A quick and crash introduction to R with a bioinformatics bent

Ramon Diaz-Uriarte*

2016-10-21 (Release 1.2: Rev: 1815c02)

Contents

*Dept. of Biochemistry, Universidad Autónoma de Madrid, Spain, http://ligarto.org/rdiaz, rdiaz02@gmail.com

1 License and copyright

2 Scenarios

- You are designing an experiment: 20 plates are to be assigned (randomly) to 4 conditions. You are too young (or too old) to cut paper into pieces, place it in a urn, etc. You want a better, faster way. Specially because your next experiment will involve 300 units, not 20.
- The authors of a paper claim there is a weak relationship between levels of protein A and growth. However, you know that some of the samples are from males and some are from females, and you suspect the correlation is present only in males. The authors provide the complete data and you want to check for differences in correlation pattern between males and females.
- You've been working on a microarray study. For 100 subjects (50 of them with leukemia, 50 of them healthy) you have the $Cy3/Cy5$ intensity ratios for 300,000 spots. You just got the email with the compressed data file. You are leaving for home. In less than five minutes you'd like to get a quick idea of what the data look like: maximum and minimum values for all spots, average for 5 specific control spots (corresponding to probes 10, 23, 56, 10,004, 20,000), and a quick-and-dirty statistical test of differences for two specific probes, probe 7000 and 99,000, that correspond to two well know genes.
- Tomorrow you'll look at the data in more detail. For a set of 20 selected probes you will want to: a) take a look at the mean of the intensity, variance of intensity, and the mean of the intensity in each of the two groups; b) plot the intensity vs. the age of the subject; c) plot the log of the intensity vs. the age of the subject.

For each of those problems, would you ...

- Know how to do it?
- Do it quickly?
- Save all the steps of what you did so that 6 months from today you know **exactly** what you did, can repeat it, and apply it to new data?

This course is a quick introduction to an "environment for statistical computing and graphics" that will allow you to carry out each of the above.

3 This document and how to use it

This document is to be used a crash and relatively quick introduction to R, with a clear Bioinformatics bias[1].

The structure and logic is as follows:

- First, with the scenarios above, we try to motivate you.
- I then (section 4) show a six-line real example of a common problem in Bioinformatics (multiple testing). You might not understand much of what is done, but I will explain it in class (this is not a textbook, but a document for a class).
- We next go over a few practical things we need to get out of the way (sections 5 and 6).

[1] OK, there is an example with birds, reptiles, metabolic rates, body size, etc, that is not really bioinformatics but ... it is a neat data set and matches with some of my early scientific loves.

- We then (sections 7 to 10) cover in some detail what are the main objects of R (vectors, data frames, matrices), how to manipulate them and some plotting. This can be boring, and this used to be after section 11 but some students argued strongly that they'd rather see this first, so this comes now first.
- After that, we jump into R with three longer examples (section 11). Again, on first reading you might not understand all of what is done, but we will go over it in class.
- Then we cover tables and a little bit of programming (sections 12 and **??**).
- We then revisit some examples.
- If you understand all that is done up to that point, you should have a decent working understanding of how to use R, and can move on your own.
- But you do not want to skip section 18: this section covers debugging. Being able to debug quickly and painlessly is essential for using R effectively (and enjoying it even while debugging).
- Finally, in section 20 I include several longer commented examples. These should bring together many of the features we have mentioned, but also introduce new functions, and will give you practice with programming and debugging.

The material has been ordered that way on purpose. Yes, expect some frustration when working through sections 4 and 11, but definitely look at them before class and try to understand what is going on. After section 11 things should be smoother (but more boring, until we get to section 20). And here, definitely, you **must** type things on your own and understand the output. I have tried to use a kind of spiraling lay out, working over several things repeatedly, and repeating and going deeper on some key ideas. Hopefully, this will allow you to understand the material better, connect it with other pieces, and retain it for longer.

3.1 The PDF and the code

The primary output of this document is a PDF.However, all the original files for the document are available (again, under a Creative Commons license —see section 1) from `https://github.com/rdiaz02/R-bioinfo-intro`. (Note that in the github repo you will not see the PDF or R-bioinfo-intro.R files, since those are derived from the Rnw file).

For many commands I do not show the output (e.g., because it would just provide boring and space-filling output). However, make sure you type and understand it. You can copy and paste, of course, but I strongly suggest you type the code and change it, modify it, etc.

3.2 Other files you need in addition to this one

You should have (or should get) the following files:

- `hit-table-500-text.txt`
- `AnotherDataSet.txt`
- `anage.RData`
- `lastExample.R`
- `Condition_A.txt`, `Condition_B.txt`, `Condition_C.txt`

- `R-bioinfo-intro.R`

All of the files above (except the last) are mentioned or used in this document. What about `R-bioinfo-intro.R`? That is all the R code used in this document.

3.3 R and Bioinformatics

If you are reading this document, it is probably because you already have some idea of what R is. So no long details here. A summary is "R is a free software environment for statistical computing and graphics." (http://www.r-project.org/) and "R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. " (http://cran.r-project.org/).

Virtually all of the statistical analysis done in Bioinformatics can be conducted with R. Moreover, "data mining" (which is, according to some authors, simply "statistics + marketing") is well covered in R: clustering (often called "unsupervised analysis") in many of its variants (hierarchical, k-means and family, mixture models, fuzzy, etc), bi-clustering, classification and discrimination (from discriminant analysis to classification trees, bagging, support vector machines, etc), all have many packages in R. Thus, tasks such as finding homogeneous subgroups in sets of genes/subjects, identifying genes that show differential expression (with adjustment for multiple testing), building class-prediction algorithms to separate good from bad prognosis patients as a function of genetic profile, or identifying regions of the genome with losses/gains of DNA (copy number alterations) can all be carried out in R out-of-the-box (see BioConductor and CRAN).

[A proselitizing note] R is free software, meaning "free" as in free speech (not free as in free beer; in Spanish, free as in "libre", not free as in "gratis"). The definition of free software is explained, for instance, in http://www.gnu.org/philosophy/free-sw.html. Why does it matter that R is free software? For one thing, it makes your access to it simple and easy. As well, you can play with the system and look at the inside (you can look at the original code) and do with that code a variety of things, including modifying it, learning from it, etc. In addition, that R is free software is, arguably, one of the reasons of its incredible success (and, for instance, one explanation for why there are over 6000 contributed, and free software, packages). Moreover, Bioinformatics, as we know it, would not exist without free software. Newton, and others before him, used the expression "standing on the shoulders of giants" when explaining how the development of science and other intellectual pursuits builds upon past accomplishments; in Bioinformatics (and many other fields), we are also standing on the shoulders of millions of lines of free software.

3.4 Some references

When you download R you also download "An introduction to R", which is an excellent intro. There are many freely available documents (of variable quality, of course) here: http://cran.r-project.org/other-docs.html. Many books are listed (and some briefly commented) here: http://www.r-project.org/doc/bib/R-books.html.

This is a partial list a books I like and use when preparing classes:

Programming As it says, just focus on programming R:

- *R Programming for Data Science*. Peng. (This is an ebook and PDF, and you can pay whatever you want for it.)
- *Advanced R*. Wickham. (If you go to the web page for the book, in github, you can download the complete sources and build your own pdf).

- *The art of R programming*. Matloff.

Stats and some programming Introductory statistics (or introductory data science) with some programming interleaved.

- *Introductory statistics with R, 2nd ed*. Dalgaard.
- *R in Action*. Kabacoff. (A second ed. available since June 2015).

Linear models et al. Linear models are fundamental in statistics. And fascinating.

- *An R companion to applied regression*. Cox and Weisberg. John Fox is also the author of an excellent textbook (now in its second edition) about linear models. This companion is absolutely fantastic (and can be used even if you don't have the other textbook). You probably want this book.
- *Regression modeling strategies, 2nd ed*. Harell. Among its many virtues, this book contains excellent discussions of the problems of variable selection.
- Faraway has two books on linear models with R, both published by CRC. Wood is the author of a great book on Generalized Additive Models (also CRC). Etc, etc.

Machine learning Machine learning, classification, etc. And many of the examples are bioinformatics-inspired.

- *Applied predictive modeling*. Kuhn and Johnson.
- *An introduction to statistical learning*. James, Witten, Hastie, Tibshirani. The PDF of the book is available for download from their web page.

There are of course many others (including classics such as the two by Venables and Ripley, or Chamber's *Software for data analysis*, many specific to some fields, several devoted to graphics, etc, etc). There is also a (short) list of books I think are not worth it; ask me about them in class.

4 This will not be mysterious at the end of the course

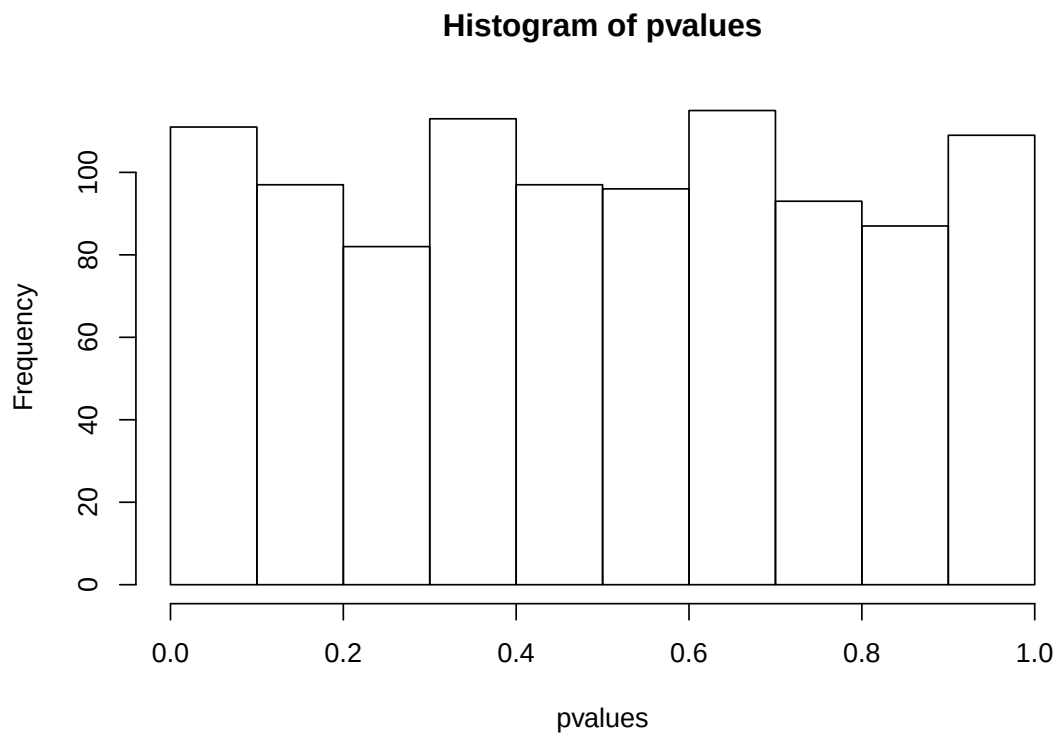
(This is an example we go over in section 16, p. 74, with a different number of genes).

We might have heard about the multiple testing problem with microarrays: if we look at the p-values from a large number of tests, we can be mislead into thinking there is something happening (i.e., there are differentially expressed genes) when, in fact, there is absolutely no signal in the data. Now, you are convinced by this. But you have a stubborn colleague who isn't. You have decided to use a simple numerical example to show her the problem.

This is the fictitious scenario: 50 subjects, and of those 30 have cancer and 20 don't. You measure 1000 genes, but none of the genes have any real difference between the two groups; for simplicity, all genes have the same distribution. You will do a t-test per gene, show a histogram of the p-values, and report the number of "significant" genes (genes with $p < 0.05$).

This is the R code:

```
randomdata <- matrix(rnorm(50 * 1000), ncol = 50)
class <- factor(c(rep("NC", 20), rep("cancer", 30)))
pvalues <- apply(randomdata, 1,
                 function(x) t.test(x ~ class)$p.value)
hist(pvalues)
```




```
sum(pvalues < 0.05)

## [1] 48
```

The example could be made faster, you could write a function, prepare nicer plots, etc, but the key is that in six lines of code you have settled the discussion.

Let's try to understand what we did. But first, we need to install R, and maybe some additional packages.

5 Very basics of using R

5.1 Installing R

Go to CRAN, http://cran.r-project.org/. Now, if you know what source code is, and you want to compile R, go to Sources (http://cran.r-project.org/sources.html). Otherwise, just download a binary for your operating system (http://cran.r-project.org/bin/).

- For Linux, most distros have pre-built binaries, so with Debian use apt-get install r-base r-base-dev, with Fedora and RH yum install whatever, etc. There are instructions in the CRAN page if you need them, though, for many distros.

  However, if you use Ubuntu, please read the instructions in http://cran.r-project.org/bin/linux/ubuntu/README.html, since the default Ubuntu packages can be outdated.

- If you use Windows, you want to install "base". It says so clearly: "Binaries for base distribution (managed by Duncan Murdoch). This is what you want to install R for the first time."

- If you use Mac, if you play with installation options, note that you need to install the tcl/Tk X11 libraries. If you run into trouble, make sure to read the FAQ (http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html).

- However you do it, please make sure you have a recent version of R.

5.2 Installing RStudio

There are a variety of ways of interacting and using R. For ease, and because it is a really nice piece of software, we will use RStudio. We want to use the "Dektop", that you can download from here: http://www.rstudio.com/products/rstudio/download/.

5.3 Editors and "GUIs" for R, et al.

Ah, this is a nice topic for a long, passionate, conversation. In this course, we will, by default, be using RStudio. I will, however, often use Emacs + ESS (ess.r-project.org/). For those used to Eclipse, there is a plug-in designed to work with R: StatET (http://www.walware.de/goto/statet). Another popular interface is JGR (http://www.rforge.net/JGR/). RKward (http://rkward.sourceforge.net/wiki/Main_Page) is also popular in some places (this was originally Linux-only, but not anymore). Some Mac users are very happy just the default, plain, interface provided by R under Mac OS X. And some Windows users like Tinn-R (http://nbcgib.uesc.br/lec/software/editores/tinn-r/en); I used to use Tinn-R in R courses I taught 8 to 10 years ago, but I think it has lost ground to RStudio, and it is only Windows. If you love vim, there is Vim-R-plugin (http://www.vim.org/scripts/script.php?script_id=2628; http://manuals.bioinformatics.ucr.edu/home/programming-in-r/vim-r). And then, there are many other options (an outdated list is available from http://www.sciviews.org/_rgui/, and some other entries around in internet land are http://www.theusrus.de/blog/r-guis-which-one-fits-you/ and http://stats.stackexchange.com/questions/5292/good-gui-for-r-suitable-for-a

If you plan to spend a fair amount of time doing Bioinformatics, then you'll spend a fair amount of time programming, probably using a variety of languages (R, Python, C, Perl, Java, PHP, etc). Becoming used to a programmer-friendly editor that "understands" all of the languages

you use is thus worth it. Choosing an editor is a highly personal issue. Emacs is an editor and then a lot of other things (that is what I use, for programming, editing text, email, etc); if you use Emacs then Emacs + ESS is the perfect combination for you. For vim users, there is the vim-R-plugin. Those who come from the Java world might be familiar with Eclipse (and, thus, you'll want to give StatET a try). Kate is another great editor that understands many editors and it easy to submit code to an R process running in the terminal, but it lacks some nice features that RStudio and Emacs+ESS have (but RKward might then be a nice option). Some people (myself) like to use a single editor for most/all editing tasks. Some other people jump around (they use RStudio for R, Eclipse for Java, and maybe Kate for Python). You get to choose.

Note, though, that one thing is syntax highlighting (and syntax highlighting for R is available for many, many editors) and another is the ability to interact with an R session, provide shortcuts for displaying help, offering object browsers, etc. Of course, you are the one who must weight the choices.

The summary (highly biased?): I definitely prefer Emacs (+ ESS), but in this course I will not attempt to teach you Emacs + ESS. So if you do not know Emacs, then try RStudio, which is what we will "officially" use. However, if you like Eclipse, then use Eclipse with StatEt. If you like Kate, use Kate, etc, but I might not be able to help you.

Note that all of the above have a different purpose from R Commander (http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/) which, as it says, is a basic statistics GUI for R. In this course we will rarely (if at all) use R Commander, since these notes are focused on programming and using R from the command line. However, I do recommend that you play around with R Commander. Another GUI for statistics with R (that I have not used but know is liked by some people) is Deducer: http://www.deducer.org.

5.4 Installing R packages

Most "for real" work with R you do will require installation of packages. Packages provide additional functionality. Packages are available from many different sources, but possibly the major ones now are CRAN and BioConductor.

If a package is available from CRAN you can do

```
install.packages("car")
```

(for example — this installs the `car` package and its dependencies).

If you want to install more than one package you can do (don't execute the code below as we will not use those packages)

```
install.packages(c("RJaCGH", "varSelRF"))
```

In Bioinformatics, BioConductor (http://www.bioconductor.org) is a well known source of many different packages. BioConductor packages can be installed in several ways, and there is a semi-automated tool that allows you to install suites of BioC packages (see http://www.bioconductor.org/install/). For example, go to http://www.bioconductor.org/packages/release/bioc/html/limma.html and see how instructions are clearly given there.

As we said above, sometimes packages depend on other packages. If this is the case, by default, the above mechanisms will also install dependencies.

With some GUIs (under some of the operating systems) you can also install packages from a menu entry. For instance, under Windows, there is an entry in the menu bar called **Packages**, which allows you to install from the Internet, change the repositories, install from local zip files, etc. Likewise, from RStudio there is an entry for installing packages (under "Tools").

Packages are also available from other places (RForge, github, etc); you will often find instructions there.

Now, make sure you install package "car", which we will use below:

```
install.packages("car")
```

(or do it from the menu of RStudio).

How do you find a package? Looking at a list of 6000 things in CRAN and another thousand in BioC is not a good idea. In addition to google et al., there are task views in CRAN: http://cran.r-project.org/web/views/, and there is a not too dissimilar thing in BioC. In addition, `findFn`, from package "sos" can help (see section 6.2).

5.5 Starting R

If you use RStudio, just start RStudio (icons should have been placed wherever they are placed in your operating system, or start if from the command line if you know how to/like to do that). From what I've been told, RStudio should be available from the menus in your desktop, in Windows, Linux, or Mac OS.

If you use other systems (Emacs + ESS, Eclipse, RKward, Kate, etc) just use the appropriate procedure (I assume that if you are using any of these you know what to do).

5.6 Stopping R

You can always just kill RStudio; but that is not nice. In all systems typing `q()` at the command prompt should stop R/RStudio. There will also be menu entries (e.g., "Quit RStudio" under "File", etc).

```
q()
```

Say no to the question about saving the workspace.

What if things hang? Try `Control-C` and/or `Esc`.

6 The R console for interactive calculations

In what follows, I will assume that you are either running R from RStudio, or that you know your way around and are using some other means (e.g., directly from the R icon in Windows, or from Emacs + ESS in any operating system, or using Eclipse, etc).

Regardless of how you interact with R, once you start an interactive R session, there will always be a console, which is where you can enter commands to have them executed by R. In RStudio, for instance, the console is usually located on the bottom left.

Now, move to the console and at the prompt (which will often start with >) type "1 + 2" (without the quotes) and press `Enter`:

```
1 + 2

## [1] 3
```

(All the code for this document is available, so you can copy and paste from the original code directly. If you copy code from other documents, say a PDF, that show the prompt, do not copy the prompt itself. That should not be an issue in this document, though, as the code sections do not show the prompt).

Look at the output. In this document, code chunks, if they show output, will show the output preceded by `##`. In R (as in Python), `#` is the comment character. In your console, you will NOT see the `##` preceding the output. This is just the way it is formatted in this document.

Note also that you see a `[1]`, before the `3`. Why? Because the output of that operation is, really, a vector of length 1, and R is showing its index. Here it does not help much, but it would if we were to print 40 numbers:

```
1:40
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40
```

Now, assign `1 + 2` that to a variable:

```
v1 <- 1 + 2
```

(you can also use = for assignment, but I prefer not to).

And now display its value

```
v1
```

```
## [1] 3
```

If you want to be more verbose, do

```
print(v1)
```

```
## [1] 3
```

Alternatively, you could surround the expression in parentheses:

```
(v1 <- 1 + 2)
```

```
## [1] 3
```

and that makes the assignment AND shows you the value just assigned to `v1`.

Finally, you could do

```
v1 <- 1 + 2; v1
```

```
## [1] 3
```

thus separating the two commands with a `;`, though that is rarely a good idea except for very special cases.

It is also possible to break commands, if it is clear to R that the expression is not yet finished:

```
v2 <-  4 - ( 3 * [Enter]
2)
```

You will see a + that indicates the line is being continued: R is still expecting more input (in this case, you must close the parenthesis and add something after the $*$). But sometimes things get confusing. You can bail out by typing `Ctrl + c` (Unix) or `Escape`, and abort the calculation.

Of course, use parenthesis as you think appropriate to make the meaning of an expression clear. R uses, for the usual functions, the usual precedence rules. If in doubt, use parentheses.

```
v11 <- 3 * ( 5 + sqrt(13) - 3^(1/(4 + 1)))
```

By the way, if you want to modify partially what you typed, you can repeat the previous commands with the up-arrow (↑) in RStudio (or Alt-p in ESS); and then move around also using ↑ , etc. You also have tab completion: if you get at the prompt, type `v` and press tab you should be given a bunch of options (that include v1 and v2, plus several functions that start with "v").

6.1 Naming variables

We created `v1` and `v2` above. Names of variables in R must begin with a letter (also a period, though this will make them hidden). Then you can mix letters, numbers, `.` and _. Variable names are case sensitive, so `v1` and `V1` are different things.

Once you have something in a variable, you can just use it instead of that something:

```
v3 <- 5
(v4 <- v1 + v3)

## [1] 8

(v5 <- v1 * v3)

## [1] 15

(v6 <- v1 / v3)

## [1] 0.6
```

Newer assignments silently **overwrite** previous assignments:

```
(z2 <- 33)

## [1] 33

z2 <- 999
z2

## [1] 999

z2 <- "Now z2 is a sentence"
z2

## [1] "Now z2 is a sentence"
```

You can delete a variable

```
rm(z2)
```

6.2 Getting help

Look at one help page:

```
help(mean)
```

Now, shorter:

```
?mean
```

Now let's use the help to l ,earn about the help system (and yes, read or take a quick look at it):

```
?help
?apropos
```

Now, try

```
?normal
?rnorm
apropos("normal")
apropos("norm")
help.search("normal")
```

Many help files include executable code (examples)

```
example(rnorm)
example(graphics)  ## will give an error
example(lm)
```

and note how you get to see the code that produced the figures. example.

Some help files include demos

```
demo(graphics)
```

again, note how you get to see the code that produced the figures. example.

And some include both

```
demo(persp)
example(persp)
```

But there are many other ways of searching for help about how to do something with R. Of course, you can google around, use stackoverflow, etc. There are mailing lists for R and for specific interest groups in R.

There is a package, "sos" (http://cran.r-project.org/web/packages/sos/index.html), that can help you search functions, etc, in packages that you do not have installed, ranks search results, etc. It is well documented (see http://cran.r-project.org/web/packages/sos/vignettes/sos.pdf). The only problem I see is that only some of the BioConductor packages are among those searched (and you need an internet connection).

Patrick Burns has a interesting blog entry about R navigation tools: http://www.burns-stat.com/r-navigation-tools/.

Oh, by the way, RStudio includes an integrated help browser. Use it if you use RStudio.

6.3 Error messages

The best way to learn to use R is to use it. As explained before, mistakes are harmless, so you should play and experiment. However, there are two key attitudes that will make your learning a lot faster: first, using the help system, and second **paying attention to the error messages**. Yes, the error messages are written in English, not some weird, unintelligible language. Sometimes they are a little bit cryptic, but more often than not, if you read them carefully, you will see how to approach to problem to fix the mistake, or will realize that what you typed makes no sense.

Lets look at a few. These are not representative or common or anything like it. But you should read them, understand them, and think about how to take corrective action (or realize that I was trying to do something silly).

```
apply(F, 1, mean)
log("23")
rnorm("a")
lug(23)
rnorm(23, 1, 1, 1, 34)
x <- 1:10
y <- 11:21
plot(x, y)
lm(y ~ x)
z <- 1:10
t.test(x ~ z)
```

6.4 Coding style

You write R code for the computer to do something, but that code should be readable by humans (including not only other people besides yourself, but yourself in the future). Please, make sure your code is tidy and respects some minimal rules of civility. In particular:

- Do not extend beyond column 80.
- Use spaces appropriately; for example, write `x <- rnorm(3, mean = 2)` and NOT `x<-rnorm(3,mean=2)` . Thesecondformisclearlyveryhardtoread.

There are many other possible coding style guides, but the above two for me are basic (if I grade code written by you, I will take into account respect of the above rules). This is not my particular silly snobbery: look at the code in the base R distribution, or look at the code in classics such as "Modern applied statistics with S" or "S programming"(Venables and Ripley), or "Software for data analysis" (Chambers), or .... Programming environments (e.g., Emacs + ESS) will offer ways of tidying your code, and there is even a package that can help you do it (http://cran.r-project.org/web/packages/formatR/index.html, http://yihui.name/formatR).

7 Entering data into R and saving data from R

There are many ways to load data into R (for example, see the book by P. Spector, or the "R Data Import/Export" manual http://cran.r-project.org/doc/manuals/R-data.html). Here we will only use `read.table`.

```
X <- read.table("hit-table-500-text.txt")
head(X)
## We could save what we care about in variables
## with better names
align.length <- X[, 5]
score <- X[, 13]
```

To see a slightly different example, open `AnotherDataSet.txt`. Now do:

```
another.data.set <- read.table("AnotherDataSet.txt",
                               header = TRUE)
summary(another.data.set)

##    ID          Age        Sex          Y
##  a1 :1   Min.   :12.0   F:3   Min.   :22.00
##  a2 :1   1st Qu.:13.0   M:2   1st Qu.:23.40
##  a3 :1   Median :14.0         Median :24.30
##  a7 :1   Mean   :14.8         Mean   :24.14
##  b15:1   3rd Qu.:16.0         3rd Qu.:25.00
##          Max.   :19.0         Max.   :26.00
```

Notice that we used the variable names (and took those names from the header), and the object is not a matrix, but a data frame (we will see this later).

7.1 But where are those files?

Of course, for R to read those files, you need to tell R **exactly** where those files are located. This is always the source of a lot of grief, but is really simple. These are some cases and ways of dealing with them:

1. The file you are trying to read lives exactly in the same working directory where R is running. OK, easy: just read as in the examples above.
   - How do you know what is the working directory where R is running? Type `getwd()`.
   - How do you know where the file you want to read is? Eh, this is up to you! You should know that (or ask your operating system or search facilities for it).
2. The file you are trying to read **DOES NOT** live exactly in the same working directory where R is running. You can either:
   (a) Tell R where the file is: specify the full path. Suppose your file, "f1.txt", is in "C:/tmp". Then, say `X <- read.table("C:/tmp/f1.txt")`.
   (b) Move R's working directory to the place where your files live. Two ways:
      i. Use `setwd("someplace")`, where "someplace" is the place where your files live.

ii. Under RStudio, go to "Session", "Set working directory" (which, in fact, is just a call to `setwd`)

This is all there is to it. And if you make a mistake, R will let you know.

Now, under Windows the true names of directories can be a mysterious thing (specially if you have things displayed in a language that is not English, and even more if you use directory names with spaces, accents, or other characters —e.g, Cyrillic). So **avoid** directories with spaces, accents, and other non-ASCII characters, and try to keep them under 8 characters (though that might not be a strong limitation nowadays). And try to place things in directories that Windows is unlikely to rename (e.g., `C:\Files-p1` is better than `C:\Archivos de Programa\Manolo Perez\Mis documentos`).

Avoiding spaces, accents and other non-ASCII characters is also a good idea under Unix/Linux (though here there is no problem with file and directory names that are very long).

Now, for the rest of the course, I will assume that you know where your data files are, the scripts are, etc. Where you place them depends on what you want (and the permissions you have in the computer you are using). You will be using either of the approaches explained in 7.1. It is up to you. In this class, I will often be running R in the very same directory where the data files and scripts are located. (You can assume that I have, sometime in the past, issued a `setwd` command.) This is just convenient.

7.2 Missing values

And what happens with missing values? Try running the examples above after doing this:

- Substitute a value by "NA" (without the quotes).
- Substitute a value by nothing; in other words, just delete a value (but not the character for separating columns). (Beware that in this case you often will want to be explicit about the separator in `read.table`.)

You can specify the character that R should interpret as a missing value, but the above two procedures are standard. And when you do either of the above, in the data that is read you should see a "NA". The best is, as usual, to be explicit: use an "NA" in your original data, or use some other special character string to identify them.

7.3 Very large data sets

Yes, R can deal with huge data sets. You just don't want to read them with `read.table`, or at least you do not want to use `read.table` without helping it recognize the types of columns, etc, etc. Look specially at the help for `scan`, try data base solutions, etc. (See the book by P. Spector, or the "R Data Import/Export" manual http://cran.r-project.org/doc/manuals/R-data.html). For even larger things, of specialized uses, there are packages such as `ff` or `bigmemory`.

7.4 Saving tables, data, and results

How can you save data, results, etc? Saving data in matrix or tabular form is easily done with `write.table`.

```
write.table(another.data.set,
            file = "the.table.I.just.saved.txt")
```

Open that file in an editor of your choice.

You can also save part or all the output from a session. You can copy and paste, or you can use commands such as `sink`.

Of course, similar considerations apply here as in section 7.1: think where you want to save things.

7.5 Saving an Rsession: .RData

And how can you save all you have been doing? The simplest way is to use `save.image`. Please, look at the help for that command. We will use a simple example:

```
save.image(file = "this.RData")
getwd()
```

Note where that file is saved (in the current working directory, which is what `getwd()` tells you).

Now open another R. Go to the directory where `this.RData` is. And do:

```
ls()

## character(0)
```

The above tells you what you have in your "working environment". There is nothing in there, since we just started. Now, do:

```
load("this.RData")
ls()
v1
v11
summary(another.data.set)
```

So all the stuff we had before is available in the new R session. (Now that we are done with this example, close the R session you just opened).

Now, lets try a different example. Do:

```
save.image()
```

And now open a new R in that directory. What happens? (Try doing `ls()` or `summary(another.data.set)`).

So be careful with this: you can end up using stuff you didn't know was there!!!! (The truth is that we were told what happened: did you notice the "[Previously saved workspace restored]"?).

And, by the way, do you understand what R tries to do when it asks "Save workspace image"?

Oh, and please go and look at the differences between `save.image` and `save`.

8 Scripts and non-interactive runs

8.1 Why use scripts

Keeping all of your code in one or more script(s) and evaluating the code from the script (instead of directly on the R console) has a couple of benefits:

- It is a complete record of all you did. And you can keep it nicely organized, with comments, etc.
- It allows you to carry out non-interactive calculations. For example, running a very long analysis, or re-running completely all the analysis and plots if you made a mistake, or new data are added, etc.

Thus, keeping all of your analysis in scripts is a fundamental step in **reproducible research**. For this section, create a very simple script typing this in a file and saving it as "script1.R":

```
x <- 1:100
print(mean(x))
plot(x)
```

8.2 Paths: where are scripts located

Before you can tell R to use your script or read some data, you need to tell R where, exactly, to find the scripts/data. Re-read again what was explained in section 7.1.

Now, for the rest of the course, I will assume that you know where your data files are, the scripts are, etc. Where you place them depends on what you want (and the permissions you have in the computer you are using). You will be using either of the approaches explained in 7.1. It is up to you. In this class, I will often be running R in the very same directory where the data files and scripts are located. (You can assume that I have, sometime in the past, issued a `setwd` command.) This is just convenient. (Yes, this is the same paragraph as above. I am repeating it on purpose.)

8.3 Using a script

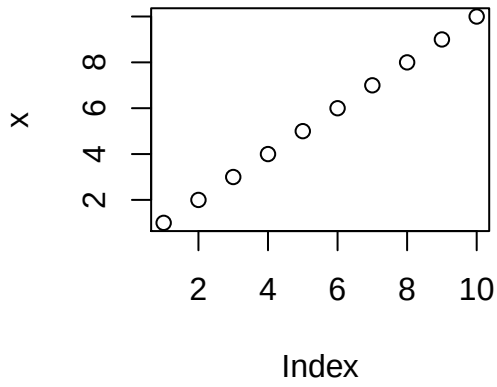
There are two basic ways of using a script:

Interactively What we have been doing so far. RStudio, Emacs, whatever, has a window (buffer, in Emacs parlance) with the code, and you select pieces of it and submit them to the R interpreter, running in the R console.

Non-interactively Two ways again:

Using source from a running R You have R running and do: `source("script1.R")`. I often add a couple of options:

```
source("script1.R", echo = TRUE,
       max.deparse.length = 999999)

##
## > x <- 1:10
##
## > print(mean(x))
## [1] 5.5
##
## > plot(x)
```



Now, did you notice that we got the mean printed and the figure produced?

Calling R from the shell Open a shell, a command window, or however that is called in your operating system, and run R telling it to use a given script file as input. This has the big advantage that you do not need to keep a window with R open until the job finishes. This is great for long running jobs (say, a set of analyses that takes two weeks).

There are several ways of doing it. Probably the preferred way is:

```
R --vanilla < script1.R > scrip1.Rout
```

I tend to use the second one, and then add things like "nohup" before invoking R, move it to the background, and also redirect standard error to the same file used for standard output (i.e., I type `&> script1.Rout` instead of `> script1.Rout`). In Windows, you might need to use `Rscript.exe` or `R.exe` .

Beware, the above are examples of simple invocations. There are many other options.

In section 16 you will have a chance to use and play with a script that reproduces what we did in section 4.

9 Basic R data structures

9.1 Vectors

Vectors are one of the simplest data structures in R. They store a set of objects (all of the same kind), one after the other, in a single dimension. We've seen many:

```
v1 <- c(1, 2, 3)
v2 <- c("a", "b", "cucu")
v3 <- c(1.9, 2.5, 0.6)
```

That, by the way, shows the simplest way of creating a vector in R: use `c` to concatenate a bunch of things.

Many functions (see `?Arithmetic`, `?log`, `?exp`, `?Trig`) operate directly on whole vectors:

```
log(v1)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```
exp(v3)
```

```
## [1]  6.685894 12.182494  1.822119
```

```
2 * v1
```

```
## [1] 2 4 6
```

```
v3/0.7
```

```
## [1] 2.7142857 3.5714286 0.8571429
```

And what functions are there for things like addition, multiplication, exponentiation, division, remainder, etc? As we said, see `?Arithmetic`, `?log`, `?exp`, `?Trig`.

9.1.1 Functions for creating vectors

We can create vectors by concatenating elements. We just saw that. But there are two very handy functions for creating vectors that have some structure: `seq` (from "sequence") and `rep` (from repeat). Examine these examples carefully:

First `seq`, in four different invocations (yes, `:` counts as an invocation of `seq`):

```
seq(from = 1, to = 10)
```

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

```
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
seq(from = 1, to = 10, length.out = 3)
```

```
## [1]  1.0  5.5 10.0
```

```
1:5
```

```
## [1] 1 2 3 4 5
```

Now `rep` in a few common invocations.

```
rep(2, 5)
```

```
## [1] 2 2 2 2 2
```

```
rep(1:3, 2)
```

```
## [1] 1 2 3 1 2 3
```

```
rep(1:3, 2:4)
```

```
## [1] 1 1 2 2 2 3 3 3 3
```

9.2 Creating vectors from other vectors

You can concatenate two vectors:

```
v1 <- 1:4
v2 <- 7:12
(v3 <- c(v1, v2))

##  [1]  1  2  3  4  7  8  9 10 11 12
```

If you use an arithmetic operation on a vector, you get another vector. E.g,

```
v1 <- 2:8
(v2 <- 3 + v1)

## [1]  5  6  7  8  9 10 11
```

And what about this?

```
v1 <- 1:5
v2 <- 11:15
(v3 <- v1 + v2)

## [1] 12 14 16 18 20
```

But what if the two vectors are not the same length? The **recycling rule** applies:

```
v1 <- 1:3
v2 <- 11:12
v1 + v2

## Warning in v1 + v2: longer object length is not a multiple of
shorter object length

## [1] 12 14 14
```

But beware! Look at this

```
v1 <- 1:3
v2 <- 11:16
v1 + v2

## [1] 12 14 16 15 17 19
```

no warnings whatsoever. Which might, or might not, be what you would have expected.

The recycling rule applies also with matrices, etc.

9.3 Logical operations

We can compare the elements of a vector with something, so as to obtain a vector of `TRUE`, `FALSE` elements. And we can combine vectors with value of `TRUE`, `FALSE` using the usual logical operations. Please, look at the help for `Comparison` and `Logic`. These are common in many programming languages (but beware of differences between `||` and `|` and, likewise, `&&` and `&`).

```
?Comparison
?Logic
```

A few examples:

```
v1 <- 1:5
v1 < 3

## [1]  TRUE  TRUE FALSE FALSE FALSE

(v2 <- (v1 < 3))

## [1]  TRUE  TRUE FALSE FALSE FALSE

v11 <- c(1, 1, 3, 5, 4)
v1 == v11

## [1]  TRUE FALSE  TRUE FALSE FALSE

v1 != v11

## [1] FALSE  TRUE FALSE  TRUE  TRUE

!(v1 == v11)

## [1] FALSE  TRUE FALSE  TRUE  TRUE

identical(v1, v11)

## [1] FALSE

v3 <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
!v3

## [1] FALSE  TRUE FALSE  TRUE FALSE

v2 & v3

## [1]  TRUE FALSE FALSE FALSE FALSE

v2 | v3

## [1]  TRUE  TRUE  TRUE FALSE  TRUE

(v1 > 3) & (v11 >= 2)

## [1] FALSE FALSE FALSE  TRUE  TRUE

(v1 > 3) | (v11 >= 2)

## [1] FALSE FALSE  TRUE  TRUE  TRUE

xor(v2, v3)

## [1] FALSE  TRUE  TRUE FALSE  TRUE
```

9.3.1 Logical values as 0, 1

In R (as in many other languages) we can use logical values as if they were numeric: we can treat `TRUE` as 1 and `FALSE` as 0. (Note that we can also treat anything larger than 0 as TRUE). This can be very handy to find out how many elements fulfill a condition.

```
vv <- c(1, 3, 10, 2, 9, 5, 4, 6:8)
```

How many elements are smaller than 5 in `vv`?

```
length(which(vv < 5))

## [1] 4
```

`which` is operating on a logical vector, not on `vv` directly, and `length` is counting the length of the output from `which`.

```
vv < 5

##  [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
## [10] FALSE

vv.2 <- (vv < 5)
vv.2

##  [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
## [10] FALSE

which(vv.2)

## [1] 1 2 4 7

vv.3 <- which(vv.2)
vv.3

## [1] 1 2 4 7

length(vv.3)

## [1] 4
```

Do you know what the output from `which` is?

Alternatively, you can do:

```
length(vv[vv < 5])

## [1] 4
```

Instead of going through `which`, we just directly extract the relevant elements of `vv`, and count how many there are. Implicitly, we are creating a new (temporary) vector, that holds only the elements in `vv` that are smaller than 5, and we are counting the length of that temporary vector.

```
vv[vv < 5]

## [1] 1 3 2 4
```

But the following is often much easier to understand (or to use)

```
sum(vv < 5)

## [1] 4
```

Why did that work? What is `vv < 5` returning?

9.4 Names of elements

The elements of a vector can have names (you should make them distinct). We will see soon why this is very helpful. For now, see this:

```
ages <- c(Juan = 23, Maria = 35, Irene = 12, Ana = 93)
names(ages)

## [1] "Juan"  "Maria" "Irene" "Ana"

ages

##  Juan Maria Irene   Ana
##    23    35    12    93
```

9.5 Accessing (and modifying) vector elements: indexing and subsetting

9.5.1 Vector indexing

There are several ways of getting access to specific elements of a vector:

- By specifying the positions you want (or do not want): giving indexes (or indices).
- By giving the names of the elements.
- By using logical vector (which is really very similar to the third).
- By using any expression that will generate any of the above.

Positions or names will be given in between `[]`.

Specifying positions you want:

```
(w <- 9:18)

##  [1]  9 10 11 12 13 14 15 16 17 18

w[1]

## [1] 9

w[2]
```

```
## [1] 10

w[c(4, 3, 2)]

## [1] 12 11 10
```

```
w[c(1, 3)] ## not the same as

## [1]  9 11

w[c(3, 1)]

## [1] 11  9
```

```
w[1:2]

## [1]  9 10

w[3:6]

## [1] 11 12 13 14

w[seq(1, 8, by = 3)]

## [1]  9 12 15

vv <- seq(1, 8, by = 3)
w[vv]

## [1]  9 12 15
```

Specifying positions you do not want (original vector is NOT modified)

```
w[-1]

## [1] 10 11 12 13 14 15 16 17 18

w[-c(1, 3)] ## of course, the same as following

## [1] 10 12 13 14 15 16 17 18

w[-c(3, 1)]

## [1] 10 12 13 14 15 16 17 18
```

Using names

```
ages <- c(Juan = 23, Maria = 35, Irene = 12, Ana = 93)
ages["Irene"]

## Irene
##    12
```

```
ages[c("Irene", "Juan", "Irene")]

## Irene  Juan Irene
##    12    23    12
```

Using a logical vector . . .

```
ages[c(FALSE, TRUE, TRUE, FALSE)]

## Maria Irene
##    35    12

ages[c(FALSE, TRUE)] ## what is this doing? Avoid these things

## Maria   Ana
##    35    93
```

. . . or something that implicitly is a logical vector

```
## All less than 12
w[w < 12]

## [1]  9 10 11

## same, but more confusing (here, not always)
w[!(w >= 12)]

## [1]  9 10 11

## All less than the median
w[w < median(w)]

## [1]  9 10 11 12 13
```

Of course, if you can access it, you can modify it

```
ages["Irene"] <- 19
ages

##  Juan Maria Irene   Ana
##    23    35    19    93

w[1] <- 9999
w

##  [1] 9999   10   11   12   13   14   15   16   17   18

w[vv] <- 103
w

##  [1] 103  10  11 103  13  14 103  16  17  18
```

But compare this:

```
w[] <- 77
w[] <- 17:55

## Warning in w[] <- 17:55:  number of items to replace is not a multiple
of replacement length

w <- 17:55
```

9.6 Interlude: comparing floats

Comparing very similar numeric values can be tricky: rounding can happen, and some numbers cannot be represented exactly in binary (computer) notation. By default R displays 7 significant digits (`options("digits")`). For example:

```
x  <-  1.999999
x

## [1] 1.999999

x - 2

## [1] -1e-06

x <- 1.9999999999999
x

## [1] 2

x-2

## [1] -9.992007e-14
```

All the digits are still there, in the second case, but they are not shown. Also note that `x-2` is not exactly -1×10^{-13}; this is unavoidable.

Why is the above unavoidable? Because of the way computers represent numbers. We cannot get into details, but see the following example, from the FAQ (question 7.31):

> 7.31 Why doesn't R think these numbers are equal?
>
> The only numbers that can be represented exactly in R's numeric type are integers and fractions whose denominator is a power of 2. Other numbers have to be rounded to (typically) 53 binary digits accuracy. As a result, two floating point numbers will not reliably be equal unless they have been computed by the same algorithm, and not always even then. For example
>
> ```
> R> a <- sqrt(2)
> R> a * a == 2
> [1] FALSE
> R> a * a - 2
> [1] 4.440892e-16
> ```

The take home message: be extremely suspicious whenever you see an equality comparison of two floating-point numbers; that is unlikely to do what you want. If you know what you are doing, take a look at `all.equal` for near equality comparisons of objects.

9.7 Factors

Factors are a special type of vectors. We need them to differentiate between a vector of characters and a vector that represents categorical variables. The vector `char.vec <- c("abc", "de", "fghi")` contains several character strings. Now, suppose we have a study where we record the sex of participants. When we analyze the data we want R to know that this is a categorical variable, where each "label" represents a possible value of the category:

```
Sex.version1 <- factor(c("Female", "Female", "Female",
                         "Male", "Male"))
Sex.version2 <- factor(c("XX", "XX", "XX", "XY", "XY"))
Sex.version3 <- factor(c("Feminine", "Feminine", "Feminine",
                         "Masculine", "Masculine"))
Sex.version4 <- factor(c("fe", "fe", "fe", "ma", "ma"))
```

We want all those codifications of the sex of five subjects to yield the same results of analysis, regardless of what, exactly, the labels say. Each set of labels might have its pros/cons (e.g., the third is probably coding gender, not sex; the last is too cryptic; the second works only for some species; etc). Regardless of the labels, the key thing to notice is that the first three subjects are of the same type, and the last two subjects are of a different type.

Recognizing factors is essential when dealing with variables that look like legitimate numbers:

```
postal.code <- c(28001, 28001, 28016, 28430, 28460)
somey <- c(10, 20, 30, 40, 50)
summary(aov(somey ~ postal.code))

##             Df Sum Sq Mean Sq F value Pr(>F)
## postal.code  1  782.5   782.5   10.79 0.0462 *
## Residuals    3  217.5    72.5
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The above is doing something silly: it is fitting a linear regression, because it is taking postal.code as a legitimate numeric value. But we know that there is no sense in which 28009 and 28016 (two districts in Madrid) are 7 units apart whereas 28430 and 28410 are 20 units apart (two nearby villages north of Madrid), nor do we expect to find linear relationships with (the number of the) postal code itself.

Sometimes, when reading data, a variable will be converted to a factor, but it is really a numeric variable. How to turn it into the original set of numbers?

This does not work:

```
x <- c(34, 89, 1000)
y <- factor(x)
y

## [1] 34   89   1000
## Levels: 34 89 1000

as.numeric(y)
```

```
## [1] 1 2 3

y

## [1] 34   89   1000
## Levels: 34 89 1000
```

Note that values have been re-codified. An easy way to do this is (you should understand what is happening here):

```
as.numeric(as.character(y))

## [1]   34   89 1000
```

```
as.character(y)

## [1] "34"   "89"   "1000"
```

9.7.1 Factors and symbols, colors, etc, in plots

You often see code as follows:

```
plot(y ~ x, col = c("red", "blue")[group])
```

where `group` is a factor. Why does that work? Oh, and then you might see code where we add a legend as

```
legend(1, 2, legend = c("A", "B"),
       col = c("red", "blue")[factor(levels(group))])
```

I won't give the solution here, but these are some hints. Try the following:

```
gr <- c("B", "A", "A", "B", "A")
group <- factor(gr)
c("red", "blue")[gr]

## [1] NA NA NA NA NA

c("red", "blue")[group]

## [1] "blue" "red"  "red"  "blue" "red"

c("red", "blue")[levels(group)]

## [1] NA NA

c("red", "blue")[factor(levels(group))]

## [1] "red"  "blue"
```

9.8 Matrices

Vectors where one-dimensional. Matrices are two-dimensional, and arrays have arbitrary dimensions. We will stick here to matrices. But you have arrays at your disposal, of course. As with vectors, all the elements of a matrix or of an array are of the same type.

9.8.1 Creating matrices from a vector

(The vector, below, is that vector that we create on the fly with `1:10`)

```
matrix(1:10, ncol = 2)
```

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

```
matrix(1:10, nrow = 5)
```

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

```
matrix(1:10, ncol = 2, byrow = TRUE)
```

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

9.8.2 Combining vector to create a matrix: `cbind`, `rbind`

You can glue vectors horizontally or vertically to create a matrix.

```
v1 <- 1:5
v2 <- 11:15
rbind(v1, v2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2   11   12   13   14   15
```

```
cbind(v1, v2)
```

```
##      v1 v2
## [1,]  1 11
## [2,]  2 12
## [3,]  3 13
## [4,]  4 14
## [5,]  5 15
```

And you can do the same with matrices (if they are of the appropriate dimensions, of course)

```
A <- matrix(1:10, nrow = 5)
B <- matrix(11:20, nrow = 5)
cbind(A, B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
rbind(A, B)
```

```
##       [,1] [,2]
##  [1,]    1    6
##  [2,]    2    7
##  [3,]    3    8
##  [4,]    4    9
##  [5,]    5   10
##  [6,]   11   16
##  [7,]   12   17
##  [8,]   13   18
##  [9,]   14   19
## [10,]   15   20
```

9.8.3 Matrix indexing and subsetting

A matrix has two dimensions, but otherwise things are very similar to what happened with vectors. The first dimension are rows, the second are columns. If you specify nothing for that dimension, it is returned completely.

```
A <- matrix(1:15, nrow = 5)
A[1, ] ## first row
```

```
## [1]  1  6 11
```

```
A[, 2] ## second column
```

```
## [1]  6  7  8  9 10
```

```
A[4, 2] ## fourth row, second column
```

```
## [1] 9

A[3, 2] <- 999
A[1, ] <- c(90, 91, 92)
A < 4

##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,]  TRUE FALSE FALSE
## [3,]  TRUE FALSE FALSE
## [4,] FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE
```

Note that `which`, by default, might not do what you expect:

```
which(A == 999)

## [1] 8
```

If you want the indices, ask for them

```
which(A == 999, arr.ind = TRUE)

##      row col
## [1,]   3   2
```

Names work too:

```
B <- A
colnames(B) <- c("A", "E", "I")
rownames(B) <- letters[1:nrow(B)]
B[, "E"]

##   a   b   c   d   e
##  91   7 999   9  10

B["c", ]

##   A   E   I
##   3 999  13
```

Beware: you can use a matrix to index another matrix. This is slightly more advanced, but extremely handy:

```
(m1 <- cbind(c(1, 3), c(2, 1)))

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    1

A[m1]
```

```
## [1] 91  3

## compare with
A[c(1, 3), c(2, 1)]

##      [,1] [,2]
## [1,]   91   90
## [2,]  999    3
```

9.8.4 Operations with matrices

There are many matrix operations available from R (open your matrix algebra book, and try to find them, if you want). And many functions operate directly, by default, on the whole matrix, or on rows/columns of the matrix:

```
sum(B)

## [1] 1366

mean(B)

## [1] 91.06667

colSums(B)

##    A    E    I
##  104 1116  146

rowMeans(B)

##        a        b        c        d        e
##  91.0000   7.0000 338.3333   9.0000  10.0000
```

And, of course, we can subset/select rows and columns using those:

```
B[rowMeans(B) > 9, ]

##    A   E  I
## a 90  91 92
## c  3 999 13
## e  5  10 15
```

9.9 Lists

A list is a more general container, where we can mix pieces of different types. In fact, there need not be any rectangular like structure:

```
listA <- list(a = 1:5, b = letters[1:3])
listA[[1]]

## [1] 1 2 3 4 5
```

```
listA[["a"]]

## [1] 1 2 3 4 5

listA$a

## [1] 1 2 3 4 5
```

so those are three basic ways of getting access to list components. And of course

```
listA[[1]][2]

## [1] 2
```

And compare with this

```
listA[1]

## $a
## [1] 1 2 3 4 5
```

A more complex list, that includes another list inside:

```
(listB <- list(one.vector = 1:10,  hello = "Hola",
               one.matrix = matrix(rnorm(20), ncol = 5),
               another.list =
               list(a = 5,
                    b = factor(c("male",
                      "female", "female")))))

## $one.vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $hello
## [1] "Hola"
##
## $one.matrix
##            [,1]       [,2]        [,3]       [,4]       [,5]
## [1,] -1.9310834 1.1270435  0.37376094  0.2959325  0.2371643
## [2,]  0.4697740 0.3186695 -0.74740119 -1.2423179  0.5038752
## [3,]  0.3932001 0.7871221 -0.06520024  1.2253646 -1.5998379
## [4,] -0.8361996 1.0866436  0.49787715  0.7456812 -0.4963571
##
## $another.list
## $another.list$a
## [1] 5
##
## $another.list$b
## [1] male   female female
## Levels: female male
```

Note that many functions in R return lists.

9.10 Data frames

Above, we ended up with a data frame when we read some data. Do you remember where? How did the object look? A data frame is, really, a list of vectors; all these vectors are of the same length, but different vectors can contain objects of different types. So we have a rectangular structure, where different columns can have objects of different types

```
(AB <- read.table("AnotherDataSet.txt",
                  header = TRUE))
```

```
##    ID Age Sex    Y
## 1  a1  12   M 23.4
## 2  a2  14   M 25.0
## 3  a3  13   F 22.0
## 4  a7  16   F 26.0
## 5 b15  19   F 24.3
```

Data frames are extremely handy for data analysis, and you will see them used extensively there.

We can access elements of data frames as if they were matrices and as if they were lists (where the list elements are the columns):

```
AB[2, 3]
```

```
## [1] M
## Levels: F M
```

```
AB[1, ]
```

```
##   ID Age Sex    Y
## 1 a1  12   M 23.4
```

```
AB["2", ] ## using the rownames
```

```
##   ID Age Sex  Y
## 2 a2  14   M 25
```

```
AB[, "Age"]
```

```
## [1] 12 14 13 16 19
```

```
AB$Age
```

```
## [1] 12 14 13 16 19
```

```
AB[["Age"]]
```

```
## [1] 12 14 13 16 19
```

```
AB[3, 4] <- 97
```

We can go from data frames from matrices, in two different ways (each of which does, therefore, a different thing):

```
data.matrix(AB)
```

```
##      ID Age Sex    Y
## [1,]  1  12   2 23.4
## [2,]  2  14   2 25.0
## [3,]  3  13   1 97.0
## [4,]  4  16   1 26.0
## [5,]  5  19   1 24.3
```

```
as.matrix(AB)
```

```
##      ID    Age  Sex Y
## [1,] "a1"  "12" "M" "23.4"
## [2,] "a2"  "14" "M" "25.0"
## [3,] "a3"  "13" "F" "97.0"
## [4,] "a7"  "16" "F" "26.0"
## [5,] "b15" "19" "F" "24.3"
```

Many matrix operations, in particular `rbind` and `cbind` will also work with data frames.

```
AB2 <- rbind(AB, AB)
```

(yes, the above is somewhat silly, but you get the point).

Of course, creating a data frame is easy:

```
(AC <- data.frame(ID = "a9", Age = 14, Sex = "M", Y = 17))
```

```
##   ID Age Sex  Y
## 1 a9  14   M 17
```

```
rbind(AB, AC)
```

```
##    ID Age Sex    Y
## 1  a1  12   M 23.4
## 2  a2  14   M 25.0
## 3  a3  13   F 97.0
## 4  a7  16   F 26.0
## 5 b15  19   F 24.3
## 6  a9  14   M 17.0
```

Data frames make it particularly easy to add new variables to the data frame.

```
AB2$status <- rep(c("Z", "V"), 5)
```

9.11 Odds and ends

Reshaping data: Sometimes you will need to go from a "wide" to a "long" format in your data, or viceversa (e.g., repeated measures of the same individuals over time). Look at `?reshape`, although I find that the "reshape2" package (and possibly also the "tidyr" package) tends to make life much easier for this task.

Data table: For dealing efficiently with large data sets, and other niceties, the "datatable" package (with its datatables) can really make a difference.

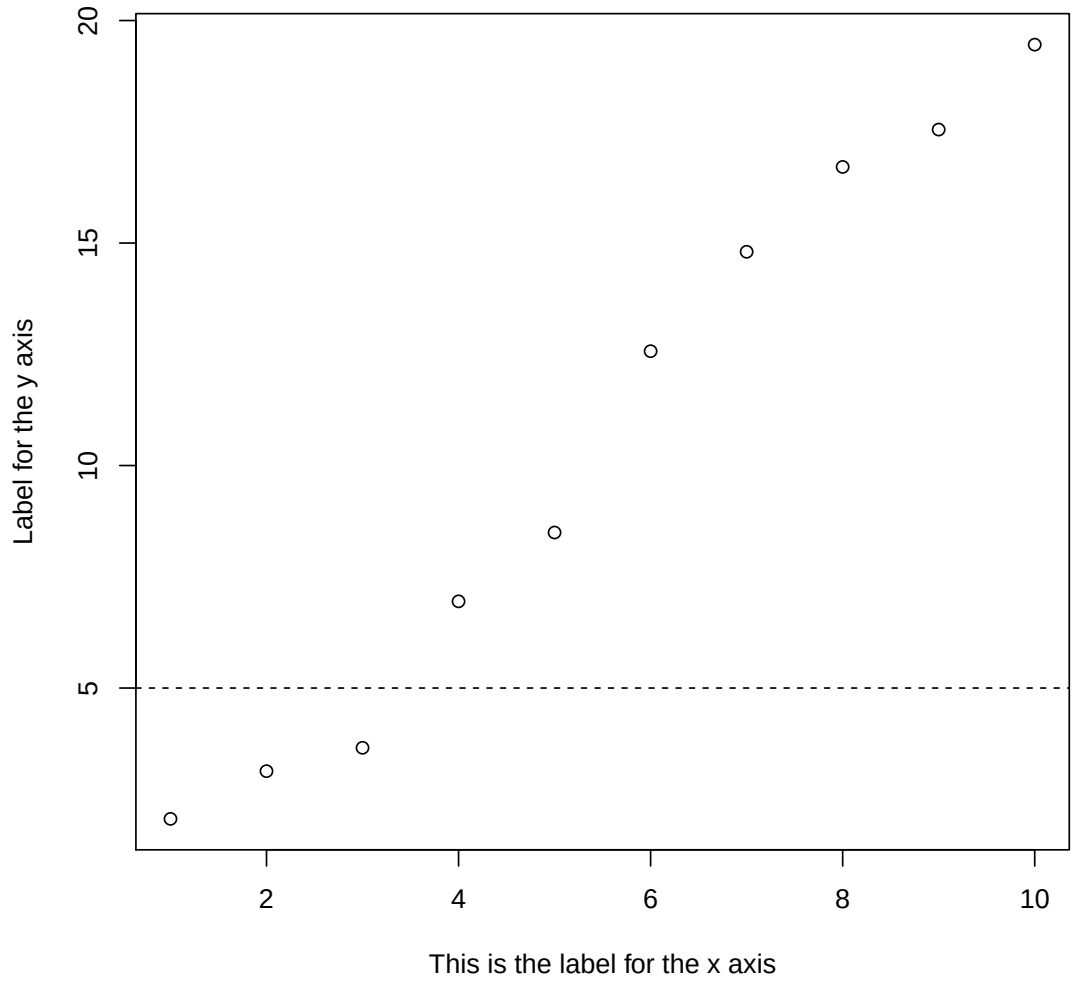
10 Plots

R is often used to produce plots as it can produce a variety of plots and they can be modified at will. In this course we will only scratch the surface. You will have a change to get additional practice from the exercises.

10.1 The very basics

The basic plot function is `plot`. It's help can be slightly misleading and many additional arguments are explained in `par`. A good analogy to begin with is that of a canvas where you go adding elements. Let's look at this simple example:

```
x <- 1:10
y <- 2 * x + rnorm(length(x))
plot(x, y, xlab = "This is the label for the x axis",
     ylab = "Label for the y axis")
## And now, we add a horizontal line:
abline(h = 5, lty = 2)
```



There are many, many other types of plots. We will see some later.

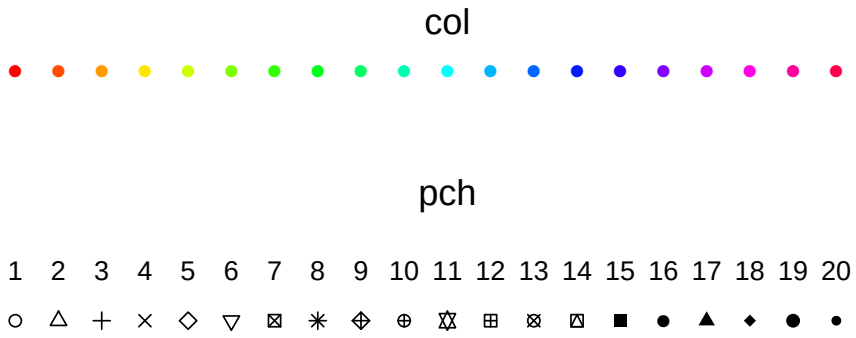


Figure 1: pch and col

10.2 Plots: Can we change colors, line types, point types, etc?

Of course. Look at `?par` and then look for `pch`, `cex`, `lty`, `col`. This code produces two figures (1 and 2) that might help (see also line types, `lty` in Figure 4).

```
plot(c(1, 21), c(1, 2.3),
     type = "n", axes = FALSE, ann = FALSE)
## show pch
points(1:20, rep(1, 20), pch = 1:20)
text(1:20, 1.2, labels = 1:20)
text(11, 1.5, "pch", cex = 1.3)

## show colors for rainbow palette
points(1:20, rep(2, 20), pch = 16, col = rainbow(20))
text(11, 2.2, "col", cex = 1.3)
```

```
plot(c(0.2, 5), c(0.2, 5), type = "n", ann = FALSE, axes = FALSE)
for(i in 1:6) {
    abline(0, i/3, lty = i, lwd = 2)
    text(2, 2 * (i/3), labels = i, pos = 3)
}
```

10.3 Saving plots

Can you save the plots as PDF, png, . . . ? Definitely. From RStudio you have a menu entry in the plot window. For non-interactive work (as when using scripts, section 8), or to make sure you have fixed things such as size, it is better to directly use functions such as `?png`, `?pdf`, etc. Look at the help of those functions. The second approach (explicitly calling, say, `pdf` from my scripts) is what I use.

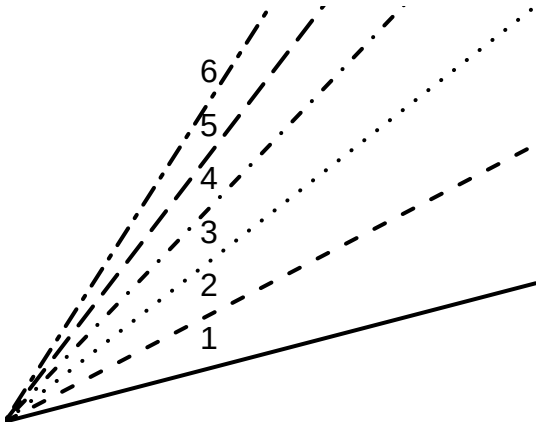


Figure 2: lty for values 1 to 6

10.4 Plots, plots, plots. Many types of plots

We have used a variety of plots. But this is only scratching the surface. Plotting is a big thing in R. We have used a few, but there are many more. In fact, there are several approaches or systems for plots in R. We will use here the basic one (the one in base R), but notice that **ggplot2** is a very popular one, that produces plots many people find nicer. If you are interested, google for "ggplot2". There are two books about it ("ggplot2: elegant graphics for data analysis", by Wickham, its creator, and "R graphics cookbook", by Chang, who has been heavily involved in ggplot2). The second one has a web page with many recipes: `http://www.cookbook-r.com/Graphs/`. Another popular package is **lattice**, also with its own book. And there are comparison pages of lattice and ggplot2. Another issue we haven't touched is choice of colors, which is not a trivial thing (and there are a variety of color palettes in R; search for `?palette`, for example). There are also ways to identify points, to use 3D plots, to have dynamic plots that allow rotation, etc, etc.

The following could give you an idea of some of the options:

```
demo(graphics)
example(graphics)
example(persp)
demo(persp)
```

And, if you have installed package "ggplot2" you can also do

```
example(qplot)
```

(though the above gives just a very, very limited view of the range of options with ggplot2).

11 Three examples with data manipulation and plots

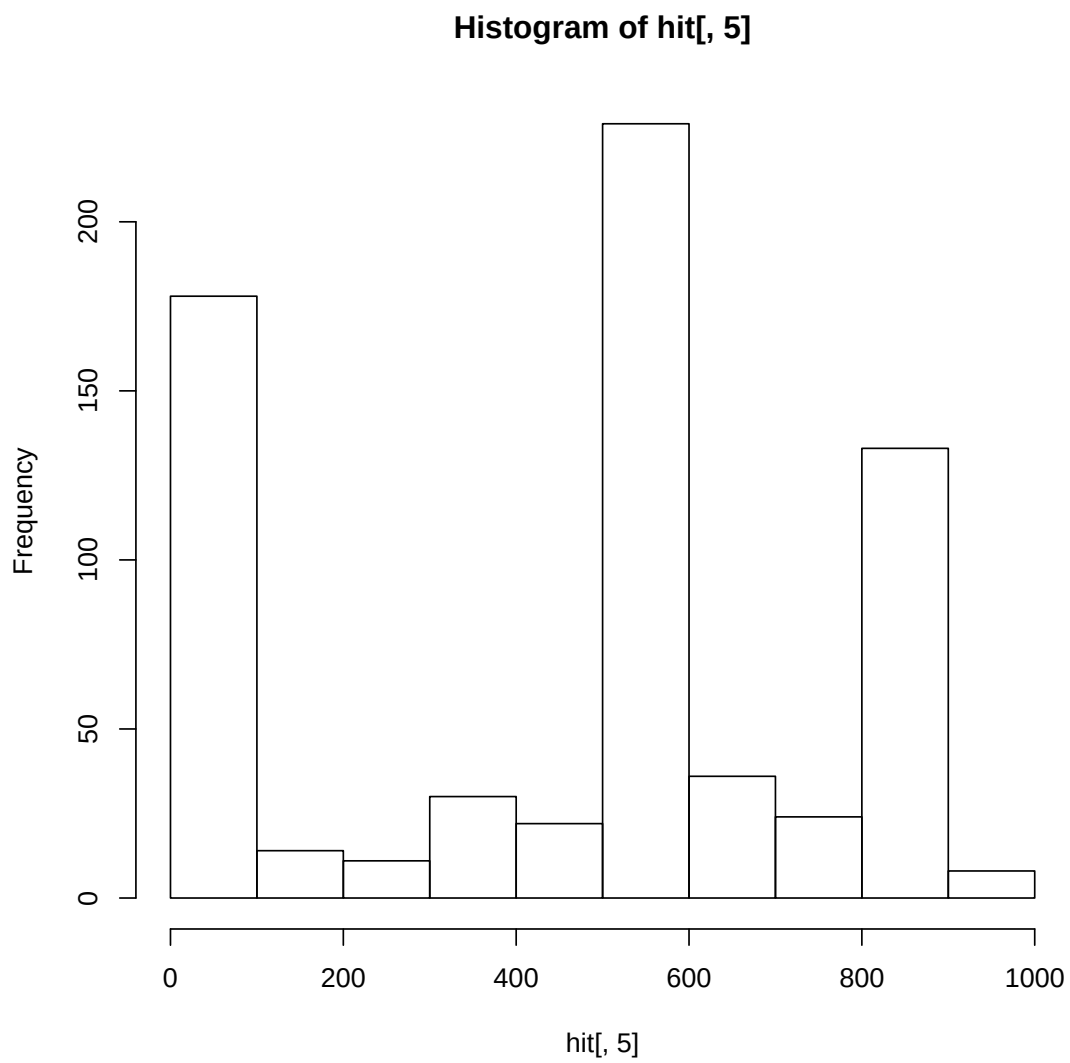
11.1 Example of reading data and plotting: Plotting the results from BLAST

This is a quick example of something that is not really just a mere toy example. We will use output from alignment statistics from BLAST. Let's take a quick look at a couple of things:

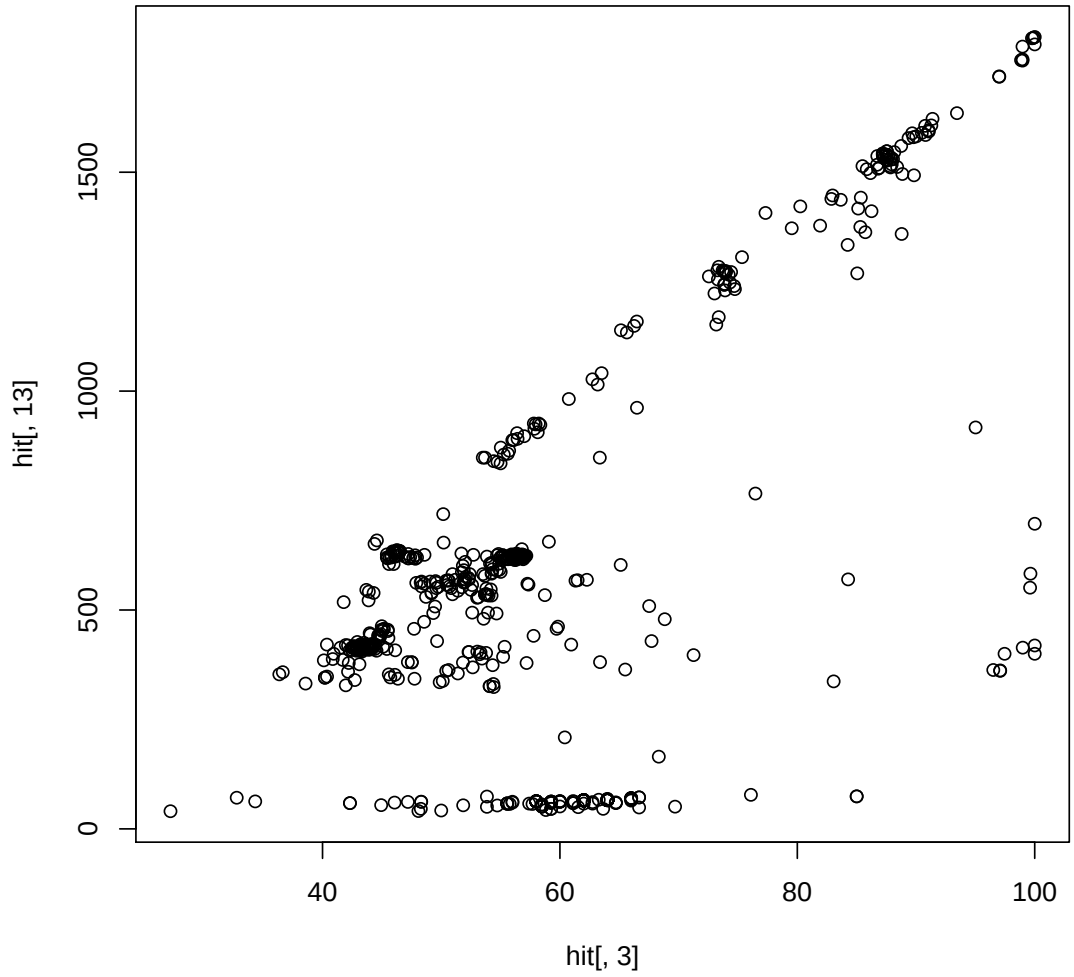
- The relationship between percentage identity and score.
- The distribution of the alignment length.

We will read the data and then the a couple of plots. Remember that the data are in "hit-table-500-text.txt" (don't worry, we will cover reading data again later, in section 7).

```
hit <- read.table("hit-table-500-text.txt")
## We know, from the header of the file, that
## alignment length is the fifth column,
## score is the 13th and percent identity the 3rd
hist(hit[, 5]) ## the histogram
```



```
plot(hit[, 13] ~ hit[, 3]) ## the scatterplot
```



But that can be easily improved

```
par(mfrow = c(1, 2)) ## two figures side by side
hist(hit[, 5], breaks = 50, xlab = "", main = "Alignment length")
plot(hit[, 13] ~ hit[, 3], xlab = "Percent. identity",
     ylab = "Bit score")
```

These are five lines of code. We will now deal with a few things in a little more detail.

And if you want to play around, you can find the best fitting plane of score on alignment length and identity[2] , and move it at will!. You will need to install two packages (section 5.4): `car` and `rgl`.

```
library(car)
scatter3d(hit[, 13] ~ hit[, 3] + hit[, 5], xlab = "Ident",
          zlab = "Length", ylab = "Score")
```

[2]This is not a great, or even a decent, model of the relationship, which we can see, for instance, from the pattern of the deviations of the points from the plane. But the key for now is the ease of plotting.

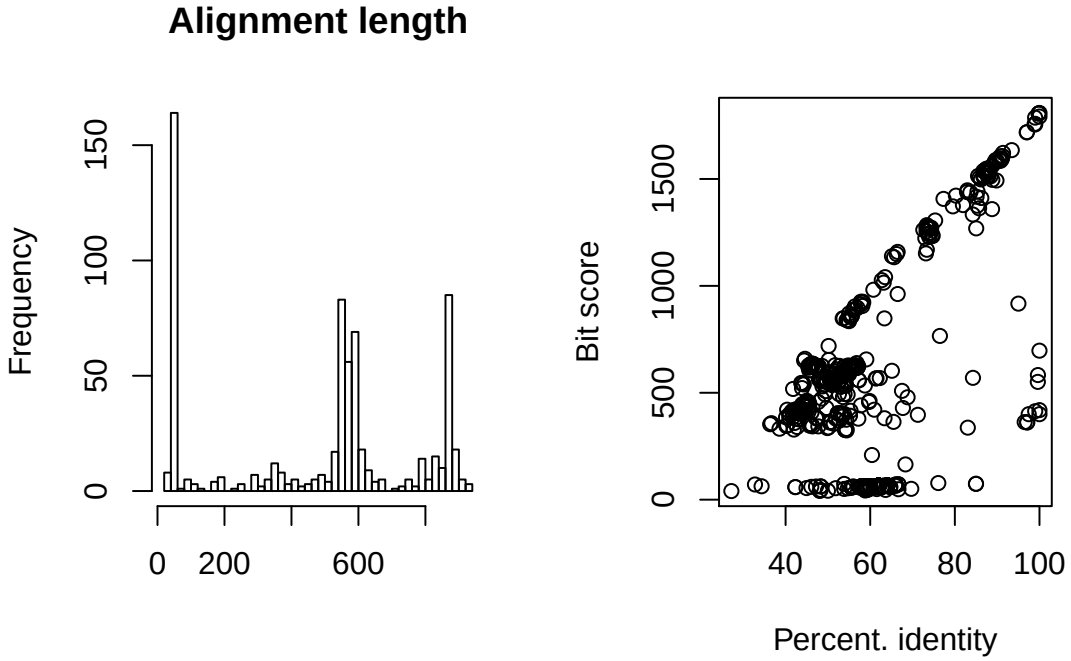


Figure 3: A quick look at the alignment results

By the way, some questions/answers arise from the figures. For instance: why does the distribution of alignment lengths seems to have three distinct modes? Or how can you explain the plot of score vs. percent identity? Or what does alignment length add to understanding the relationship? We could, in fact, fit (in one line) a linear regression to the relationship between Score and the other variables to formally explore what is going on[3]. Or ....

[3]Not that this would be, in this case, all that needed: from what we know about BLAST we can tell a lot about the expected relationship between Score and the other variables

11.2 More plots and a regression example: Metabolic rate and body mass

You are interested in the relationship between metabolic rate and body mass in birds. We will use a subset of data from the AnAge data set (Animal Ageing and Longevity Database) (accessed on 2014-08-19) from http://genomics.senescence.info/species/. This file contains longevity, metabolic rate, body mass, and a variety of other life history variables. The data I provide you are a small subset that includes only some birds and reptiles.

What I provide you is that is already stored as an R object (so I already took care of the `read.table` business for you, but if you are curious and want the data, skip to 11.2.5).

First, let us **load** the data set, the RData file:

```
load("anage.RData")
ls() ## a new anage object is present
```

```
##  [1] "A"             "AB"            "AB2"
##  [4] "AC"            "ages"          "anage"
##  [7] "B"             "gr"            "group"
## [10] "hit"           "i"             "listA"
## [13] "listB"         "m1"            "postal.code"
## [16] "Sex.version1" "Sex.version2" "Sex.version3"
## [19] "Sex.version4" "somey"         "v1"
## [22] "v11"           "v2"            "v3"
## [25] "vv"            "vv.2"          "vv.3"
## [28] "w"             "x"             "y"
```

Now, look at the data

```
str(anage)
```

```
## 'data.frame': 1726 obs. of  9 variables:
##  $ Class                  : Factor w/ 2 levels "Aves","Reptilia": 1 1 1 1 1
##  $ Order                  : Factor w/ 37 levels "Accipitriformes",..: 1 1 1
##  $ Family                 : Factor w/ 162 levels "Acanthizidae",..: 2 2 2 2
##  $ Genus                  : Factor w/ 784 levels "Acanthocercus",..: 3 3 3
##  $ Species                : Factor w/ 1412 levels "aalge","abacura",..: 331
##  $ Metabolic.rate..W.     : num  NA NA 0.952 NA NA ...
##  $ Body.mass..g.          : num  NA NA 135 NA NA NA NA 3000 NA NA ...
##  $ Temperature..K.        : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ Maximum.longevity..yrs.: num  20.3 22 20.2 19.9 39 44.5 40 48 20 56 ...
```

```
head(anage)
```

```
##   Class           Order       Family     Genus   Species
## 1  Aves Accipitriformes Accipitridae Accipiter  cooperii
## 2  Aves Accipitriformes Accipitridae Accipiter  gentilis
## 3  Aves Accipitriformes Accipitridae Accipiter     nisus
## 4  Aves Accipitriformes Accipitridae Accipiter  striatus
## 5  Aves Accipitriformes Accipitridae  Aegypius  monachus
## 6  Aves Accipitriformes Accipitridae    Aquila adalberti
##   Metabolic.rate..W. Body.mass..g. Temperature..K.
## 1                 NA            NA              NA
```

```
## 2                  NA            NA               NA
## 3              0.9516           135               NA
## 4                  NA            NA               NA
## 5                  NA            NA               NA
## 6                  NA            NA               NA
##   Maximum.longevity..yrs.
## 1                    20.3
## 2                    22.0
## 3                    20.2
## 4                    19.9
## 5                    39.0
## 6                    44.5
```

```
summary(anage)
```

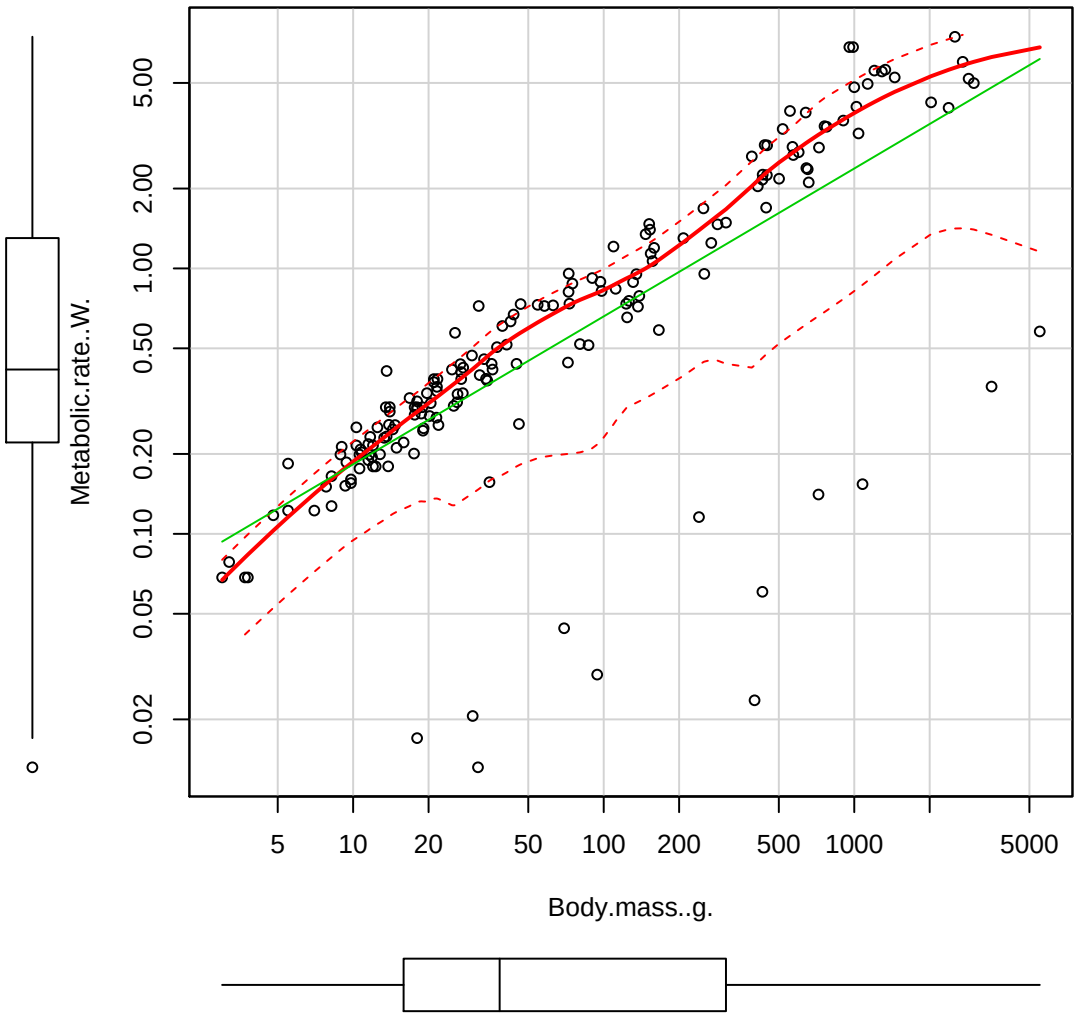
```
##       Class                   Order              Family
##  Aves    :1186   Passeriformes  :417   Psittacidae : 169
##  Reptilia: 540   Squamata       :397   Colubridae  :  86
##                  Psittaciformes :169   Anatidae    :  67
##                  Charadriiformes:132   Viperidae   :  66
##                  Testudines     :121   Accipitridae:  54
##                  Anseriformes   : 67   Emberizidae :  50
##                  (Other)        :423   (Other)     :1234
##        Genus           Species     Metabolic.rate..W.
##  Dendroica:  20   americana:  11   Min.   :0.0132
##  Crotalus :  19   alba     :   9   1st Qu.:0.2233
##  Amazona  :  17   mexicanus:   8   Median :0.4160
##  Falco    :  15   minor    :   8   Mean   :1.1664
##  Varanus  :  15   elegans  :   7   3rd Qu.:1.2880
##  Anas     :  14   australis:   6   Max.   :7.4620
##  (Other)  :1626   (Other)  :1677   NA's   :1548
##  Body.mass..g.     Temperature..K. Maximum.longevity..yrs.
##  Min.   :   3.00   Min.   :289.5   Min.   :  0.40
##  1st Qu.:  16.12   1st Qu.:295.4   1st Qu.: 10.80
##  Median :  38.45   Median :297.1   Median : 16.10
##  Mean   : 326.61   Mean   :300.4   Mean   : 19.55
##  3rd Qu.: 301.88   3rd Qu.:304.9   3rd Qu.: 24.50
##  Max.   :5500.00   Max.   :314.1   Max.   :177.00
##  NA's   :1548      NA's   :1711    NA's   :97
```

You should be able to tell what is going on with the data. And notice those many "NA".

11.2.1 A plot with changed scale

I want to plot metabolic rate vs. body mass. But I know (from theory and previous experience) that I probably want to use the log. And I want to get a quick idea of the spread of the points, etc. So I will use a function from the package `car` (for this to work, thus, you need to install that package).

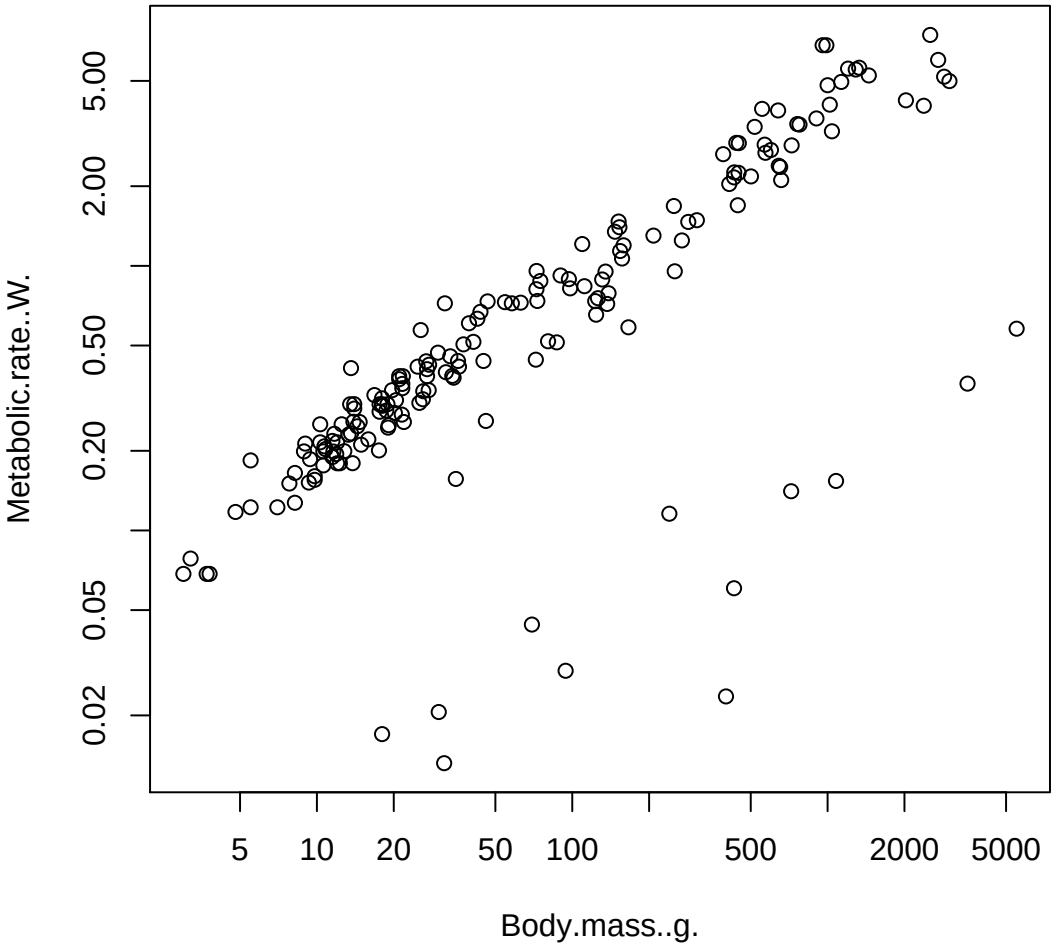
```r
library(car)  ## make the car package available; this is NOT
              ## installing it. It is making it available
scatterplot(Metabolic.rate..W. ~ Body.mass..g., log="xy",
            data = anage)
```



Look at the axes, etc.

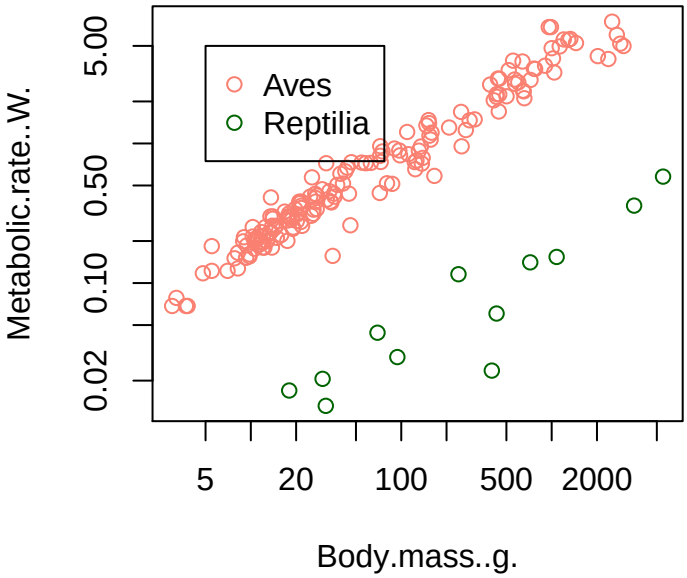
You could do something very similar with the basic plot function, but it would not add the extra lines, boxplots, etc.

```r
plot(Metabolic.rate..W. ~ Body.mass..g., log="xy", data = anage)
```



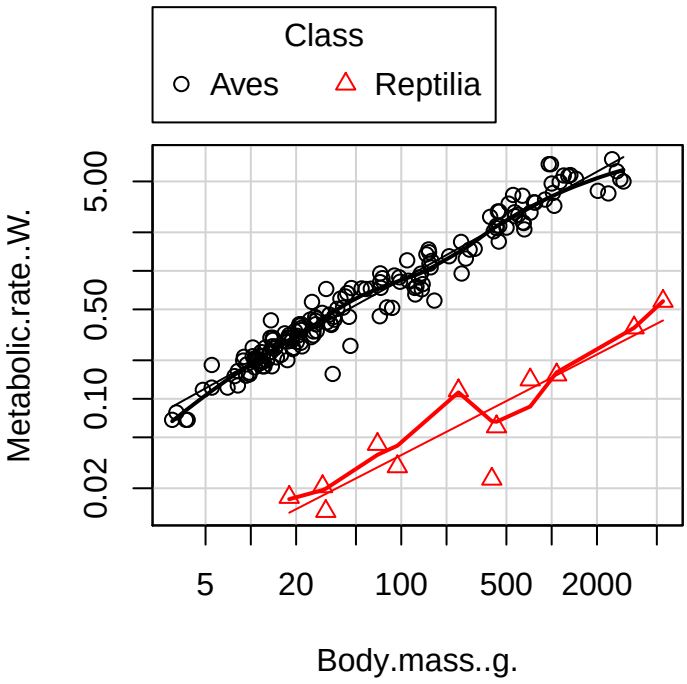
Recall we had both birds (Aves) and reptiles, in variable "Class". Let us use different plotting colors for each, and let us add a legend. Notice how the different colors by Class is done in the call to plot:

```
plot(Metabolic.rate..W. ~ Body.mass..g., log="xy",
     col = c("salmon", "darkgreen")[Class], data = anage)
legend(5, 5, legend = levels(anage$Class),
       col = c("salmon", "darkgreen"),
       pch = 1)
```



Would the above work with scatterplot? Nope. You need to use a slightly different argument for the call, and you get automatic legend, etc:

```
scatterplot(Metabolic.rate..W. ~ Body.mass..g.|Class, log="xy",
            data = anage)
```



And how did I know? Hummm . . . : I tried, failed, and then looked at the help for `scatterplot`.

11.2.2 Transforming variables

It is actually simpler, for the regression later, to log-transform the variables and add them to the data set:

```
anage$logMetab <- log(anage$Metabolic.rate..W.)
anage$logBodyMass <- log(anage$Body.mass..g.)
```

We just created two variables, and added them to `anage`.

11.2.3 Only the birds! Selecting specific cases

Eh, but I want only the birds. Let's select only the birds using function `subset`

```
birds <- subset(anage, Class == "Aves")
```

Alternatively, we could have done

```
birds <- anage[anage$Class == "Aves", ]
```

11.2.4 The regression only for the birds

```
(lm1 <- lm(logMetab ~ logBodyMass, data = birds))

##
## Call:
## lm(formula = logMetab ~ logBodyMass, data = birds)
##
## Coefficients:
## (Intercept)  logBodyMass
##     -3.1595       0.6504
```

A little bit more detail:

```
summary(lm1)

##
## Call:
## lm(formula = logMetab ~ logBodyMass, data = birds)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.00686 -0.14349  0.01545  0.16584  0.61638
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -3.15949    0.04895  -64.55   <2e-16 ***
## logBodyMass  0.65037    0.01095   59.38   <2e-16 ***
## ---
## Signif. codes:
```

```
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2452 on 164 degrees of freedom
##   (1020 observations deleted due to missingness)
## Multiple R-squared:  0.9556,Adjusted R-squared:  0.9553
## F-statistic:  3527 on 1 and 164 DF,  p-value: < 2.2e-16
```

We will explain what the output from that table is, if you do not remember your statistics classes.

Note that, as with the t-test, we can access the elements of `lm1`:

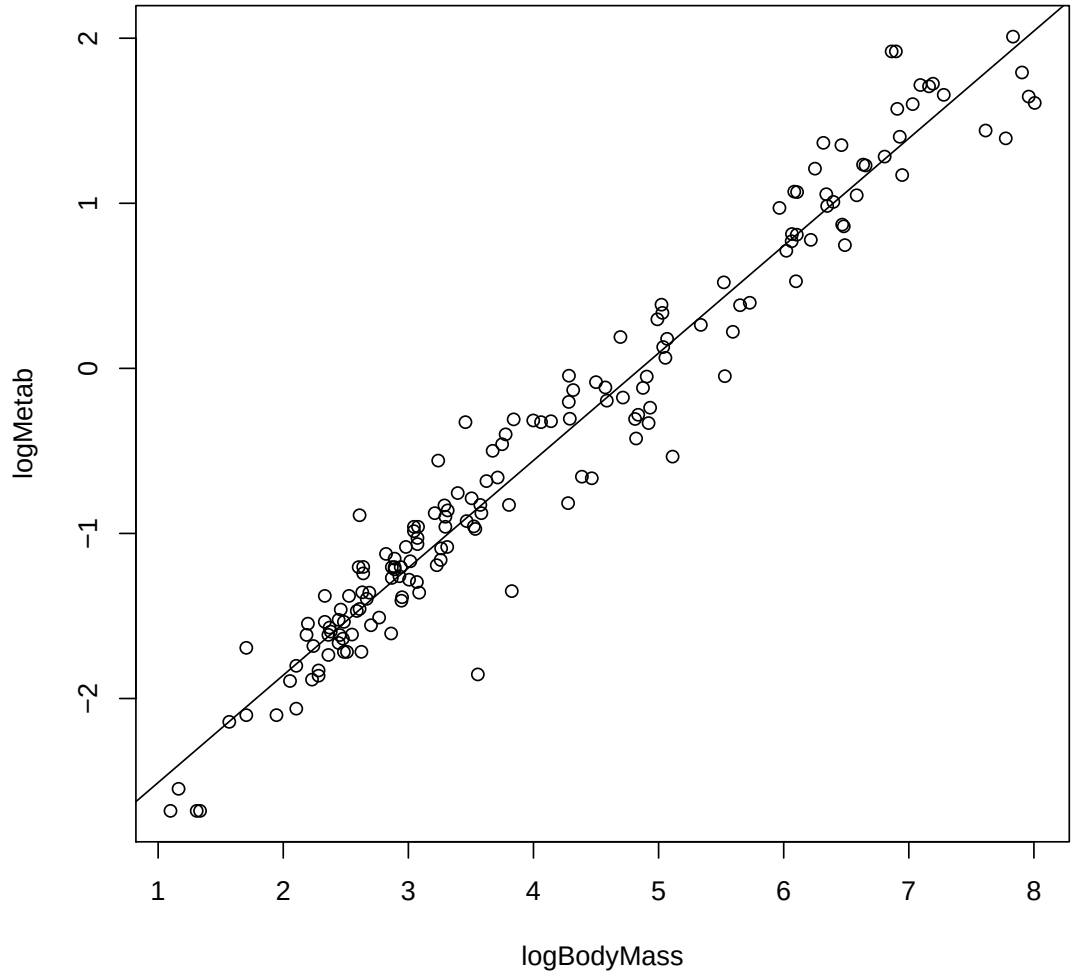
```
names(lm1)

##  [1] "coefficients"  "residuals"     "effects"
##  [4] "rank"          "fitted.values" "assign"
##  [7] "qr"            "df.residual"   "na.action"
## [10] "xlevels"       "call"          "terms"
## [13] "model"

lm1$coefficients

## (Intercept) logBodyMass
##  -3.1594877   0.6503675
```

but, in general, it often makes more sense (when they are available) to use the specific accessor functions:

```
coef(lm1)

## (Intercept) logBodyMass
##  -3.1594877   0.6503675
```

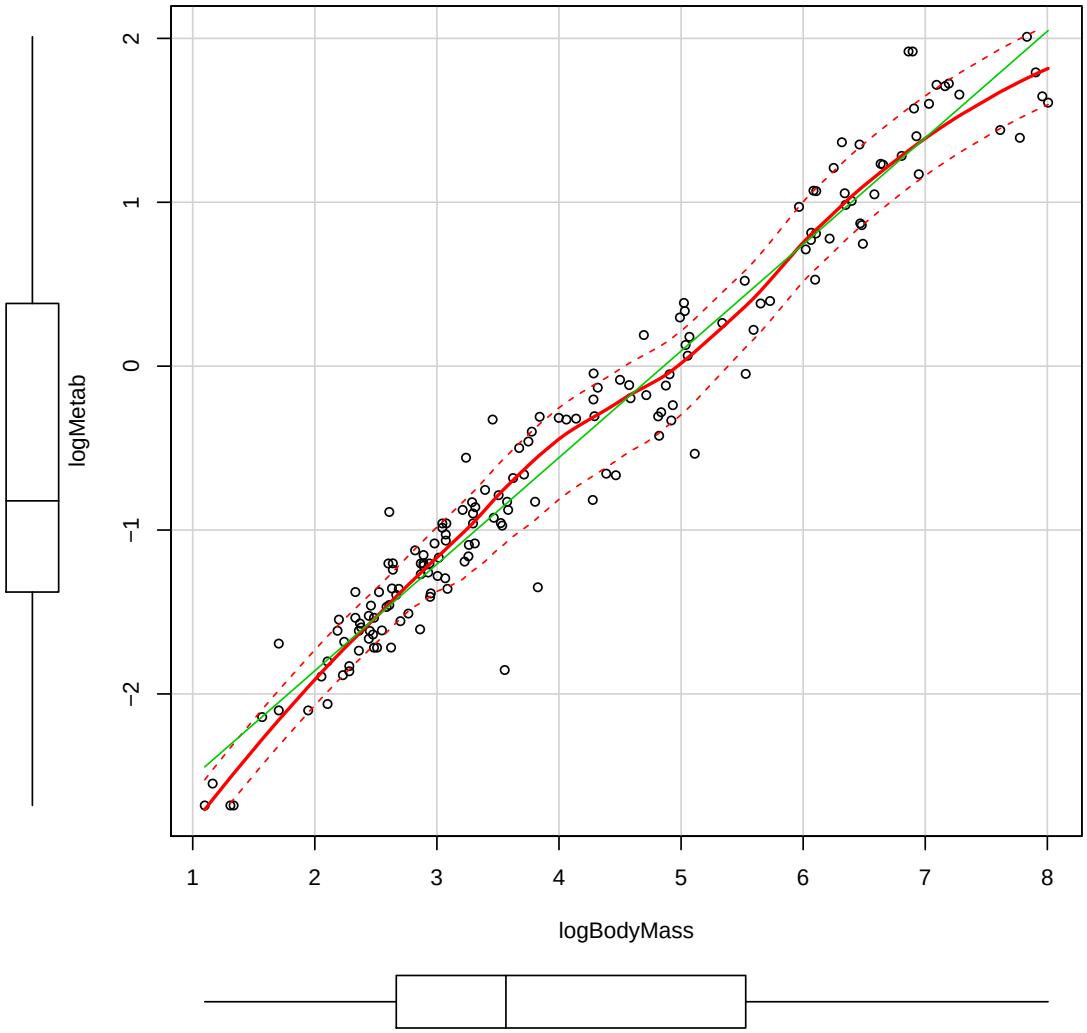
Now, do a plot and add the fitted line:

```
plot(logMetab ~ logBodyMass, data = birds)
abline(lm1)
```



Note that the above is actually the same as doing (except the one below adds more stuff):

```
scatterplot(logMetab ~ logBodyMass, data = birds)
```



but the procedure above is a general one that will work even if John Fox had not written `scatterplot`.

11.2.5 How I read and saved the data?

This is all I did (the `save` thing will become clearer below —section 7.5):

```
anage <-  read.table("AnAge_birds_reptiles.txt",
                     header=TRUE, na.strings="NA",
                     strip.white=TRUE)
save(file = "anage.RData", anage)
```

The file "AnAge_birds_reptiles.txt" is a subset of the original data I downloaded.

11.3 Simulations and plots. A simple hypothesis test: the *t*-test

Suppose we have 50 patients, 30 of which have colon cancer and 20 of which have lung cancer. And we have expression data for one gen (say GenA). We would like to know whether the (mean) expression of GenA is the same or not between the two groups. A *t*-test is well suited for this case. Here, we will use simulated data.

Simulated data? Generating simulated data is extremely useful for testing many procedures, emulating specific processes, etc.

11.3.1 Generating random numbers

R offers (pseudo)random number generators from which we can obtain random variates for many different distributions. For instance, do

```
help(rnorm)
help(runif)
help(rpois)
```

(i.e., look at the help for those functions).

In fact, those numbers are generated using an algorithm. This comes from the Wikipedia (http://en.wikipedia.org/wiki/Pseudorandom_number_generator):

> A pseudorandom number generator (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's state.

And

> A PRNG can be started from an arbitrary starting state, using a seed state. It will always produce the same sequence thereafter when initialized with that state.

Now, since we all have different machines, the actual outcome of doing, say

```
rnorm(5)
```

```
## [1] -0.44176403 -0.01940106  0.41219722 -0.83721002
## [5] -0.91237251
```

is likely to differ.

What can we do to get the exact same random numbers? The quote from the Wikipedia just told us: we will set the seed of the random number generator to force the generator to produce the same sequence of random numbers on all computers:

```
set.seed(2)
```

(seting the seed to 2 has no particular meaning; we could have used another integer; what matters is that we all use the same seed). So lets obtain 50 independent random samples from a normal distribution and create a vector for the identifiers (the "labels") of the subjects.

11.3.2 The t-tests and some plots

```
GeneA <- rnorm(50)
round(GeneA, 3)
```

```
##  [1] -0.897  0.185  1.588 -1.130 -0.080  0.132  0.708 -0.240
##  [9]  1.984 -0.139  0.418  0.982 -0.393 -1.040  1.782 -2.311
## [17]  0.879  0.036  1.013  0.432  2.091 -1.200  1.590  1.955
## [25]  0.005 -2.452  0.477 -0.597  0.792  0.290  0.739  0.319
## [33]  1.076 -0.284 -0.777 -0.596 -1.726 -0.903 -0.559 -0.247
## [41] -0.384 -1.959 -0.842  1.904  0.622  1.991 -0.305 -0.091
## [49] -0.184 -1.199
```

```
(Type <- factor(c(rep("Colon", 30), rep("Lung", 20))))
```

```
##  [1] Colon Colon Colon Colon Colon Colon Colon Colon Colon
## [10] Colon Colon Colon Colon Colon Colon Colon Colon Colon
## [19] Colon Colon Colon Colon Colon Colon Colon Colon Colon
## [28] Colon Colon Colon Lung  Lung  Lung  Lung  Lung  Lung
## [37] Lung  Lung  Lung  Lung  Lung  Lung  Lung  Lung  Lung
## [46] Lung  Lung  Lung  Lung  Lung
## Levels: Colon Lung
```

Let's do a t-test:

```
t.test(GeneA ~ Type)
```

```
##
##  Welch Two Sample t-test
##
## data:  GeneA by Type
## t = 1.2592, df = 44.05, p-value = 0.2146
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.2394925  1.0371620
## sample estimates:
## mean in group Colon  mean in group Lung
##           0.2286718          -0.1701629
```

As should be the case (we know it, because we generated the data) the means of the two groups are very similar, and there are no significant differences between the groups.

Let us now generate data where there really are differences between the groups, and repeat the test.

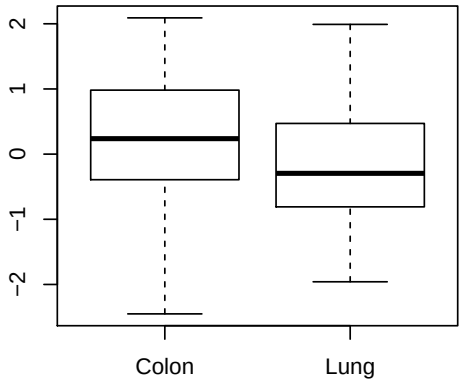
```
(GeneB <- c(rep(-1, 30), rep(2, 20)) + rnorm(50))
```

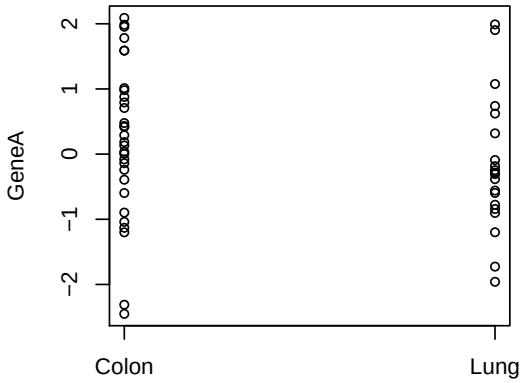
```
##  [1] -1.83828715  1.06630136 -1.56224705  0.27571551
##  [5] -2.04757263 -2.96587824 -1.32297109 -0.06413747
##  [9]  0.13922980  0.67161877 -2.78824221  1.03124252
## [13] -1.70314433 -0.84183524 -0.49376520 -1.81999511
## [17] -2.99884699 -1.47929259 -0.91582010 -1.89548661
## [21] -1.92127567 -0.66955050 -1.14166081 -0.56515224
## [25] -1.05372263 -1.90711038  0.30351223 -0.22821022
```

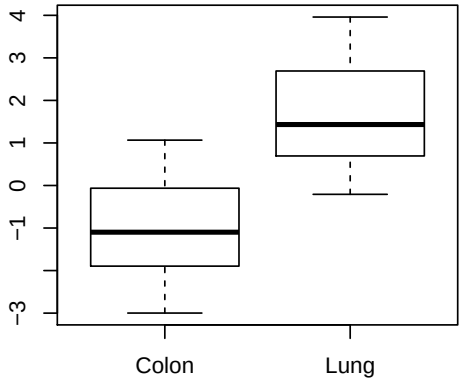
```
## [29]  0.05252560 -2.41003834  2.99598459  0.30423510
## [33]  1.46662786  0.62773055 -0.20791978  3.82212252
## [37]  1.34660659  1.71531878  1.61305040  2.38669497
## [41]  3.60039085  3.68115496  0.81639361  0.64154275
## [45]  0.48732921  0.74689510  3.95935708  2.00764587
## [49]  1.15738480  1.39883989

t.test(GeneB ~ Type)

##
##  Welch Two Sample t-test
##
## data:  GeneB by Type
## t = -7.8114, df = 37.764, p-value = 2.106e-09
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.481516 -2.048162
## sample estimates:
## mean in group Colon  mean in group Lung
##           -1.036470            1.728369
```

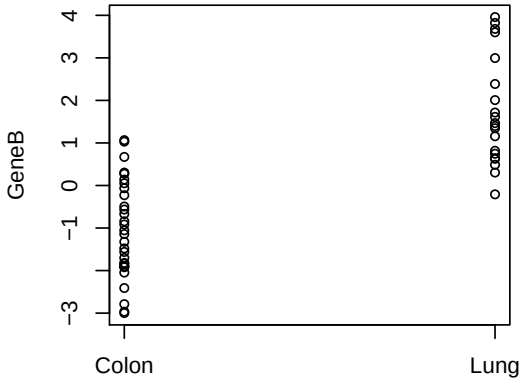
Now we find a large and significant difference between the two groups.
How about some plots?

```
par(mfrow = c(2, 2)) ## a 2-by-2 layout
boxplot(GeneA ~ Type)
stripchart(GeneA ~ Type, vertical = TRUE, pch = 1)
boxplot(GeneB ~ Type)
stripchart(GeneB ~ Type, vertical = TRUE, pch = 1)
```


2
1
0
−1
−2
Colon
Lung


GeneA
2
1
0
−1
−2
Colon
Lung


4
3
2
1
0
−1
−3
Colon
Lung


GeneB
4
3
2
1
0
−1
−3
Colon
Lung

12 Tables

Tabulating data is a very common operation. Let's use again the data frame we played with above:

```
table(AB2$Sex)

##
## F M
## 6 4

with(AB2, table(Sex, status)) ## note "with"

##    status
## Sex V Z
##   F 3 3
##   M 2 2

xtabs( ~ Sex + status, data = AB2)

##    status
## Sex V Z
##   F 3 3
##   M 2 2
```

Now, look at this:

```
(freqs <- as.data.frame(xtabs( ~ Sex + status, data = AB2)))

##   Sex status Freq
## 1   F      V    3
## 2   M      V    2
## 3   F      Z    3
## 4   M      Z    2
```

Did you see how easy it was?

Can we tabulate AB2 completely? (I am not showing the output, but please do it)

```
table(AB)
```

here it does not make much sense, since some variables seem to be continuous. But lets create another data set:

```
AC <- AB2[, c("Sex", "status")]
table(AC)

##    status
## Sex V Z
##   F 3 3
##   M 2 2
```

Of course, you could do

```
table(AB2[, c("Sex", "status")])

##     status
## Sex V Z
##   F 3 3
##   M 2 2
```

So if all you have are categorical variables, the above works. And all this applies to more than two variables, of course.

```
x <-  data.frame(a = c(1,2,2,1,2,2,1),
                 b = c(1,2,2,1,1,2,1),
                 c = c(1,1,2,1,2,2,1))

## tabulate: so create a data frame with a "Freq" column
dfx <- as.data.frame(table(x))

## Recover the table (you will not do this often?)
xtabs(Freq ~ a + b + c, data = dfx)

## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2

## of course, this is the same
xtabs(~ a + b + c, data = x)

## , , c = 1
##
##    b
## a   1 2
##   1 3 0
##   2 0 1
##
## , , c = 2
##
##    b
## a   1 2
##   1 0 0
##   2 1 2
```

12.1 Tables, II

Tabulating data is a very common operation. The following are slightly more advanced exercises. Run them, and see if you understand them, but skip this is if you do not care for now.

```
y <- x
y$d <- y$a + 1

## table of b, c, d, counts are the a
## (this is like example above with Freq on the left)
xtabs(a ~ ., data = y)

## , , d = 2
##
##    c
## b   1 2
##   1 3 0
##   2 0 0
##
## , , d = 3
##
##    c
## b   1 2
##   1 0 2
##   2 2 4

## table of c, d; counts are the a and the b
## (instead of just "a" or Freq)
## "columns are intrepreted as the levels of a variable"
xtabs(cbind(a, b) ~ ., data = y)

## , ,  = a
##
##    d
## c   2 3
##   1 3 2
##   2 0 6
##
## , ,  = b
##
##    d
## c   2 3
##   1 3 2
##   2 0 5

## the usual table of all
xtabs(~ c + d + a + b, data = y)

## , , a = 1, b = 1
##
##    d
## c   2 3
```

```
##   1 3 0
##   2 0 0
##
## , , a = 2, b = 1
##
##    d
## c   2 3
##   1 0 0
##   2 0 1
##
## , , a = 1, b = 2
##
##    d
## c   2 3
##   1 0 0
##   2 0 0
##
## , , a = 2, b = 2
##
##    d
## c   2 3
##   1 0 1
##   2 0 2

## maybe easier to see?
ftable(y)

##       d 2 3
## a b c
## 1 1 1   3 0
##     2   0 0
##   2 1   0 0
##     2   0 0
## 2 1 1   0 0
##     2   0 1
##   2 1   0 1
##     2   0 2

## One more twist:
## "columns are intrepreted as the levels of a variable"
y$e <- y$a + y$c + 9
xtabs(cbind(a, b, e) ~ ., data = y)

## , ,  = a
##
##    d
## c    2  3
##   1  3  2
##   2  0  6
##
## , ,  = b
```

```
##
##    d
## c    2  3
##   1  3  2
##   2  0  5
##
## , ,  = e
##
##    d
## c    2  3
##   1 33 12
##   2  0 39
```

13 R programming

R is a programming language. Comprehensive coverage is given in the books by Chambers "Software for Data Analysis: Programming with R", Matloff's "The art of R programming", Wickham's "Advanced R", and Venables and Ripley "S Programming" (see details in http://www.r-project.org/doc/bib/R-books.html). Chapters 9 and 10 of "An Introduction to R" (http://cran.r-project.org/doc/manuals/R-intro.html) provide a fast but thorough introduction of the main concepts of programming in R.

It is important to emphasize that the ease of combining programming with "canned" statistical procedures gives R a definite advantage over other languages in Bioinformatics (and explains its fast adoption).

13.1 Flow control

R contains usual constructions for flow control and conditional execution: `if`, `ifelse`, `for`, `while`, `repeat`, `switch`, `break`. Before continuing, please note that `for` loops are rarely the best solution: the `apply` family of functions —see below— is often a better approach.

A few examples follow. Make sure you understand them.

`for` iterates over sets. They need not be numbers:

```
names.of.friends <- c("Ana", "Rebeca", "Marta",
                      "Quique", "Virgilio")
for(friend in names.of.friends) {
  cat("\n I should call", friend, "\n")
}

##
##  I should call Ana
##
##  I should call Rebeca
##
##  I should call Marta
##
##  I should call Quique
##
##  I should call Virgilio
```

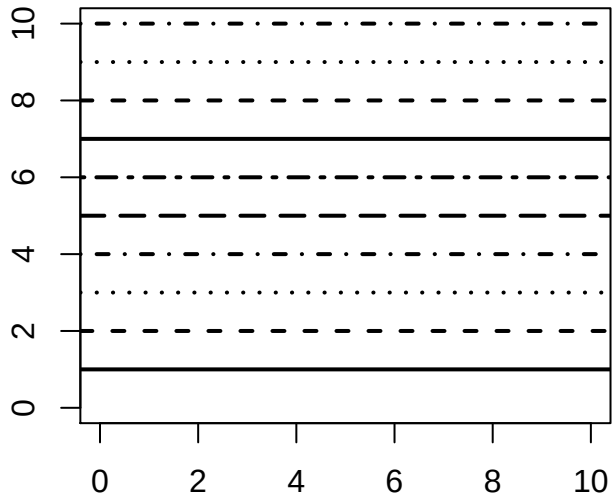


Figure 4: Those line types (lty) again

but they can be numbers:

```
plot(c(0, 10), c(0, 10), xlab ="", ylab ="", type = "n")
for(i in 1:10)
  abline(h = i, lty = i, lwd = 2)
```

(note that in the example above I could get away without using braces —"`{}`"— after the `for` because the complete expression is only one command, `abline`).

`while` keeps repeating a set of instructions until a condition is fulfilled. In this example, the condition is actually two conditions. Make sure you understand what is happening.

```
x <- y <- 0
iteration <- 1
while( (x < 10) & (y < 2)) {
  cat(" ... iteration", iteration, "\n")
  x <- x + runif(1)
  y <- y + rnorm(1)
  iteration <- iteration + 1
}

##  ... iteration 1
##  ... iteration 2
##  ... iteration 3
##  ... iteration 4
##  ... iteration 5
##  ... iteration 6
```

```
##   ... iteration 7
##   ... iteration 8
##   ... iteration 9
##   ... iteration 10
##   ... iteration 11
##   ... iteration 12

x

## [1] 6.842638

y

## [1] 2.602371
```

`while` is often combined with `break` to bail out of a loop as soon as something interesting happens (and that something is detected with an `if`). A common approach is to set the loop to continue forever (I won't show the output, but please do try it, and understand it).

```
iteration <- 0
while(TRUE) {
  iteration <- iteration + 1
  cat(" ... iteration", iteration, "\n")
  x <- rnorm(1, mean = 5)
  y <- rnorm(1, mean = 7)
  z <- x * y
  if (z < 15) break
}
```

Did you notice that `iteration <- iteration + 1` is now in a different place, and we initialize it with a value of 0?

13.2 Defining your own functions

As the "Introduction to R" manual says (http://cran.r-project.org/doc/manuals/R-intro.html#Writing-your-own-functions)

> (...) learning to write useful functions is one of the main ways to make your use of R comfortable and productive.
>
> It should be emphasized that most of the functions supplied as part of the R system, such as mean(), var(), postscript() and so on, are themselves written in R and thus do not differ materially from user written functions.

Here we only cover the very basics. See the "Introduction to R" for more details, and the books above for even more coverage.

You can define a function doing

```
the.name.of.my.function <- function(arg1, arg2, arg3, ...) {
#what my function does
}
```

In the above, you substitute "the.name.of.my.function" by, well, the name of your function, and "arg1, arg2, arg3", by the arguments of the function. Then, you write the R code in the place where it says "#what my function does". The ... refer to other arguments passed on to functions called inside the main function (we won't get into this).

For example

```
multByTwo <- function(x) {
  z <- 2 * x
  return(z)
}

a <- 3
multByTwo(a)

## [1] 6

multByTwo(45)

## [1] 90
```

Another example

```
plotAndSummary <- function(x) {
  plot(x)
  print(summary(x))
}
x <- rnorm(50)
```

(I won't be showing the output of plotAndSummary: but make sure you do it, and understand what is going on).

```
plotAndSummary(x)
plotAndSummary(runif(24))
```

Using more arguments, one of them default:

```
plotAndLm <- function(x, y, title = "A figure") {
  lm1 <- lm(y ~ x)
  cat("\n Printing the summary of x\n")
  print(summary(x))
  cat("\n Printing the summary of y\n")
  print(summary(y))
  cat("\n Printing the summary of the linear regression\n")
  print(summary(lm1))
  plot(y ~ x, main = title)
  abline(lm1)
  return(lm1)
}

x <- 1:20
y <- 5 + 3 *x + rnorm(20, sd = 3)
```

(Again, I am not showing the output here. But make sure you understand what is happening!)

```
plotAndLm(x, y)
plotAndLm(x, y, title = "A user specified title")
output1 <- plotAndLm(x, y, title = "A user specified title")
```

Make sure you understand the difference between

```
plotAndLm(x, y)
```

and

```
out2 <- plotAndLm(x, y)
```

(hint: in the last call, you are assigning something to "out2". Look at the last line of the function "plotAndLm"). Not all functions return something, and many functions do not plot or print anything either. You decide what and how your functions do, print, plot, etc, etc, etc.

And, as we will see below, many functions are often "defined on the fly".

13.3 Order of arguments, named and unnamed arguments, etc

R is fairly flexible in how you can invoke a function and the order in which you pass arguments. But there are better and worse ways of doing it. In general, use positional matching only for the first (or first two) arguments, and avoid passing unnamed arguments after named ones. For instance:

```
f1 <- function(one, two, three) {
    cat("one = ", one,
        " two = ", two,
        " three = ", three, "\n")}

## We are OK
f1(1, 2, 3)

## one =  1  two =  2  three =  3

## We are OK, but this is getting risky
f1(two = 2, three = 3, 1)

## one =  1  two =  2  three =  3

## We are no longer OK. Nothing "strange" happened
## but we would need to be very careful. So avoid it.
f1(two = 2, 3, 1)

## one =  3  two =  2  three =  1
```

The above rules are even more important when arguments with default arguments are considered.

13.4 Scoping, frames, environments, etc

This is more advanced material. Read through it, and go deeper if you want. This is kind of jumping around on purpose; do this when you are awake.

Suppose this:

```
f1 <- function(x) {
    x + z
}
```

What will be the value of "z" used by f1? Play with it. Try to get that to work.

Or what will happen here?

```
v <- 1000
f3 <- function(x, y) {
    v <- 3 * x
    f2 <- function(u) {
        u + v
    }
    f2(y)
}
f3(2, 9)
```

(In R it is perfectly OK to define a function inside another function. In fact, it can make some code much easier to understand.)

The above questions are, basically, about "where is the value of free variables found". R uses lexical (or static) scoping. It can be extremely handy and fun, but it can lead to surprises. Good explanations here: http://adv-r.had.co.nz/Functions.html#lexical-scoping and http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf

In fact in the examples above there are several issues we might want to think about and a few terms we would need to define. There are the issues of using global variables in "f1" (which is often regarded as poor style) and there are issues of lexical scoping in what f2 and f3 are doing (and I would regard f2 as a perfectly OK function, given how f3 is, even if there is a free variable in f2). And all these relate, and need to be answered, in the context of these concepts/terms (I take here the definitions from the extremely clear PDF by John Fox, linked above, with a few modifications following Chamber's "Software for data analysis", p. 119 and ff. and chapter 8 of Wickham's "Advanced R"):

binding In `y <- 9` y is bound to the value 9.

free variable "z" is a free variable in function f1, above. It is not bound to anything (at least in that frame).

frame A set of bindings (y to 9, maybe x to 77, etc).

environment You can think of it as a sequence of frames. When f2 (well, R) goes looking for the value of "v" it will do so looking through a sequence of frames. In fact, an environment has two components: a frame and a reference to another environment, its parent environment (or its enclosing environment); since each environment has a reference to another environment, you can now easily understand the idea of an environment as a sequence of frames.

And this should bring us to consider, too, why things such as this work at all in the way we expect:

```
c

## function (..., recursive = FALSE)  .Primitive("c")

c <- 95
c

## [1] 95

c(5, 6)

## [1] 5 6

c + 8

## [1] 103
```

Which, of course, is asking questions about where (and how) R searches for things (even if we never write code such as that in f3, f2, or f1). Read, for instance, chapter 8 of "Advanced R", available for example here http://adv-r.had.co.nz/Environments.html. But maybe before that do this

```
search()
```

and try to figure out what is happening.

This is used, implicitly of explicitly, in a lot of code. We cannot pursue it here any further (though we might talk about it in class). But now, before we leave, we will remove our "c":

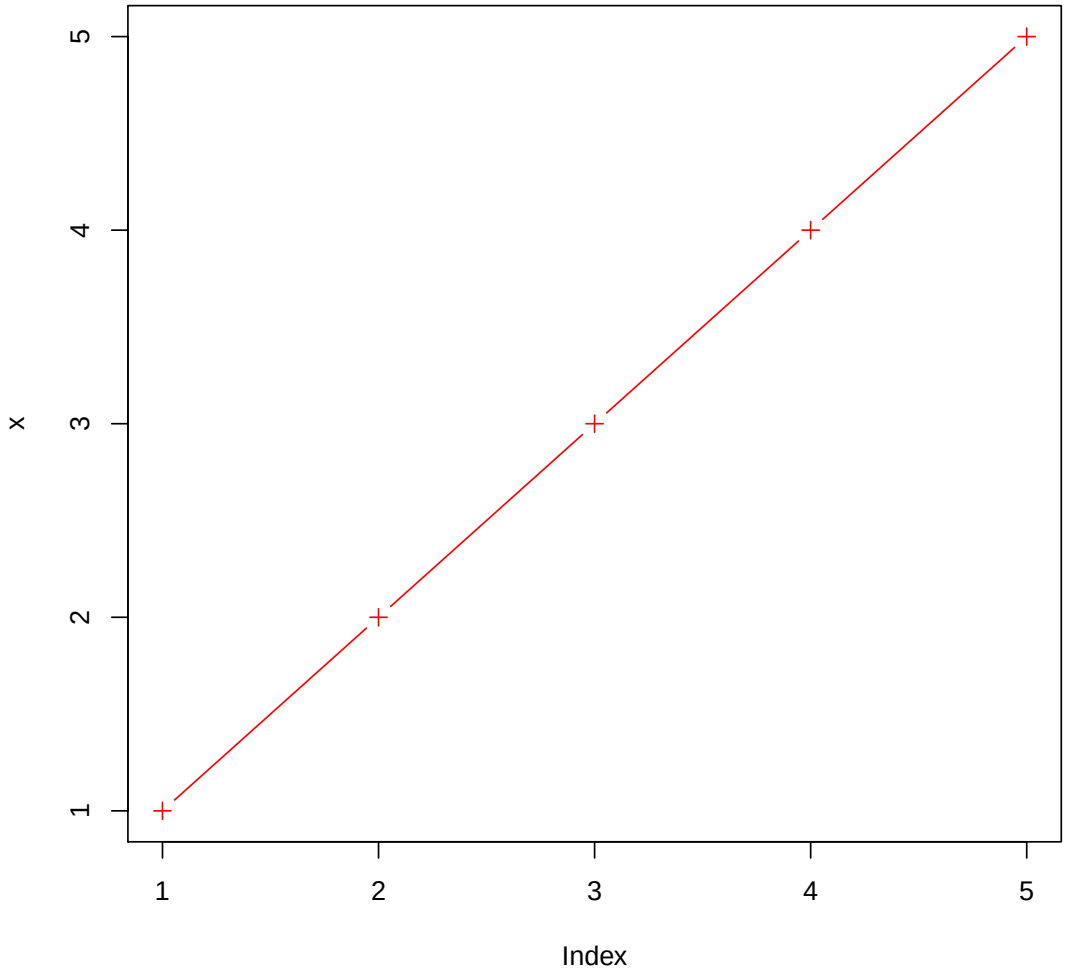
```
rm(c)
```

Oh, do you know what "c" we just removed?

13.5 The ...

The ... allows you to pass further arguments that some function inside your function will take care of:

```
f0 <- function(x, pch = 3, ...) {plot(x, pch = pch, ...)}

f0(1:5, col = "red", type = "b")
```



Make sure you understand why only `fa` does what you want:

```
fa <- function(x, col = "red", ...) {plot(x, col = col, ...)}
fa(1:5, "blue", pch = 8)

fb <- function(x, col = "red", ...) {plot(x, col = col)}
fb(1:5, "blue", pch = 8)

fc <- function(x, col = "red") {plot(x, col = col, ...)}
fc(1:5, "blue", pch = 8)
```

14 The "apply" family, "aggregate", etc

One of the great strengths of R is operating over whole vectors, arrays, lists, etc. Some available functions are: `apply, lapply, sapply, tapply, mapply`. Please look at the help for these functions. Here we will show some examples, and you should understand what is happening:

```
(Z <- matrix(c(1, 27, 23, 13), nrow = 2))

##      [,1] [,2]
## [1,]    1   23
## [2,]   27   13

apply(Z, 1, median)

## [1] 12 20

apply(Z, 2, median)

## [1] 14 18

apply(Z, 2, min)

## [1]  1 13
```

For those of you who have programmed before: using the "apply" family is often much, much, much more efficient (and elegant, and easy to understand) than using explicit loops.

With lists we will use `lapply`. For example, lets look at the first element of each of the components of the list (see how we define a function on the fly):

```
(listA <- list(one.vector = 1:10,  hello = "Hola",
               one.matrix = matrix(rnorm(20), ncol = 5),
               another.list = list(a = 5,
                 b = factor(c("male", "female", "female")))))

## $one.vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $hello
## [1] "Hola"
##
## $one.matrix
##            [,1]       [,2]       [,3]       [,4]       [,5]
## [1,]  0.1607072 -1.8864750 -1.4572324 -1.7624221 -0.1874197
## [2,] -1.6956207 -0.3518371  1.1474975  0.2601748  2.1833342
## [3,] -0.1543436  0.3042488  1.2125352  0.1620782  0.3433743
## [4,] -0.5642441 -1.3212939  0.1585878 -1.5184712  1.9336829
##
## $another.list
## $another.list$a
## [1] 5
##
## $another.list$b
```

```
## [1] male   female female
## Levels: female male

lapply(listA, function(x) x[[1]])

## $one.vector
## [1] 1
##
## $hello
## [1] "Hola"
##
## $one.matrix
## [1] 0.1607072
##
## $another.list
## [1] 5
```

When we have data that can be used to stratify or select other data, we often use `tapply`:

```
(one.dataframe <- data.frame(age = c(12, 13, 16, 25, 28),
                             sex = factor(c("male", "female",
                               "female", "male", "male"))))

##   age    sex
## 1  12   male
## 2  13 female
## 3  16 female
## 4  25   male
## 5  28   male

tapply(one.dataframe$age, one.dataframe$sex, mean)

##   female     male
## 14.50000 21.66667
```

However, `aggregate` often returns things in a more convenient form:

```
(one.dataframe <- data.frame(age = c(12, 13, 16, 25, 28),
                             sex = factor(c("male", "female",
                               "female", "male", "male"))))

##   age    sex
## 1  12   male
## 2  13 female
## 3  16 female
## 4  25   male
## 5  28   male

aggregate(one.dataframe$age, list(one.dataframe$sex), mean)

##   Group.1        x
## 1  female 14.50000
## 2    male 21.66667
```

```
## make the aggregating variable explicit,
## and give it another name
aggregate(one.dataframe$age,
          list(Sexo = one.dataframe$sex), mean)

##     Sexo        x
## 1 female 14.50000
## 2   male 21.66667

## or use the name of the column/variable
aggregate(one.dataframe$age,
          one.dataframe[2], mean)

##      sex        x
## 1 female 14.50000
## 2   male 21.66667
```

The function `by` is related to `aggregate` and `tapply`, but you can use functions that return several different things (e.g., the mean and standard deviation) and the return value is a list:

```
by(one.dataframe$age, list(one.dataframe$sex),
   function(x) c(mean(x), sd(x)))

## : female
## [1] 14.50000  2.12132
## ------------------------------------------------
## : male
## [1] 21.666667  8.504901
```

You can do much of that with aggregate too, but the output is different:

```
aggregate(one.dataframe$age, one.dataframe[2],
          function(x) c(Mean = mean(x), SD = sd(x)))

##      sex    x.Mean      x.SD
## 1 female 14.500000  2.121320
## 2   male 21.666667  8.504901
```

Since the need for these operations is very common (getting summaries or applying functions to subsets of the data), there are a variety of ways of accomplishing them with the basic functions of R, as well as several additional packages that provide for different (easier? faster?) approaches/syntax. Here is a blog with some examples and links: http://lamages.blogspot.com.es/2012/01/say-it-in-r-with-by-apply-and-friends.html. For instance, the `doBy` package is really nice.

=======

14.1 Matrices: Dropping dimensions

Look at the different outputs of the selection operation:

```
(E <- matrix(1:9, nrow = 3))
```

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

```
E[, 1]
```

```
## [1] 1 2 3
```

```
E[, 1, drop = FALSE]
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

```
E[1, ]
```

```
## [1] 1 4 7
```

```
E[1, , drop = FALSE]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
```

Unless we use `drop = FALSE`, if we select only one row or one column, the result is not a matrix, but a vector[4]. But sometimes we do we need them to remain as matrices. That is often the case in many matrix operations, and also when using `apply` and related.

Suppose we select automatically (with some procedure) a set of rows that interest us in the matrix.

```
rows.of.interest <- c(1, 3)
```

We can do

```
apply(E[rows.of.interest, ], 1, median)
```

```
## [1] 4 6
```

Now, imagine that in a particular case, `rows.of.interest` only has one element:

```
rows.of.interest <- c(3)
```

```
apply(E[rows.of.interest, ], 1, median)
```

What is the error message suggesting?

But we can make sure our procedure does not crash by using `drop = FALSE`:

[4] row vector? column vector? that is a longer discussion than warranted here; nothing with two dimensions, anyway

```
apply(E[rows.of.interest, , drop = FALSE], 1, median)

## [1] 6
```

The situation above can be even more confusing with some matrix operations.

To summarize: when you select only a single column or a single row of an array, think about whether the output should be a vector or a matrix

15 (Optional: two handy packages `readr` and `dplyr`)

We won't get into details about this: Hadley Wickham has two packages, `readr` and `dplyr` that can ease considerably some tasks of data manipulation (dplyr) and reading large flat files (readr). Both are explained, in addition to the documentation of each of the packages, in Peng's "R programming for data science" book.

16 Revisiting an example that brings a few things together

This should not be mysterious now (you might want to look at the help for `hist` and `order`). We want to reproduce a fairly common analysis that is done in genomics; here we will simulate the data. The steps are:

1. Generate a random data set (samples in columns, variables or genes in rows); there are 50 subjects and 500 genes.
2. Of the 50 subjects, the first 30 are patients with colon cancer, the next 20 with lung cancer
3. For each "gene" (variable, row) do a t-test
4. Find out how many p-values are below 0.05, and order p-values. Plot p-values.

```
randomdata <- matrix(rnorm(50 * 500), ncol = 50)
class <- factor(c(rep("NC", 20), rep("cancer", 30)))
```

Do we know what the output from a *t-test* looks like? What do we want to select? Lets play a little bit:

```
tmp <- t.test(randomdata[1, ] ~ class)
tmp

##
##  Welch Two Sample t-test
##
## data:  randomdata[1, ] by class
## t = 0.66111, df = 45.11, p-value = 0.5119
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.3876809  0.7665803
## sample estimates:
## mean in group cancer     mean in group NC
##           0.17418645          -0.01526323

attributes(tmp)

## $names
## [1] "statistic"   "parameter"   "p.value"     "conf.int"
## [5] "estimate"    "null.value"  "alternative" "method"
## [9] "data.name"
##
## $class
## [1] "htest"

tmp$p.value

## [1] 0.5119058
```

OK, so now we know what to select (note: I do NOT show the results of the computations here!):

```
pvalues <- apply(randomdata, 1,
                 function(x) t.test(x ~ class)$p.value)
hist(pvalues)
order(pvalues)
which(pvalues < 0.05)
```

```
sum(pvalues < 0.05)
```

Now, repeat all of this by running the script:

```
source("lastExample.R")
sum(pvalues < 0.05)
```

Please, look at "lastExample.R", and understand what happened. You need to repeat the `sum(pvalues < 0.05)`. You could modify "lastExample.R", to explicitly print results. Or you can use `source("lastExample.R", echo = TRUE)`.

Please, understand what is happening.

17 Go back to the scenarios

You should now be able to go back to section 2 and solve all of them. If they ask you for data, you should know how to simulate data to pretend you have been given some data.

18 Debugging and catching exceptions

Often, things do not work as expected, and we need to debug our code. Debugging is a big issue. As Matloff insists, debugging is basically about testing your assumptions about what a piece of code is doing. In what follows, I provide just a quick summary of really helpful things that I use often (there are some extra techniques and approaches, but really understanding the ones below will take you far).

Anyway, here are some additional places to look at: http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR/, http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR/debug.shtml, http://www.biostat.jhsph.edu/%7Erpeng/docs/R-debug-tools.pdf, and chapter 9 in "Advanced R" (available here : http://adv-r.had.co.nz/Exceptions-Debugging.html) and chapter 13 in the "The art of R programming".

(Oh, the examples below will show little output in these notes. Make sure you do type the code and play around: these are necessarily interactive things, and that does not play well with static notes).

18.1 What exactly broke?

`traceback` shows the call stack and helps you identify where things crashed, so you can tell which is the function where the problem shows up:

```
f1 <- function(x) 3 * x

f2 <- function(x) 5 + f1(x)

f3 <- function(z, u) {
    v <- runif(z)
    a <- f2(u)
    b <- f2(3 * v)
    return(a + b)
}
f3(3, 7)

## [1] 32.78428 36.27160 36.15692
```

```
f3(-5, 6)
traceback()

f3(5, "a")
traceback()
```

That, however, is only part of the process. In the case above, we know sometimes it is the call to `runif` and some times is the call to `f1`. But we might want to look closer at what is happening.

18.2 `debug` and `browser`

If something breaks, you can go step by step over the execution of the code. For that, call `debug(theFunctionYouWantToDebug)`. Note that you can use `debug` with functions that you have not written. You do not need to edit or modify the source. Once you are done, `undebug(theFunctionYouWantToDebug)`.

Try the following. When the evaluation stops, press "c" if you do not want to go into debugging mode. And you can exit from the whole debugging process by "Q"

```
debug(f3)
f3(3, 5)
undebug(f3) ## stop debugging
f3(3, 5)
```

You can also use `browser`. It is very easy to add that to a function you wrote, wherever you want it to stop. Make sure you try this, and when it stops, look around (e.g., type `ls()`, reassign values, etc).

```
f3(3, 5)
f3(3, 11)
```

Or you can set up conditional browsers. For example, call it only if `z > 5`:

```
f3 <- function(z, u) {
    v <- runif(z)
    if (z > 5) browser()
    a <- f2(u)
    b <- f2(3 * v)
    return(a + b)
}
```

It is bad practice to leave `browsers` around (for obvious reasons). Likewise, you should `undebug` when you are done, or source again your function (which will have the effect of undebugging it).

18.3 Browsing arbitrary functions at arbitrary places

Can you place `browser` at arbitrary places in arbitrary functions including, say, from a package or the base system you have not written? Yes. You just have to pass it the line number, using the function `trace`. And how do you find the line number? Use `as.list(body(theFunctionname)`.

Let's start with our simple function:

```
as.list(body(f3))
trace(f3, tracer = browser, at = 3)

f3 # notice the message
```

```
f3(4, 5)
untrace(f3) ## stop tracing
```

This uses `lm`, which we have not written.

```
as.list(body(lm))
trace(lm, tracer = browser, at = 5)
y <- runif(100)
x <- 1:100
lm(y ~ x)
## stop tracing
untrace(lm)
```

18.4 Autopsies when things fail

But sometimes you only want to stop when/if things fail, and look inside. We can ask R to do that: stop when the error occurs, and allow us to look around:

```
opt <- options(error = recover)
## Notice the next show, as they should,
## different frame numbers. The error is in different
## places. You can enter in the relevant frames
## and look around
f3(3, "a")
f3(-5, 3)
```

You will be shown the possible frames, you can enter the one you want, and look around, etc. This is **extremely handy** in some situations, or when you only get errors from time to time, etc.

Once you are done, set the option back to what it was:

```
options(opt)
```

18.5 And RStudio?

The above are general mechanisms that will work regardless of how you use R. If you use RStudio, there is additional functionality available. Suppose you have code like this:

```
f1 <- function(x) 3 * x

f2 <- function(x) 5 + f1(x)

f3 <- function(z, u) {
    v <- runif(z)
    f2(v + u)
}
f3(3, "a")
```

If you submit all that code to be executed, you will get an error, and RStudio will offer you to run it again with debug (it will be fairly obvious, just do it).

The chapter from "Advanced R" explains in detail RStudio's functionality in this area, which includes setting breakpoints and other additional features.

Emacs + ESS has also its own way of doing these things.

18.6 And warnings?

If you get warnings in your own code, but you don't understand it, you can turn warnings into errors, and use the above debugging approaches. Just do

```
opt <- options(warn = 2)
```

And then, a warning will behave as an error. Once you are done, return things back to what they were:

```
options(opt)
```

18.7 Confused about where you are?

Sometimes, while debugging, especially if you have turned debugging on several functions, some of which call some others, you might get lost, and not know where you are. In such a case, type `where`. Yes, like that, not `where()`. It will give you the call stack and you will see where you are. You can try it:

```
debug(f1); debug(f2); debug(f3)
f3(4, 5) ## now, keeping pressing enter or n
         ## and you'll get deeper and deeper
         ## while in browser mode, type where
```

Return things to normal

```
undebug(f3)
undebug(f2)
undebug(f1)
```

18.8 Protecting from possible failures

There is exception handling in R, of course, and there are several mechanisms. I often use `try`, which is simple. For instance:

```
ft <- function(x) {
    tmp <- try(log(x), silent = TRUE)
    if(inherits(tmp, "try-error")) {
        warning(paste("It looks like something did not work:\n",
                      "   ", tmp))
    } else{
        return(tmp)
    }
}

ft(9)

## [1] 2.197225

ft("a")

## Warning in ft("a"):  It looks like something did not work:
##      Error in log(x) :  non-numeric argument to mathematical function
```

The above is a silly example, of course, but suppose taking the log is not a fundamental part of the code, and you want to be allowed to continue, or you want to be the one who decides what to do without breaking a long analysis. This prevents the code from just failing. I use try very, very often when I run long simulation analysis that can take weeks: I record fully when a case fails (keep track of data, random number seeds, etc), but I allow the rest of the process to continue.

More sophisticated approaches are available. Look at the references above.

19 Object-oriented programming and classes S3 and S4

Is object-oriented programming possible in R? Yes. In fact, there are three different systems for OOP. But we will not cover any in this course and, in fact, nothing in these notes (except for a possibly cryptic reference to `predict.randomForest`, in section 20.3) requires you to understand any of this. But it can become important when you write code and it will become relevant when you read code from others. Look at section 10.9 in "An introduction to R" (http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Object-orientation), included with your R, which only covers S3, and chapter 7 of Wickham's "Advanced R", http://adv-r.had.co.nz/OO-essentials.html, which covers S3, S4, and reference classes, for more details. S3 is a relatively simple approach, and is the one used by most CRAN packages (and the one used in base and stats in R); S4 is very popular in BioConductor, but is much more complicated (and, I'd say, cumbersome).

20 Additional programming practice

The following sections offer some additional programming practice. Please use what you have seen above, and especially take the following as opportunities to practice debugging.

20.1 Those common genes and some Venn diagrams

Someone has examined what genes are over-expressed in three different conditions (three different drugs)[5]. They are called "Condition_A.txt", "Condition_B.txt", "Condition_C.txt". You want to understand which are over-expressed under more than one drug. You might want to know which are overexpressed in three conditions, or which are overexpressed in exactly these two conditions, etc. Or find out how many are overexpressed under two of the drugs but not the third. Etc.

We will walk over a case here, thinking about making the solution as general as possible (e.g., with other file names, and with different number of drugs), in case we face the same problem in the future.

First, read the data. They are just three vectors:

```
A <- scan("Condition_A.txt", what = "")
B <- scan("Condition_B.txt", what = "")
C <- scan("Condition_C.txt", what = "")
```

Which are overexpressed in both A and B? And in A, B, C?

```
sort(intersect(A, B))
```

```
##  [1] "ADM"    "BNIP3"  "BNIP3L" "CXCR4"  "DDIT4"  "EGLN1"
##  [7] "ENO1"   "ERO1L"  "HK1"    "LDHA"   "LOX"    "MIF"
## [13] "MXI1"   "P4HA1"  "PFKP"   "PGK1"   "PLOD2"  "RORA"
## [19] "VEGFA"
```

```
sort(intersect(A, intersect(B, C)))
```

```
##  [1] "ADM"    "BNIP3"  "BNIP3L" "CXCR4"  "DDIT4"  "EGLN1"
##  [7] "ENO1"   "ERO1L"  "HK1"    "LDHA"   "LOX"    "MIF"
## [13] "MXI1"   "P4HA1"  "PFKP"   "PGK1"   "PLOD2"  "RORA"
## [19] "VEGFA"
```

But that is ugly because, what if we had 15 treatments? Or different names? Let's assume only "Condition_" is fixed. And this thing of calling `intersect` many times ... ugly too. And it is hard to see how many are common to A and B, and A and C, etc easily. So we would like a matrix that has, as columns, the conditions, and as rows the gene names. In each entry, a TRUE means overexpressed, and a FALSE not-overexpressed.

Let's try something else, and try to be a little bit more general.

First, try to make reading the genes a simple thing. Place all of them in a list, with as many elements as conditions.

```
####   Read all lists of genes that start with "Condition"
####    and store   as a list
gf <- dir(pattern = "^Condition_")
ovxpGenes <-  sapply(gf, function(x) scan(x, what = ""))
```

There is nothing wrong with the call to `sapply`, but using `lapply` there might be preferable.

But "Condition_" is repeated as a prefix. And txt as postfix. Get rid of them in the names of the list:

[5] This data have been kindly provided by Luis del Peso, at the Department of Biochemistry, Universidad Autónoma de Madrid. http://www.iib.uam.es/persona?id=lpeso

```
names(ovxpGenes) <-
    sapply(names(ovxpGenes),
           function(x) strsplit(strsplit(x,
                                         "Condition_")[[1]][2],
                                "\\.txt")[[1]][1])
```

The above can also be done as:

```
## First line is to get the ovxpGenes names
ovxpGenes <-  sapply(gf, function(x) scan(x, what = ""))

names(ovxpGenes) <- sapply(strsplit(names(ovxpGenes),
                                    "Condition_"),
                           function(x)
                               strsplit(x[[2]], "\\.txt")[[1]][1])
```

Now, prepare an object that is very easy to use for any further analysis: matrix that has, as columns, the conditions, and as rows the gene names. In each entry, a 1 means overexpressed, and a 0 not-overexpressed.

There are several ways of doing this. This is the simplest I came up with.

First, find the union of all gene names and return a single entry of each name. We do not use `union`, but `unique`, which does what we want here.

```
all.the.genes <- sort(unique(unlist(ovxpGenes)))
```

Which of those genes are in the first condition? Or which, among all.the.genes, are overexpressed in condition A? This works

```
head(match(ovxpGenes[[1]], all.the.genes))

## [1]  920  138  229    9 1532 1247
```

(I used `head` to show only the first part of the output).

Can we do that for all the lines, without having to explicitly do that for every single cell line? This kind of works . . . but the output is not in an easily usable form: `sapply` cannot simplify as the return objects are of different length.

```
str(sapply(ovxpGenes, function(z) {match(z, all.the.genes)}))

## List of 3
##  $ A: int [1:487] 920 138 229 9 1532 1247 770 930 348 391 ...
##  $ B: int [1:737] 125 333 1125 1657 309 1003 1427 1582 497 74 ...
##  $ C: int [1:503] 904 532 1365 99 520 1140 350 1387 1021 1523 ...
```

(I used `str` to show only the first part of the output).

We want to return a TRUE where the gene is overexpressed and a FALSE if not. If we[6]. In other words, we want a vector for each experiment with the positions filled up. Lets do that directly as the return of a function:

[6] I find it better to be as explicit as possible. An alternative would be to return a 1 of type integer, using "1L", and a 0 of type integer (0L).

```
f1 <- function(x, all.the.genes) {
    ## Note: this could be done in fewer lines, or even
    ## by an anonymous function used in sapply
    vv <- rep(FALSE, length(all.the.genes))
    pos.match <- match(x, all.the.genes)
    vv[pos.match] <- TRUE
    return(vv)
}
```

See if it works

```
head(f1(ovxpGenes[[1]], all.the.genes))

## [1] FALSE FALSE FALSE  TRUE FALSE FALSE

## and on the third?
head(f1(ovxpGenes[[3]], all.the.genes))

## [1]  TRUE FALSE FALSE FALSE  TRUE FALSE
```

But this is probably clearer? Using a lookup table kind of approach

```
f2 <- function(x, all.the.genes) {
    vv <- rep(FALSE, length(all.the.genes))
    names(vv) <- all.the.genes
    vv[x] <- TRUE
    return(vv)
}
```

I leave it as an exercise to show that both functions, `f1` and `f2` are doing the same thing. We are almost done:

```
overexpressed <- sapply(ovxpGenes,
                        function(x) f1(x, all.the.genes))
rownames(overexpressed) <- all.the.genes
```

That works. However, I think there is merit in following Hadley Wickham's advice of avoiding `sapply` inside functions (see section 9.4 of his "Advanced R"), and in a second we will be putting some of the above inside functions. So let's be more strict:

```
overexpressed <- vapply(ovxpGenes,
                        function(x) f1(x, all.the.genes),
                        logical(length(all.the.genes)))
rownames(overexpressed) <- all.the.genes
```

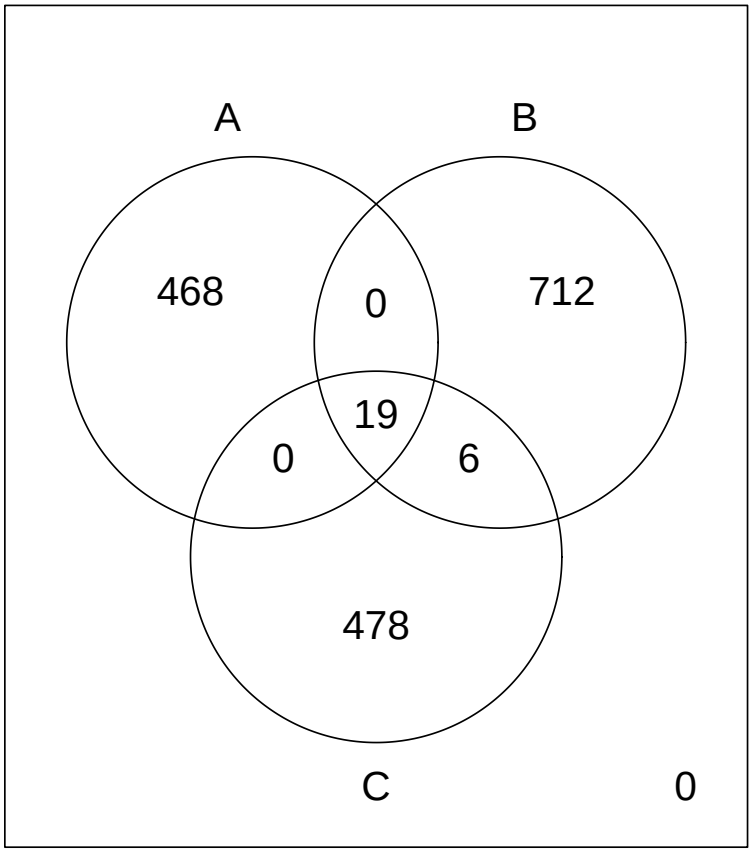
Now, a few checks:

```
## note the following two match
colSums(overexpressed)

##   A   B   C
## 487 737 503
```

```
lapply(ovxpGenes, length)

## $A
## [1] 487
##
## $B
## [1] 737
##
## $C
## [1] 503

# But if automatic, better to use stopifnot
stopifnot(colSums(overexpressed) ==
              unlist(lapply(ovxpGenes, length)))

## And check names of genes
h <- rownames(overexpressed)[which(overexpressed[, "A"] == 1)]
stopifnot(identical(h, sort(ovxpGenes[["A"]])))
```

Now, a plot after all this work (you will need to install "limma" —go to section 5.4)

```
library(limma)
vennDiagram(overexpressed)
```



And it is now also trivial to find out which are overexpressed in arbitrary conditions. For instance (the number is the row number), overexpressed in both A and B:

```
which(overexpressed[, "A"] & overexpressed[, "B"])
```

```
##     ADM   BNIP3  BNIP3L   CXCR4   DDIT4   EGLN1    ENO1   ERO1L
##      30     148     149     305     325     396     415     438
##     HK1    LDHA     LOX     MIF    MXI1  P4HA1    PFKP    PGK1
##     626     765     785     861     893     991    1045    1048
##   PLOD2    RORA   VEGFA
##    1077    1224    1583
```

And which are induced in all?

```
which(rowSums(overexpressed) == 3)
```

```
##     ADM   BNIP3  BNIP3L   CXCR4   DDIT4   EGLN1    ENO1   ERO1L
##      30     148     149     305     325     396     415     438
##     HK1    LDHA     LOX     MIF    MXI1  P4HA1    PFKP    PGK1
```

```
##    626    765    785    861    893    991   1045   1048
##  PLOD2   RORA  VEGFA
##   1077   1224   1583

## or safer?
which(rowSums(overexpressed) > 2.99)

##    ADM  BNIP3 BNIP3L  CXCR4  DDIT4  EGLN1   ENO1  ERO1L
##     30    148    149    305    325    396    415    438
##    HK1   LDHA    LOX    MIF   MXI1  P4HA1   PFKP   PGK1
##    626    765    785    861    893    991   1045   1048
##  PLOD2   RORA  VEGFA
##   1077   1224   1583
```

And how many are overexpressed in 3, in 2, and in 1?

```
table(rowSums(overexpressed))

##
##    1    2    3
## 1658    6   19
```

Etc, etc.

By the way, once we have the list of genes, finding the intersection of all is simple (this might be considered more advanced, but is great because this will work with lists of arbitrary number of components):

```
sort(Reduce(intersect, ovxpGenes))

##  [1] "ADM"    "BNIP3"  "BNIP3L" "CXCR4"  "DDIT4"  "EGLN1"
##  [7] "ENO1"   "ERO1L"  "HK1"    "LDHA"   "LOX"    "MIF"
## [13] "MXI1"   "P4HA1"  "PFKP"   "PGK1"   "PLOD2"  "RORA"
## [19] "VEGFA"
```

20.1.1 A few functions to do the job automatically

Drawing the Venn diagram and the last operations of tables and finding out which are overexpressed is the fun thing. But we had to do some work that, if we are going to repeat, we might want to automate. Let's do that: Notice that all we need to do is just generalize slightly over what we did. That is all.

Note, however, that inside the new functions I will often shorten the names of variables or at least make them slightly different from above. Why? Because inside the function I can afford to shorten it, and it leads to cleaner code, and because I avoid accidentally using objects previously created (this could happen because of R's scoping rules).

First, the reading data part. We will assume that we will always find all files with the gene identity in a directory, and that all files will have a common prefix and end in a common postfix. We will use default values, though.

```
readListGenes <- function(prefix = "Condition_",
                          postfix = "txt") {
```

```
    ## Read a bunch of files, named "prefixSOMETHING.postfix",
    ## and place the gene names inside in a list
    gg <- dir(pattern = paste0("^", prefix))
    ovG <-  sapply(gg, function(x) scan(x, what = ""))
    matchpost <- paste0("\\.", postfix)
    names(ovG) <-
    sapply(names(ovG),
           function(x) strsplit(strsplit(x,
                                         prefix)[[1]][2],
                                matchpost)[[1]][1])
    return(ovG)
}
```

Now, place everything in a matrix. First, give `f1` a better name (and you could even define `trueIfMatch` inside `geneListToMatrix`):

```
trueIfMatch <- function(x, allgenes) {
    ## Note: this could be done in fewer lines, or even
    ## by an anonymous function used in sapply
    vv <- rep(FALSE, length(allgenes))
    pos.match <- match(x, allgenes)
    vv[pos.match] <- TRUE
    return(vv)
}
geneListToMatrix <- function(lgenes) {
    ## For you: add a meaningful comment here!
    allgenes <- sort(unique(unlist(lgenes)))
    ox <- vapply(lgenes,
                 function(x) trueIfMatch(x, allgenes),
                 logical(length(allgenes)))
    rownames(ox) <- allgenes
    return(ox)
}
```

Now, create a function that will do everything:

```
geneFiles2Mat <- function(prefix = "Condition_",
                          postfix = "txt") {
    l1 <- readListGenes(prefix = prefix, postfix = postfix)
    return(geneListToMatrix(l1))
}
```

And now just call it:

```
ovxABC <- geneFiles2Mat()
```

Wait: two final checks:

```
stopifnot(identical(sort(Reduce(intersect, ovxpGenes)),
                    sort(Reduce(intersect, readListGenes()))
```

```
                        ))

stopifnot(identical(ovxABC, overexpressed))
```

20.1.2 And how do we know it works?

If we were more serious about this, probably formally adding tests would be warranted. See section 20.2.2. We've done some testing about, but this can be done much more seriously.

20.2 Permutation test

We will write a permutation test for comparing two means. Before we get carried away, the code below is not the most efficient. More importantly, we will not deal with important time saving ideas (such as using equivalent statistics, an idea Edgington stressed a lot in his books) nor with important statistical ideas (such as using sufficient statistics). This is just for the sake of writing a quick and dirty permutation test and gaining some programming practice. Note also that you have code readily available for this and similar tasks in several R packages. Permutation tests are extremely powerful, but also misleadingly simple; using them correctly is often more subtle than some "data scientists" who've never taken a stats course would lead you to believe (e.g., are you really testing what you think you are testing?), and there are many scenarios/questions that might be hard to fit in them (i.e., they are not **the** solution to the world's problems); *caveat emptor*.

If you do not remember or know what a permutation test is, look it up. I'll assume you have a general idea. Stop here about the main steps (remember that breaking the problem down into manageable and meaningful, smaller, self-contained, problems is a key part of programming).

Now, after thinking, this is what we need to do:

1. Compute our statistic (the difference in means here).
2. Generate data distributions according to the null hypothesis.
3. Compute the statistic for each new data configuration.
4. Compare what we compute from our original data with what we obtain under the null (H_o).

To play around, let's create some extreme fake data:

```
# no difference
d11 <- rnorm(15)
d12 <- d11[1:7]
# difference
d21 <- rnorm(13)
d22 <- d21[1:9] + 3
```

You can check the data will do what you want

```
t.test(d11, d12)
t.test(d21, d22)
```

20.2.1 First attempt

Now, the code:

```
# for steps 1 and 3
mean.d <- function(x1, x2)
  mean(x1) - mean(x2)

## seems to work
mean.d(d11, d12)

## [1] -0.116088
```

For step 3 we will examine to ways of doing it. Make sure you understand the differences

```
sample.and.stat1 <- function(x1, x2) {
    tmp <- sample(c(x1, x2))
    g1 <- tmp[seq(x1)]
    g2 <- tmp[seq(from = length(x1) + 1,
                  to = length(tmp))]
    return(mean.d(g1, g2))
}

sample.and.stat2 <- function(x1, x2) {
    indices <- seq(from = 1,
                   to = length(x1) + length(x2))
    indices1 <- sample(indices, length(x1))
    indices2 <- setdiff(indices, indices1)
    alldata <- c(x1, x2)
    return(mean.d(alldata[indices1],
                  alldata[indices2]))
}
```

Compare them:

```
set.seed(1)
sample.and.stat1(d11, d12)

## [1] -0.8058759

sample.and.stat1(d11, d12)

## [1] -0.2023938

set.seed(1)
sample.and.stat2(d11, d12)

## [1] -0.8058759

sample.and.stat2(d11, d12)

## [1] -0.2841083
```

Make sure you understand what happened. (Hint: are both functions implicitly calling the random number generator the same number of times?)

Now, wrap the above, and create a single function:

```
permut.testA <- function(data1, data2,
                         fsample = sample.and.stat1,
                         num.permut = 100) {
    obs.stat <- mean.d(data1, data2)

    permut.stat <- replicate(num.permut,
                             fsample(data1, data2))
    pv <- (sum(abs(permut.stat) >
                   abs(obs.stat)) + 1)/(num.permut + 1)
```

```
    message("\n p-value is ", pv, "\n")
    return(list(obs.stat = obs.stat,
                permut.stat = permut.stat,
                pv = pv))
}
```

Make sure you understand what we've done. For instance, note that we are passing a function as an argument[7]

Is it working?

```
tmp <- permut.testA(d11, d12)

##
##  p-value is 0.742574257425743

tmp <- permut.testA(d21, d22)

##
##  p-value is 0.0099009900990099
```

It looks like it is.

Well, in fact it is **wrong**, but we might not have noticed it. What p-value would you expect?

```
tmp <- permut.testA(d11, d11)

##
##  p-value is 1

tmp <- permut.testA(d12, d12)

##
##  p-value is 0.96039603960396
```

Moral: check, check, check. One should check against different scenarios. (In fact, the above might often look correct a lot of the time; try running the comparison of d11 with itself with 100000 permutations).

20.2.2 Test!!!

If we had more time, we would **definitely** write a test suite, and add some of the above as test cases. We cannot get into details, but there is one thing called **test-driven development** (http://en.wikipedia.org/wiki/Test-driven_development).

Maybe you do not need/want to follow it completely, but writing and using tests should be a must for any serious project. In R there are a couple of packages that help: "RUnit" is a popular one, though "testthat" might be a lot simpler(http://4dpiecharts.com/2014/05/12/automatically-convert-runit-tests-to-testthat-tests/). For using "testthat" see http://r-pkgs.had.co.nz/tests.html and http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

[7] As for the denominator and numerator (the `+ 1`): that is a minor technical detail, and it is true that it makes little difference as $n \to \infty$. However, that is the correct way of doing it (even if many implementations get it wrong and do not include it). Some of the popular classics, such as Edgington, or Noreen, etc, already mention it very clearly. Think about it: why should you add that 1?

20.2.3 Second attempt

I will give the fixed code. You have to understand what changed:

```
permut.testB <- function(data1, data2,
                         fsample = sample.and.stat1,
                         num.permut = 100) {
    obs.stat <- mean.d(data1, data2)

    permut.stat <- replicate(num.permut,
                             fsample(data1, data2))
    pv <- (sum(abs(permut.stat) >=
                   abs(obs.stat)) + 1)/(num.permut + 1)

    message("\n p-value is ", pv, "\n")
    return(list(obs.stat = obs.stat,
                permut.stat = permut.stat,
                pv = pv))
}
```

Quick check (you should do it more thoroughly)

```
tmp <- permut.testB(d11, d11)

##
##  p-value is 1

tmp <- permut.testB(d12, d12)

##
##  p-value is 1
```

20.2.4 Final thing

OK, that is good. But let's add a figure so things look more like in textbooks:

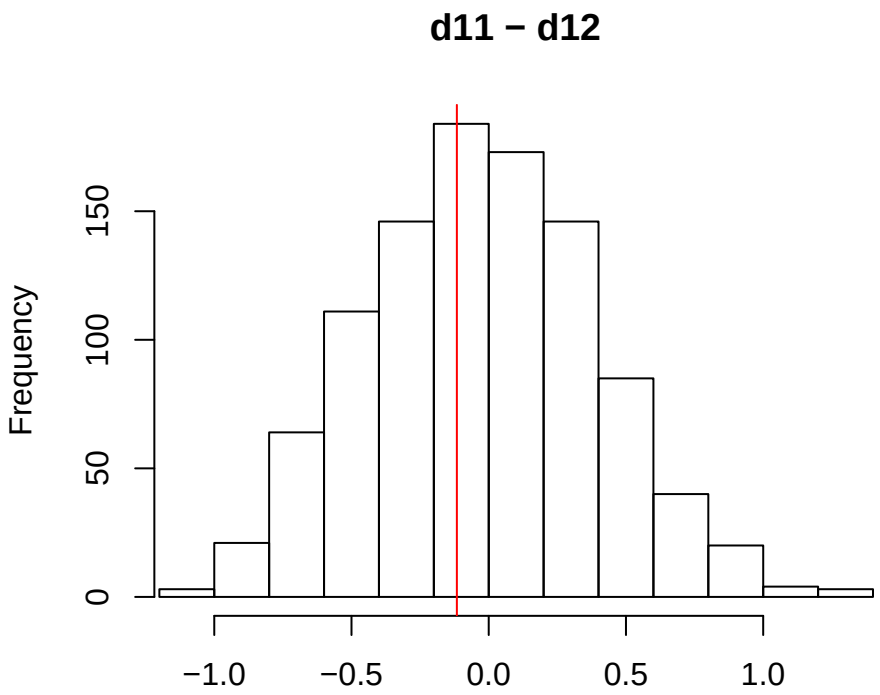
```
permut.test <- function(data1, data2,
                        fsample = sample.and.stat1,
                        num.permut = 100) {
  obs.stat <- mean.d(data1, data2)
  permut.stat <- replicate(num.permut,
                           fsample(data1, data2))
  pv <- (sum(abs(permut.stat) >= abs(obs.stat)) +
         1) / (num.permut + 1)
  message("\n p-value is ", pv, "\n")
  title <- paste(deparse(substitute(data1)), "-",
                 deparse(substitute(data2)))
  subtitle <- paste0("Distribution of permuted statistic.",
           " In red, observed one.")
  hist(permut.stat, xlim = c(min(obs.stat,
                       min(permut.stat)),
```

```
                        max(obs.stat,
                            max(permut.stat))),
       main = title, xlab = "", sub = subtitle)
  abline(v = obs.stat, col = "red")
  return(list(obs.stat = obs.stat,
              permut.stat = permut.stat,
              pv = pv))
}
```

Run it

```
par(mfrow = c(2, 1))
tmp <- permut.test(d11, d12, num.permut = 1000)

##
##  p-value is 0.794205794205794

tmp <- permut.test(d21, d22, num.permut = 1000,
                   fsample = sample.and.stat2)

##
##  p-value is 0.000999000999000999
```

You might want to compare with t-tests, etc.



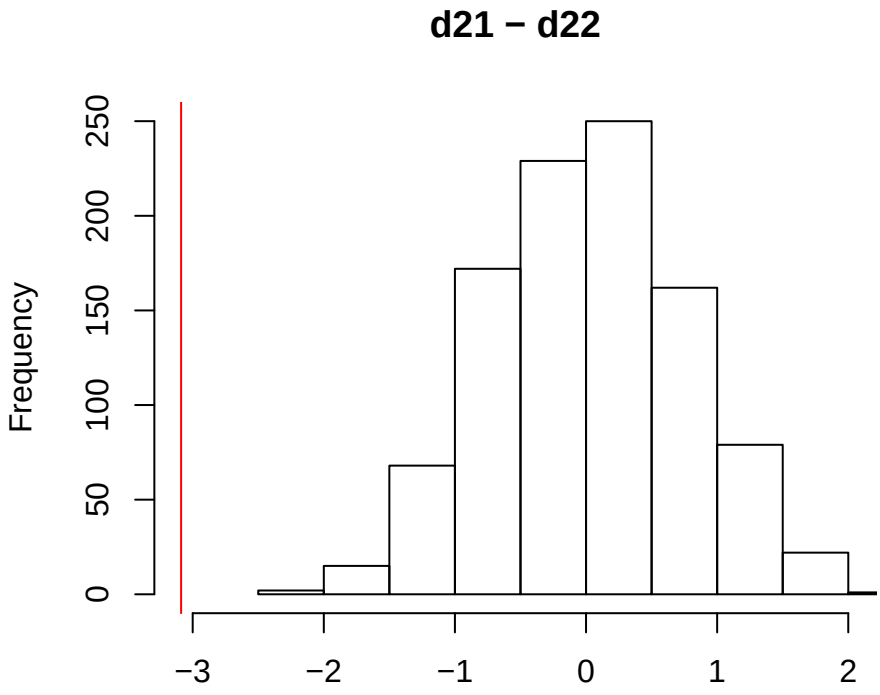


Figure 5: Two examples of permutation tests

20.3 Selection bias in classification

This is for you to work on your own. Show the effects of selection bias in classification problems. This is what we want:

- Simulate some data, without signal, for a number of subjects and a two-class problem.
- For simplicity use the randomForest algorithm.
- For simplicity, use cross-validation.
- Filter genes using the p-value from a t-test.
- Of course, the selection bias problem will arise when you filter genes using all subjects.
- You should also cross-validate including filtering within the cross-validation (i.e., do not filter genes using all subjects). This is, as we all know, the correct procedure.

A few hints:

- Install the "randomForest" package.
- Note that in randomForest you can fit a random forest to a set of data, and predict on another.
- The simplest thing is to get just the class prediction. It would be even better to also get the "votes" and assess classification accuracy using something like Brier's score or coefficient of concordance.
- For cross-validation, the following lines of code show a very compact way of doing it. The code is taken from Venables and Ripley, "S programming", 2000, Springer, p. 175, and I have added a few notes and an example:

```
## x2 is a data vector
x2 <- rnorm(27)
N <- length(x2)
knumber <- 10  ## the k in k-fold
## In the k-th time, we leave out as testing set those
## subjects that have index.select =
## Thus, index.select is the vector of indices
index.select <- sample(rep(1:knumber,
                           length = N),
                       N, replace = FALSE)

rep(1:knumber, length = N)
table(index.select)
sum(index.select != 1)
sum(index.select != 10)
sum(index.select != 6)
```

Once you have that, you can do something like

```
do.something.with.train.and.test <- function(train, test) {
    ## As it says, it does something with the
    ## two sets of data
}

for(sample.number in 1:knumber) {
    x2.train <- x2[index.select != sample.number]
    x2.test <- x2[index.select == sample.number]
    do.something.with.train.and.test(x2.train, x2.test)
}
```

- We will train a model with a set of data, and test it with those left out. Again, with random forest, use `randomForest` and `predict` (which is really `predict.randomForest`). Look at the help.
- For the simplest "classification error" we just compared observed class with predicted class. How do you do it?
- Beware: randomForest (as most other modeling functions) expects data with subjects in rows and variables (genes) in columns.
- For one simulated data set, you can get the effects of selection bias. But you will want to repeat this several times.
- Oh, how do you compare? Is this like a paired design (for each simulated data set, you get two numbers, one under selection bias and one under non-selection bias)? Or is it something else? Make sure that how you show your results reflects how you perform the simulations.
- The following should be function arguments (in parentheses, suggested default arguments, not because they are realistic now, but just to speed up the process): number of genes (1000), "k" in k-fold cross-validation (10), number of genes that are selected to be used in the classifier (10), number of times the whole process is repeated (10). Number of subjects can be set to a number (50), or you can pass the sizes of each of the two classes (e.g., 25 and 25); if number of subjects, what would you do with odd numbers?
- The above are default arguments. But enlightenment will come easily if you play around with those: is selection bias more severe the more genes you select? The more genes you can select from? What about number of subjects? Etc, etc.
- You will want textual output and figures, of course.

A basic piece of code that does most of the above takes less than 70 lines.

21 Session info

```
sessionInfo()

## R version 3.3.1 Patched (2016-07-13 r70907)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Debian GNU/Linux stretch/sid
##
```

```
## locale:
##  [1] LC_CTYPE=en_GB.utf8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.utf8        LC_COLLATE=en_GB.utf8
##  [5] LC_MONETARY=en_GB.utf8    LC_MESSAGES=en_GB.utf8
##  [7] LC_PAPER=en_GB.utf8       LC_NAME=C
##  [9] LC_ADDRESS=C              LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.utf8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] tools     stats     graphics  grDevices utils
## [6] datasets  base
##
## other attached packages:
## [1] limma_3.29.17      car_2.1-3          patchSynctex_0.1-3
## [4] stringr_1.0.0      knitr_1.14
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.6         lattice_0.20-33
##  [3] MASS_7.3-45         grid_3.3.1
##  [5] MatrixModels_0.4-1 nlme_3.1-128
##  [7] formatR_1.4         magrittr_1.5
##  [9] evaluate_0.9        highr_0.6
## [11] stringi_1.1.1       SparseM_1.7
## [13] minqa_1.2.4         nloptr_1.0.4
## [15] Matrix_1.2-6        splines_3.3.1
## [17] lme4_1.1-12         parallel_3.3.1
## [19] pbkrtest_0.4-6      mgcv_1.8-13
## [21] nnet_7.3-12         quantreg_5.26
## [23] methods_3.3.1
```