statistical testing. Let's say that we want to get the top 100 genes. We can do this by first getting the rownames of those gene from the `toptable()` result. Here we generate the toptable results for the best 100 genes only, and save their rownames in the vector `rn`.

```
rn<-rownames(toptable(fit, coef=2, n=100))
```

Or we can extract all the genes that have the unadjusted p-value at most 0.001. First we create the toptable result for all genes in the analyzed data set (here `dat.m`). Then we get from this toptable-result only the rows (genes) that have a p-value less than 0.001, and put their gene names in the vector `rn`. Note that the `toptable()` output is a matrix, so individual columns can be extracted using the $-notation - this is used for the p-value column in the example below.

```
tt<-toptable(fit, coef=2, n=nrow(dat.m))
rn<-rownames(tt)[tt$P.Value<=0.001]
```

In both cases object rn now stores the numbers of the rows of the original data matrix. Those names are actually stored in a character vector, but we need to convert it into a numeric vector first:

```
rn<-as.numeric(rn)
```

Then we can extract the genes from the original data:

```
dat.s<-dat.m[rn,]
```

Now object `dat.s` stores the data for the genes that were selected (as differentially expressed).

8 Gene set enrichment analysis

Term gene set enrichment analysis (GSEA) is used in several discrepant senses. First of all, it is the name of the tool from the Broad Institute, MIT. Second, GSEA is also used to describe all methods that are used for statistically testing whether genes in our list of interesting genes are enriched in some pathways or functional categories. Typically these methods employ hypergeometric test -based statistics. Third, approaches where genes are first assigned to pathways or categories and their statistical significance is tested using both the knowledge of the category and the expression data, are also called gene set enrichment analyses.

Although it is nowadays possible to use R for GSEA analyses in Broad Institute sense, we are not touching on that topic any further in this chapter. Rather, we introduce the two other mentioned approaches. We call the former approach gene set enrichment analysis and the latter approach gene set test according to the Bioconductor package it is implemented in.

Prior to gene set enrichment analysis a statistical test is typically conducted in order to find the statistically significantly regulated genes from the data. We assume here that such a test has been conducted using limma or by some other means. The results of the statistical test are stored in a matrix, which has a similar shape to the original data matrix got after normalization, but it contains less rows.

In contrast, gene set test takes the original, unfiltered data matrix as it come out after the normalization.

8.1 Gene set enrichment analysis for GO categories

Library GOstats implements some tools for gene set enrichment analysis. Before the analysis, microarray probe IDs need to be converted to EntrezIDs. EntrezIDs are gene-specific identifiers used by NCBI to cross-link different databases together. Fortunately, annotation packages produced in the Bioconductor project contain the mappings from probe IDs to EntrezIDs.

Conversion is rather simple. First an EntrezID is searched for every microarray probe ID using the annotation package. Those are initially contained in an environment (R's version of hash tables), which needs to be converted into a data frame. This requires a little bit of R gymnastics. First, the annotation package is loaded into memory, and the environment ENTREZID from the package is saved as a new object `allg`. Next, `allg` is converted from an environment to a data frame. This requires the following steps: converting the environment to a list, unlisting the list, and creating the data frame.

Last, we retain the unique EntrezIDs only for the genes in our list of interesting genes. We get the matrix of interesting genes (`dat.s`) from the data matrix created after the limma analysis. Let's assume that this is a HG-U133a array from Affymetrix, and the correct annotation package is thus hgu133a. The conversion proceeds as follows:

```
> library(hgu133a)
> allg<-get("hgu133aENTREZID")
> allg<-as.data.frame(unlist(as.list(allg)))
> myids<-unique(allg[rownames(dat.s),])
```

Now `myids` contains all the probes that we considered interesting.

The actual test is run in three steps, since the GO hierarchy consists of three distinct ontologies. These are biological process (BP), molecular function (MF) and cellular component (CC). First the hypergeometric test parameters are initialized using the command `new()`. It takes several arguments, such as the names of the interesting genes (`geneIds`), an annotation package (`annotation`), which ontology to test (`ontology`), p-value cutoff (`pvalueCutoff`), and whether to test over- or under-enrichment (`testDirection`). After initialization, the test is calculated using the command `hyperGTest()`. The following commands test all three ontologies, and store the results in objects `resultBP`, `resultMF` and `resultCC`. They all use the p-value of 0.05, which is a rather typical choise, and test for over-enrichment, which is also a typical choise.

```
> params<-new("GOHyperGParams", geneIds=myids,
+ annotation=c("hgu133a"), ontology="BP", pvalueCutoff=0.05,
+ conditional=FALSE, testDirection="over")
> resultBP<-hyperGTest(params)
> params<-new("GOHyperGParams", geneIds=myids,
+ annotation=c("hgu133a"), ontology="MF", pvalueCutoff=0.05,
+ conditional=FALSE, testDirection="over")
> resultMF<-hyperGTest(params)
> params<-new("GOHyperGParams", geneIds=myids,
+ annotation=c("hgu133a"), ontology="CC", pvalueCutoff=0.05,
+ conditional=FALSE, testDirection="over")
> resultCC<-hyperGTest(params)
```

Once the analysis is ready, you can check how many of the pathways were significant simply just printing the object to the screen:

```
> resultBP
```

A report can also be generated. Command `htmlReport()` generates from the test results an HTML-page that can be viewed using a web browser. It takes three essential arguments, the name of the object that contains the results of the gene enrichment analysis, name of the output file and a logical argument indicating whether to add (append) the results to end of the file, if the file already exist. The following commands save the results from all the three tests to the same file (hypergeo.html):

```
> htmlReport(resultBP, "hypergeo.html", append=T)
> htmlReport(resultMF, "hypergeo.html", append=T)
> htmlReport(resultCC, "hypergeo.html", append=T)
```

8.2 Gene set enrichment analysis for KEGG pathways

Analysis for KEGG pathways is very similar to the one outlined for GO categories above. Prior to the test, we need to generate a list of EntrezIDs as outlined above. These are saved in an object `myids`. The following command runs the test using the same settings that were used for the GO category test. The only difference is in the first argument, which is now `KEGGHyperGParams` instead of `GOHyperGParams`:

```
> params<-new("KEGGHyperGParams", geneIds=myids,
+ annotation="hgu133a", pvalueCutoff=0.05,
+ testDirection="over")
> result<-hyperGTest(params)
```

Similarly to the GO analysis, once the analysis is ready, you can check how many of the pathways were significant by printing the object to the screen. For these demodata, 5 KEGG pathways were significant with a p-value less than 0.05:

```
> result

Gene to KEGG  test for over-representation
49 KEGG ids tested (5 have p < 0.05)
Selected gene set size: 42
    Gene universe size: 2032
    Annotation package: hgu133a
```

And the report can be generated for the KEGG analysis the same way as for the GO analysis:

```
> htmlReport(result, "hypergeo.html", append=T)
```

8.3 Performing the gene set test

Gene set test is implemented in the Bioconductor package globaltest. In contrast to a simple enrichment analysis outlined above, gene set test takes both pathway information and expression data into account at the same time. Gene set test requires an unfiltered data set, otherwise the analysis in the form presented here will crash.

The testing procedure outlined below goes through all KEGG pathways of GO categories, and tests whether the genes that belong to those groups are statistically significantly under- or over-expressed as a group. In other words, the information of the grouping of the genes and their expression are taken into account at the same time.

8.3.1 KEGG pathways

First we need to extract the pathway information from the annotation package. If we assume that the chiptype is hgu133a, the pathway information can be extracted using the command `get()`, and then saved in an object `kegg` as a list as follows:

```
> library(hgu133a)
> pathway2probe<-get("hgu133aPATH2PROBE")
> kegg<-as.list(pathway2probe)
```

No matter what the annotation package is, the PATH2PROBE is always appended to the end of its name as above.

Now that we have the KEGG pathway information in a suitable format, we can perform the gene set test using the function `globaltest()`. It requires the normalized data (`dat.m`), knowledge of grouping of the samples (which are, say, controls and which are treatments, here a vector called `groups`, with zero coding for a control sample, and a one coding for a treatment sample), and the KEGG pathway information (in object `kegg`):

```
> library(globaltest)
> groups<-c(0,1,0,1)
> test.kegg<-globaltest(as.matrix(dat.m), groups, kegg)
```

The results (in object `test.kegg`) can be adjusted for multiple tests, also. Function `gt.multtest()` does the adjustment using the Benjamini and Hochberg's false discovery rate as follows:

```
> test.kegg<-gt.multtest(test.kegg)
> test.kegg<-sort(test.kegg)
```

Now, the object `test.kegg` contains the final results of the test. The simplest way to view the results is to print the object on the screen:

```
> test.kegg

Global Test result:
Data: 4 samples with 22283 genes; 6 gene sets
Model: logistic
Method: All 3 permutations

      Genes Tested Statistic Q Expected Q  sd of Q P-value FDR.adjusted
00900    12     12      24.168     13.708   9.0608 0.33333      0.69097
00020    44     44      94.983     47.008  41.5620 0.33333      0.69097
00190   153    153      34.256     17.413  14.5950 0.33333      0.69097
00290    18     18      20.136     10.156   8.6531 0.33333      0.69097
00281     6      6     219.300     86.895 114.8200 0.33333      0.69097
00750     6      6     137.310     58.205  68.6300 0.33333      0.69097
...

03050    36     36    8.586300    11.3870  2.9499       1            1
00563    31     31    5.004200     6.4045  1.4102       1            1
00785     2      2    0.045724     1.8137  1.7124       1            1
00471     9      9   17.853000    25.0310  6.5186       1            1
00860    51     51   22.184000    24.3290  1.8815       1            1
00520     6      6    4.998800    18.4790 11.7350       1            1
```

The first column contains the KEGG pathway identifiers. The Genes column and Tested column list the number of genes in each pathway and the number of genes in our list that belong to that particular pathway. Q statistic gives the observed and expected number of genes in each category. P-value column reports the raw p-values, and the FDR.adjusted column gives the FDR-values that are p-values corrected for the number tests.

We can also write out a report of the results. At first the following commands look a little bit hard to comprehend, but the four first commands just assemble a table in a suitable format so that it can be written out using the standard R function `write.table()`. The resulting tab-delimited text file (globaltest-result-table.tsv) contains KEGG pathway IDs and names, as well as p-values for all pathways.

```
> table.out<-data.frame(pathway=names(test.kegg),
+ pvalue=p.value(test.kegg))
> names(test.kegg)<-as.list(KEGGPATHID2NAME)[names(test.kegg)]
> table.out<-data.frame(table.out,
+ Description=names(test.kegg))
> table.out<-table.out[order(table.out$pvalue),]
> write.table(table.out, file="globaltest-result-table.tsv",
+ sep="\t", row.names=T, col.names=T, quote=F)
```

8.3.2 GO categories

The gene set test for GO pathways is slightly more complicated than the one for KEGG pathways. Getting the GO category information is similar to the KEGG procedure, but instead of PATH2PROBE after the annotation package name we use GO2ALLPROBES:

```
> library(hgu133a)
> go2allprobes<-get("hgu133aGO2ALLPROBES")
> go<-as.list(go2allprobes)
```

The actual test is carried out exactly the same way as for the KEGG pathways:

```
> test.go<-globaltest(as.matrix(dat.m), groups, go)
```

As is the multiple testing correction:

```
> test.go<-gt.multtest(test.go)
> test.go<-sort(test.go)
```

The results can be written on the screen the same way as the KEGG results:

```
> test.go

Global Test result:
Data: 4 samples with 22283 genes; 6 gene sets
Model: logistic
Method: All 3 permutations

           Genes Tested Statistic Q Expected Q  sd of Q P-value FDR.adjusted
GO:0004964     1      1     0.25324    0.20165 0.044674 0.33333      0.86419
GO:0030350     1      1     0.39569    0.13307 0.227430 0.33333      0.86419
GO:0046904     2      2     4.58120    1.69710 2.497700 0.33333      0.86419
GO:0030568     2      2     4.58120    1.69710 2.497700 0.33333      0.86419
GO:0042362     2      1     3.40470    2.65430 0.649810 0.33333      0.86419
GO:0042840     1      1    35.46000   24.50100 9.491300 0.33333      0.86419
...
GO:0001774     2      1    0.043965   0.043965        0       1            1
GO:0005766     2      1    0.043965   0.043965        0       1            1
GO:0045360     2      1    0.043965   0.043965        0       1            1
GO:0045362     2      1    0.043965   0.043965        0       1            1
GO:0032603     2      1    0.043965   0.043965        0       1            1
GO:0016263     1      1    3.561200   3.561200        0       1            1
```

The results table is read exactle the same way as the results table for KEGG analysis.

The largest difference lies in the procedure that puts the results in a suitable format for writing on a disk. For example, we need a small loop (command `for()`) for extraction of the GO category names from the result object. In the end, the output is a table similar to the one produced by the gene set test for KEGG pathways.

```
> test.go<-sort(test.go)
> test.go2<-test.go[1:x,]
> table.out<-data.frame(pathway=names(test.go2),
+ pvalue=p.value(test.go2))
> n<-c()
> for(i in 1:x)
>    n<-c(n, get(names(test.go2[i,]),GOTERM)@Term)
>
> names(test.go2)<-n
> table.out<-data.frame(table.out, Description=names(test.go2))
> table.out<-table.out[order(table.out$pvalue),]
> write.table(table.out, file="globaltest-result-table.tsv",
+ sep="\t", row.names=T, col.names=T, quote=F)
```

8.3.3 Extracting the genes from a particular pathway

The analysis is now ready, but we would like to know what are the genes in the category or pathway that come up on the top. The genes can be extracted using the pathway of category identifier. We will illustrate this using the KEGG pathway result. The pathway that was the most significant was 00900. To get from the pathway identifier to the genes, we have to extract the gene names from the object kegg that was created during the analysis above:

```
> genes<-kegg[["00900"]]
```

The notation used extracts an entry for 00900 from the list object `kegg`. Object `genes` now holds the affymetric gene IDs:

```
> genes

 [1] "208647_at"   "210950_s_at" "201275_at"   "217344_at"   "204615_x_at"
 [6] "208881_x_at" "209218_at"   "213562_s_at" "213577_at"   "202321_at"
[11] "202322_s_at" "217631_at"
```

That list of gene names can be turned into a list of annotations. This procedure is outlined in the chapter Annotating a genelist. Follow the chapter onwards from the point where the object `genes` was created.

9 Annotating a genelist

Annotating the genes, or in other words, combining the gene expression data with other knowledge, is typically carried out after statistical testing. Bioconductor project produces annotation packages for many chiptypes, and these can be directly used for annotating the results. As an input, the annotation process takes a vector of gene names. Those can typically be extracted from a matrix of limma results. Output of the process is a text or an HTML file containing the annotations.

9.1 Generating the report

First we need a list of genenames that we want to combine with other information, i.e., annotations. These can be very easily extracted from a matrix with the command `rownames()`, e.g.:

```
> genes<-rownames(dat.m)
```

or from the limma results:

```
> genes<-as.numeric(rownames(toptable(fit)))
> genes<-rownames(dat.m[genes,])
```

Now, the vector genes contains the gene identifiers.

The actual annotation process is implemented in package annaffy. Despite its name this package can be used for annotating other chiptypes also, as long as they have a valid annotation package. The annotation process consists of three steps. Selecting the annotation fields, constructing an annotation table and writing this table to an HTML file.

Here, we will select all available fields to be included in the annotations:

```
> library(annaffy)
> annot.cols<-aaf.handler()
```

Next the annotation table is built. Here the name of the annotation package is needed. As an example, we will annotate Affymetrix data (chiptype hgu133a). The first argument is a vector of genenames (here `genes`), the next is the name of the annotation package, and the last is the object that specifies which fields we want to include to the annotations:

```
annot.table<-aafTableAnn(genes, "hgu133a", annot.cols)
```

Building the annotation table might take several minutes, even closer to an hour, if the whole chip is being annotated.

Once the annotation table is ready, it can be written to a file using the command `saveHTML()` or `saveText()`. The first argument is always the annotation table created above, and the second is the name of the file the information should be saved in:

```
> saveHTML(annot.table, "annotations.html")
```

or:

```
> saveText(annot.table, "annotations.txt")
```

10 Clustering and visualization

Clustering is a very common analysis performed for DNA microarray data. The most often used clustering is hierarchical clustering, typically in a form of a heatmap. Another very common clustering is K-means. This chapter will present these two methods.

10.1 Heatmap

Heatmap presents hierarchical clustering of both genes and arrays, and additionally displays the expression patterns, all in the same visualization. Before visualization the genes and arrays need to be clustered. Clustering consists of two separate phases. In the first phase all pairwise distances a) between genes and b) between samples are calculated using a selected distance method. There's a plethora of different distance measures available, but probably the most used ones are Euclidean distance and Pearson (or Spearman) corralation. The choise of the distance measure affects the results, but there are usually no good reasons to select one over another. After calculation of distances, a tree construction method needs to be selected. The typical choise is average linkage (same as UPGMA), but other methods, such as single or complete linkage are also available.

10.1.1 Constructing a heatmap

Library amap offers some convenient functions for calculating the hierarchical clustering for both genes and chips. Command `hcluster()` can calculate the differences using any of the aforementioned distances. Here, we will use Pearson correlation (argument `method="pearson"`), but it can be changed to, e.g., Euclidean distance (argument `method="euclidean"`) or Spearman correlation (argument `method="spearman"`). For tree construction we use average linkage (argument `link="average"`), and it can also be changed to, e.g., complete linkage (`link="complete"`) or single linkage (`link="single"`).

Let's calculate a clustering first for the genes and then for the chips. Here we are clustering the normalized data, but we could equally well cluster filtered data (`dat.f`) or the differentially expressed genes (`dat.s`).

```
> library(amap)
> clust.genes<-hcluster(x=dat.m, method="pearson",
+ link="average")
> clust.arrays<-hcluster(x=t(dat.m), method="pearson",
+ link="average")
```

The normalized data can be so large that clustering all the genes (or arrays) becomes impossible. Clustering about 23000 genes takes about 1 GB of memory, and clustering 45000 genes would consume about 4 GBs of memory, and would not be feasible on a standard Windows workstation. If the genes really need to be clustered, the data can be sampled, and this sample is then clustered. This should convey approximately the same information as the clustering of the whole dataset.

Sampling is done in two phases. First we create a vector of numbers from 1 to the number of rows in the dataset. This vector is stored in the object `n`. Command `sample()` does the sampling. We sample the vector `n` to create a new vector `n.s` that contains the row indexes of the rows to be sampled from the original dataset. The sample size is here 10% of the original dataset. Then we create a new data set `dat.sample` that now contains randomly selected 10% of the original dataset. Last, we cluster the genes and arrays in the sampled dataset as already described above.

```
> n<-1:nrow(dat.m)
> n.s<-sample(n, nrow(dat.m)*0.1)
> dat.sample<-dat.m[n.s,]
> library(amap)
> clust.genes<-hcluster(x=dat.sample, method="pearson",
+ link="average")
> clust.arrays<-hcluster(x=t(dat.sample), method="pearson",
+ link="average")
```

Before visualizing the clustering results as a heatmap, we might want to think about the coloring of the image. The usual coloring scheme for microarray data in heatmaps is to present down-regulated genes with green, and up-regulated genes with red. This kind of a scheme can be generated with the command `colorRampPalette()`. The first argument specifies the color for the smallest observation (the most down-regulated gene), and second argument the color for the largest observation (the most up-regulated gene). The number after the command specifies how many different colors between the extremes should be generated, here we use 32 colors.

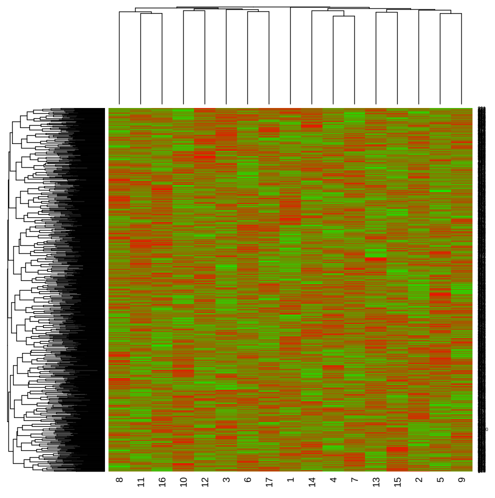
```
> heatcol<-colorRampPalette(c("Green", "Red"))(32)
```

Other coloring schemes may be generated by changing the names of the colors in the `colorRampPalette()`.

Finally, the heatmap can be generated using the command `heatmap()`. It takes four arguments: `x` specifies the dataset, which should be a matrix. Here we convert `dat.m` to a matrix to make sure that it really is a matrix. Arguments `Rowv` and `Colv` take the clustering results that were generated earlier. The clustering results need first to be converted to dendrograms with the command `as.dendrogram()` as shown here. The last argument `col` specifies the coloring scheme to be used while generating the image.

```
> heatmap(x=as.matrix(dat.m), Rowv=as.dendrogram(clust.genes),
+ Colv=as.dendrogram(clust.arrays), col=heatcol)
```

The resulting heatmap resembles something like the one below. Hierarchical clustering of genes and arrays are on the left side and on the top of the colored area, respectively. In the colored area, every gene is represented by a colored bar. Colors represent the down- (green) or up (red) -regulation of the genes.



10.2 K-means clustering

K-means clustering does not produce a tree, but divides the genes or arrays into a number of clusters. In contrast to hierarchical clustering, K-means clustering is feasible even for very large datasets. Before the analysis user has to specify how many clusters should be returned. Unfortunately, there are no good rules of thumb for estimating the starting number of clusters before the analysis. Therefore, the analysis proceeds by changing the number of clusters, checking the results, and finally picking the solution that appears to be reasonable. Optimality of the solution can be checked using the within clusters sum of squares (within SS), and the idea is to minimize the within SS, but not to overfit the data, in other words, not to use too many clusters. Analysis will always return the same number of clusters as the user specified before the analysis. The small caveat of K-means is that it might return a different results, even if run with the same parameters and the same dataset. Therefore, it has to be run several times for every number of clusters, just to make sure that a near optimal solution using that number of clusters is found.

10.2.1 Performing the K-means clustering

Command `kmeans()` calculates a K-means clustering result. It takes four arguments, `x`, name of the data object, `centers`, number of clusters to create, `iter.max`, iteration maximum, and `nstart` which specifies how many times the run should be performed using the same settings.

The analysis for the normalized dataset with 5 clusters is performed as follows. The analysis is repeated 10 times. Note that `kmeans()` wants to get a matrix, so we convert the data to a matrix with `as.matrix()`.

```
> km<-kmeans(x=as.matrix(dat.m), centers=5, iter.max=100000,
+ nstart=10)
```

The within SS can be extracted from `km`. Here we extract the cluster-specific within SS values, and take a sum of them:

```
> sum(km$withinss)
```

```
[1] 642.1721
```

Similarly, a numerical vector that specifies into which clusters the genes go after the K-means clustering can be extracted. The cluster assignment for the first gene of the data is the first in the cluster vector, also.

```
> km$cluster
```

The genes, say, in the first cluster can be easily extracted as a new dataset. Only the rows for which the `km$cluster` is equal to 1 are extracted from the normalized data. This is sometimes handy, if the cluster of genes warrants some further analyses.

```
> dat.c<-dat.m[km$cluster==1,]
```

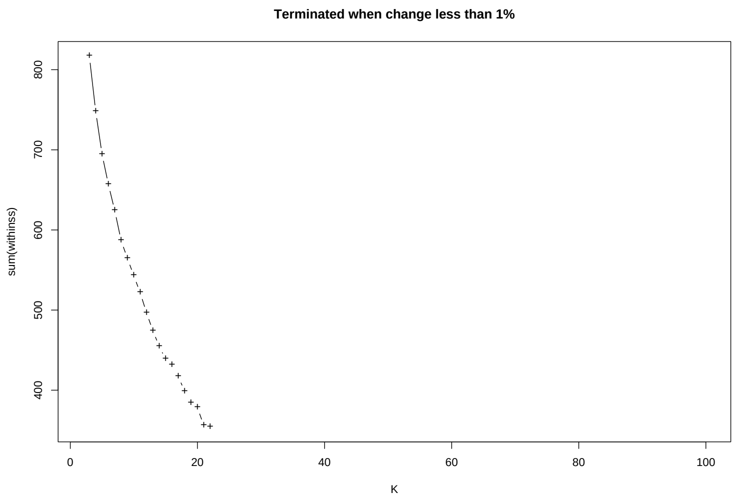
10.2.2 How to find the optimal number of clusters?

The optimal number of clusters is a slightly subjective matter. Within SS can be used for measuring the goodness of fit for the clustering, but simply searching for the minimum is not an viable option, since within SS will reach it's minimum when the number of clusters is equal to the number of genes or arrays, depending what we are clustering. Therefore, the K-means analysis is carried out using different numbers of clusters in each run. Then the number of clusters is plotted against within SS, and a point where the steep decent of the within SS starts to level off is the optimal number of clusters.

The whole analysis can be carried out by hand, but we present a small loop that performs the calculation automatically. The run is terminated when the within SS changes less than 1% when one more cluster is added to the analysis.

```
> # Test a maximum of 100 clusters
> kmax<-c(100)
> # If there are less than 100 genes or arrays
> # make the max. no. of cluster equal to the
> # number of genes or arrays
> if(nrow(dat2)<100) {
>    kmax<-nrow(dat2)
> }
> # Create an empty vector for storing the
> # within SS values
> km<-rep(NA, (kmax-1))
> # Minimum number of cluster is 2
> i<-c(2)
> # Test all numbers of clusters between 2
> # max. 100 using the while -loop
> while(i<kmax) {
>   km[i]<-sum(kmeans(dat2, i, iter.max=20000,
+   nstart=10)$withinss)
> # Terminate the run if the change in within SS is
> # less than 1%
>   if(i>=3 & km[i-1]/km[i]<=1.01) {
>      i<-kmax
>   } else {
>      i<-i+1
>   }
> }
> # Plot the number of K against the within SS
> plot(2:kmax, km, xlab="K", ylab="sum(withinss)", type="b",
+ pch="+", main="Terminated when change less than 1%")
```

The resulting image should look something like the one below:



Reading from the image, the optimal number of clusters is about 20, since then the line starts to level off.

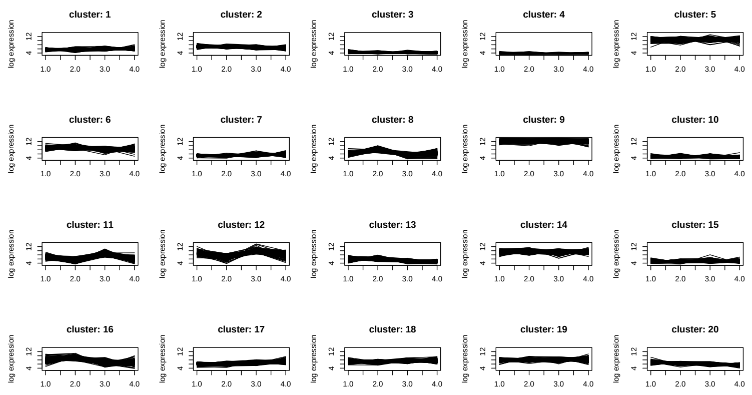
The final step would be to rerun the K-means analysis using 20 as the number of clusters. After getting the optimal result, the clustering can be visualized.

10.2.3 Visualizing the K-means clustering

K-means clustering is usually visualized by drawing the gene expression pattern across the arrays using a single line graph per cluster. The following loop-structure draws several smaller images inside a larger image. Every single subplot represent a cluster from the K-means clustering. First the plotting area (the larger image) is split into a number of subplots. Here we first take a square root of the number of clusters and create as many subplot rows and columns. The individual clusters are plotted from left to bottom of the plotting area. Command `matplot()` does the actual plotting. Every single gene is represented by one line in the plot. Often genes mask each others lines, but the idea is to get a general view of the pattern in the data, and it does not usually hinder this purpose. Object `km` holds the result from the K-means analysis performed by the command `kmeans()`.

```
> max.dat.m<-max(dat.m)
> min.dat.m<-min(dat.m)
> par(mfrow=c(ceiling(sqrt(k)), ceiling(sqrt(k))))
> for(i in 1:k) {
>    matplot(t(dat.m[km$cluster==i,]), type="l",
+    main=paste("cluster:", i), ylab="log expression",
+    col=1, lty=1, ylim=c(min.dat.m, max.dat.m))
> }
```

The resulting image should resemble this:



Now that the expression pattern has been visualized, genes from an interesting cluster can be extracted as already explained in more details above.

```
> dat.c<-dat.m[km$cluster==1,]
```

Part IV

Extras

11 Estimating the sample size

Sample size estimates have long been demanded for epidemiological studies. To have any change to get published, the epidemiological study has to have an estimate of the power of the study. This has not been the tradition of microarray research, but should be encouraged. Good experimental design is very much about good research practises. One might even state that bad experimental design is bad science, especially since it often leads to loss of money and effort and possibly human or animal suffering. For example, if we make an experiment using laboratory animals, using too a small sample size would probably lead into unconclusive findings, and in the worst case, the whole experiment might need to be started over again from the scratch. The animals sacrificed in the first, unconclusive experiment would have wasted their lives in vain, and repeating the experiment would lead to more animal suffering. The same applies to a too large sample size. Both cases are equally unacceptable. Every self-aware and ethics-keen researcher should therefore be familiar with methods of sample size extimation.

11.1 Current knowledge

To correctly estimate a sample size, estimates of expected effect size and variability, and desired false positive and negative rates should be available. Typically false positive rate of 0.05 (p-value) and false negative rate of 0.8 (power) are used. Estimates of expected effect size and variability can be guessed, or better still, estimated from previous studies. As a rule of thumb, if the estimated difference in gene expression between the groups (effect size) is large, smaller groups will suffice. If the difference is small, larger groups are needed to detect it. In other words, if even small changes need to be detected, it is necessary to use a larger sample size than if it is enough to detect only the larger changes.

Traditional methods are not very easy to generalize to cover DNA microarray experiments, although methods have been modified to better suit microarray research. For example, Yang et al. (2003) estimate that assuming the false discovery rate of 0.05, and power of 0.80, effect size of 2.0, and the number of selected genes of 50, the sample size that is needed is more than 10 but less than 30 per group. The number of genes that come up significant using the same amount of replicates and the same false discovery rate and power are highly dependent on the experiment (Pavlidis et al., 2003). The same phenomenan is noted by Han et al. (2004). It might therefore be better to base the sample size estimation on earlier studies using a similar biological material on the same chip type. Since much data is freely available in the microarray databases, this should not be overly complicated to perform.

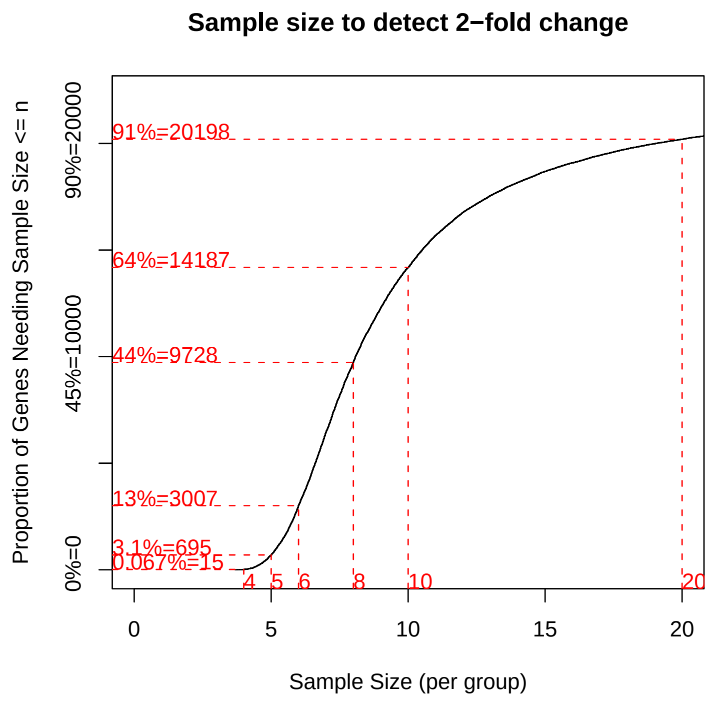
11.2 How to estimate the sample size?

One possibility to conduct a sample size estimation is offered in package ssize from the Bioconductor project. The method is applicable for comparison of two groups using t-test. The following example demonstrates the use of the method. Public dataset GSE2787 from the GEO database was retrieved, and the sample size to detect at least a two-fold expression change between the two groups at power 0.8 and significance level 0.05 was determined.

The following R code accomplishes the sample size estimation.

```
> library(genefilter)
> library(ssize)
> # Calculates the row-wise standard deviations
> sds<-rowSds(dat2.m)
> # Calculates sample size estimates for all genes
> size<-ssize(sd=sds, delta=log2(2), sig.level=0.05, power=0.8)
> # Creates a plot of the results
> ssize.plot(size, xlim=c(0,20), main=paste("Sample size to
+ detect 2-fold change", sep=""))
```

The results are shown in the figure below.



In order to detect 91% of the genes as differentially expressed, assuming that they have at least two-fold change, the sample size should be 20. The graph starts to level of after 10 replicates per group, and adding more replicates would probably mean an excess of replication. Therefore, for this dataset, 10 replicates per group could be a good tradeoff between the cost of the experiment and statistical power.

12 R at CSC

Jarno Tuimala

12.1 R is available in Murska

12.1.1 R versions

Several versions of R are available on CSC's server murska.csc.fi. These versions are all 64-bit version, and can be invoked with R combined with the version number, for example, `R272`. The advantage of using the 64-bit version of R is that it can utilize a maximum of 32 GBs of memory on Murska server.

12.2 Available libraries

12.2.1 The default selection

The selection of R and Bioconductor packages contains mainly the packages installed by default from CRAN and Bioconductor project. In addition, some DNA microarray annotation packages are installed. The current selection of packages can be checked directly from R using the command `library()`.

12.2.2 Installing new packages

Libraries for some specific jobs might be missing. R system administrators at CSC are Jarno Tuimala and Esa Lammi, and if any specific needs for libraries arise, they might be contacted.

Another possibility for installing missing libraries is to put them in the user's home directory on Murska. This is probably the fastest way to get new packages ready for use. Packages can be downloaded and installed in R using the command `install.packages()`. For example, downloading a package arules from CRAN and installing it into user's home directory can be done as follows:

```
> install.packages("ape", lib="/home/csc/jtuimala",
+ repos="http://cran.r-project.org")
```

Installing packages from Bioconductor site can be accomplished similarly as:

```
> install.packages("rama", lib="/home/csc/jtuimala",
+ repos="http://www.bioconductor.org")
```

The first argument in the `install.packages()` call is the name of the library. The next argument is the destination directory. The correct directory can be found out using the UNIX command `pwd()`. The last argument is the Internet address to the correct repository.

Sometimes you need to first download the package from the repository and then copy it to Murska for installation. For instance, this is the case with some alternative CDF environments for Affymetrix chips and some other specialized packages that are not available from the common CRAN or Bioconductor repositories. Packages can be copied to Corona using a secure FTP connection. Let's assume that package arules needs to be installed. The installation command in R is:

```
> install.packages("arules_0.5-0.tar.gz",
+ lib="/home/csc/jtuimala", repos=NULL)
```

Note that in contrast to previous installation commands, if you have downloaded the package by hand, you need to give its name in the installation command.

Some of the packages might not install without tweaking. In such cases system administrators might be contacted for help.

After package has been installed to the home directory, an R profile needs to be created. This is a simple text file with a single line giving the path where R should search for the installed packages. This file can be created in UNIX (quit R first) with a text editor. Open pico-editor (command `pico .Rprofile`) and type in just one line:

```
.libPaths("/home/csc/jtuimala")
```

The path inside the brackets should be identical to the path used in the `install.packages()` command. Save the file in your home directory (Ctrl + X in picoeditor).

After creating the .Rprofile file R should be able to find the libraries from your home directory.

12.3 Scripting on CSC server Murska

R jobs that take a maximum of one hour can be run interactively on Murska, although it is not a recommended practise. After an hour, the job is automatically cancelled, and the user is logged out. Longer jobs need to be submitted to a batch job queue. For that purpose you'll need a batch job file. A batch job file can be created using, for example, the pico editor on Murska. Let's create a simple batch job file called R-batch. It contains the following lines of code:

```
#!/bin/csh
#BSUB -L /bin/csh
#BSUB -e mrb1M_err_%J
#BSUB -o mrb1M_out_%J
#BSUB -N
#BSUB -M 4194304
#BSUB -W 168:00
#BSUB -n 1
cd $WRKDIR
R272 --no-save <<EOF
sink("R-batch.out")
print("Hello World")
sum(1:10)
sink()
EOF
```

The first eight lines starting with `#`-signs are used for reserving resourses from the server. For example, lines `#$ -l h_rt=2:00:00` and `#$ -l h_vmem=2G` reserve two hours of CPU time and 2 GBs of memory for the job. These can be changed to some other values, if your job takes a longer time or requires more memory. Typically, RMA or GC-RMA normalization for a few dozen or more Affymetrix microarrays requires much more memory. The maximum values for the CPU time and memory are 504 hours and 32 GBs. Using these values, even large Affymetric datasets can be preprocessed.

The middle line (`cd $WRKDIR`) changes directory to the working directory. All the datafiles and the batch job file we are preparing should be copied into that folder. Otherwise the run will fail.

The last six lines start R, and save all output to a file called R-batch.out. First, R prints "Hello world", and then calculates the sum of numbers between 1 and 10. After that R quits. All R specific commands are written between `<<EOF` and `EOF`, one R command per line.

Batch job is submitted to run with the UNIX command `bsub < R-batch`. Its state can be checked using the UNIX command `bjobs`.

After the job has finished, the results can be read from the file called R-batch.out. It should appear as follows (UNIX command less R-batch.out):

```
[1] "Hello World"
[1] 55
```