Szymon M. Drobniak

Introduction to R

Materials from the R workshop in the Institute of Environmental Sciences, Jagiellonian Univeristy

Kraków 2011

Prepared in the Group of Population Ecology
Institute of Environmental Sciences
Jagiellonian University, Kraków

This publication is financially supported by
the FRISC project (Factors of Population
Extinction Risk)

Project number GOL PL 0419

Cover project: Szymon Drobniak

Technical processing:
ART-TEKST
Publisher and editor:
ART-TEKST – dr inż. Mariusz Sierpień

ISBN 978-83-7783-004-8

Agencja
Wydawniczo-Poligraficzna
"ART-TEKST"
dr inż. Mariusz Sierpień
"ART-TEKST" 31-866 Kraków Osiedle Akademickie 1/7 tel.: 012 358 84 48 tel. kom.: 0501 534 061 e-mail: art-tekst@ceti.pl

Table of contents

Preface

In recent years biological and ecological applications of statistics reached a new level, both in the sense of sophistication of used techniques and of the theoretical knowledge of biologists. Software providers compete to improve their products and provide biologists with the most recent and advanced solutions. In spite of this fierce competition – there is just a handful of software packages worth consideration at the moment: SAS, SPSS, ASReml, S-PLUS. And last but not least – R. They're all good, with their drawbacks and advantages – and they all have fans all over the world. However one simple factor – price – divides them into two major groups: R and everything else. If you will decide to work with R you'll see that there are more division points – and that the choice of R is more than obvious. I hope that this volume will be of use to all that want to start their adventure with R and that they will discover its diverse applications and beautiful simplicity.

What should not be expected from this book? First of all – it's not a handbook. If you are looking for a complete description of all R's features and utilities you should use more specific literature, such as excellent Crawley's "The R Book". Even with regard to subjects covered here, this book is far from being a step-by-step guide through the R's environment. It is rather a record of two workshops on R that were held in the Institute of Environmental Sciences of the Jagiellonian University (the basic part) and in the Evolutionary Biology Centre of the Uppsala University (the advanced part). That is also the reason for which it's written in English – both workshops were attended by people of different nationalities. One can think of it as a cook-book providing the necessary procedures with some statistical commentary, but without unnecessary details. Also, because of the limited space, only some issues were covered, bearing in mind the needs and requirements of most ecologists that begin their adventure with R.

I hope you will find this book useful, no matter if you're just starting with R or have already done something remarkable with it. I am also aware that mistakes and errors are inevitable, even in the best written handbook. If you come across any inconsistencies and errors – please let me know so that I could update the on-line erratum. All suggestions about the scope and usefulness of this text are also welcome!

Acknowledgments

I would like to thank all that helped to complete this text. Special thanks to all that attended my workshops – your questions were always stimulating and helped to

improve this book. I'm also thankful to Jarrod Hadfield who let me adapt some parts of his "Course Notes [onMCMCglmm]" in this text and to Michael Crawley who let me modify some of the datasets from his "The R Book". I also adapted several datasets available on the Internet for the purposes of this book – you'll find the list of relevant web-pages in the References section.

This publication was supported by the FRISC project, supported by the Financial Mechanism of the European Economic Area (you'll find more about FRISC here: http://www.eko.uj.edu.pl/frisc).

Technical note

In order to be able to follow exercises from this book, you will need the necessary files (data sets etc.) Everything that's required, together with the PDF version of the book, can be downloaded from the following website: http://www.eko.uj.edu.pl/drobniak/r.htm. Additionally, you can also download complete code for the Advanced part in the form of R-script files. Currently (March 2011) it is not available for the Basic part, but it will become available as soon as the coding into the R-script files has been completed.

Basic Issues

Szymon Drobniak

Part 1 ~ Getting started

During this course you will learn how to use one of the most versatile and powerful statistical packages currently available – the R. The course will cover basics of R, essentials of its syntax and the most important mathematical and statistical functions. You will also learn how to create elegant graphs to illustrate your analyses. More advanced statistical issues will also be covered, including generalized linear and mixed models, multivariate statistics, phylogenetic analyses and they can be coupled with linear modeling. Finally, there will also be a short introduction to quantitative genetic analysis and other, more specific uses of statistics.

As R was derived from a well-established language S, it allows not only for statistical calculations. You could also use it as a mathematical modeling language, just in the same way as you would do with Matlab. Compared to Matlab, ASREML or S – R has one great advantage: it's completely free and open-source. It means that you don't have to pay for using it, you don't even have to pay for scientific or commercial use of results obtained using R. Open source also means that anyone can access the source code of R or any R's package. Thank to this there's a great chance that someone already had the same problems you may have and looked through the source code – or wrote new package from a scratch – to solve this problem. In other words – usually you won't have to laboriously invent your own way through your analyses; you'll just need to find the right package among thousands of available, ready to use packages available through the CRAN website. So, let's begin!

Notation

Throughout this book I will be using standard notation for all code blocks. While working with R, one have to remember about several simple rules:

- lower and upper-case letters are distinguished so **`function`** has different meaning than **`FunCTioN`**;
- there are no restricted keywords in R – but there are several names that are recommended to be used only in their default, built-in meaning; for good reasons avoid using the following words for your user-defined names: **`mean`**, **`sd`**, **`fix`**, **`random`**, **`units`**, **`residuals`**, **`animal`**, **`data`**, **`vector`**, **`factor`**, **`list`**, **`for`**, **`if`**, **`else`**, **`function`**, **`var`**; you'll see some of them (eg. data) used later together with some additional letters, but never alone;

- when you do a lot of analyses in R it's easy to get lost among dozens of objects and names – try inventing your own strategy of naming things in R and follow it strictly; it will spare you many stressful disappointments;
- Within the block codes I will use some additional formatting to make reading the code easier; names of the functions will be in **`bold`**, arguments' names within functions will be `underlined`, and everything else will be in `non-bold, non-italic font`; outputs presented within block codes are in *`italics`* and using smaller font; I will use `Courier` as the font for block codes – it will prevent some tabular outputs from being incorrectly displayed
- In R you can break lines to everything without the need for scrolling the window horizontally; in block codes, most code is broken into lines that fit the width of the printed page, but of course you can use your own line-breaking or nit use it at all;
- R displays so called prompt: **>** at the beginning of every line; it's also present in code blocks but remember to remove it (and also the broken line prompt: **+**) if you want to copy-and-paste code lines directly from the PDF version of this book;
- quotation marks denote text variables in R; you can use either single or double quotation marks interchangeably, just remember not to mix them;
- sometimes I use the **#** sign inside the block code to provide additional comments; lines beginning with the hash sign are not executed and you should obviously omit them executing the code, but most often they provide important notes on the code and should be read as integral parts of the text; for ease of reading I use shaded background in comments.

R can be used in all most popular operating systems. As it is Unix-derived it is recommended to be used in Linux systems. Currently one of UNIX systems – Debian-based **Ubuntu** – is very user friendly and it can be recommended for using with R. Advantages of using Ubuntu are several: you're sure that no problems with file compatibility will occur; importing your data is much more straightforward; as R is run in basic Linux console – it operates much quicker (in Windows after running some complex analysis system for several seconds looks like crashed) and you can run 2-3 instances of R simultaneously, which in Windows would cause system crash (provided you would be able to do this in Windows in the first place!). However, availability of Windows and it's general popularity decided that most people use R in Windows, despite it's obvious limitations and drawbacks. Here we will also use Windows environment, but you can easily 'translate' all commands and procedures to Linux- or MacOS-based R installations simply by translating all source paths into system-specific syntax.

R is not very resource-demanding and can be easily run even on older machines – but beware! The more complex analysis you should try to run, the more memory- and processor-demanding it would be. Quickly you'll realize that trying to analyse thousands of records with multiple responses and dozens of explanatory variables will be difficult using older PCs. If you plan to use R for elaborate and computer-intensive applications – make sure your computer is well suited for this (at least 2 GHz processor and 1 GB of RAM). Otherwise it will be a rather disappointing experience...

Downloading and installing R

The software can be downloaded from the CRAN (The Comprehensive R Archive Network) webpage: **http://cran.R-project.org**. The package is downloaded as an ordinary binary file (.exe) that can be executed directly; it automatically installs R on your computer. In UNIX based systems you need to follow the usual way you use to install any other software; e.g. in Ubuntu you have to use Package Manager to find R in the repository and then install it on your system. After installation R is available for running as any other program.

Basic distribution of R contains the most essential and widely used packages (e.g. basic tests, procedures for mathematical calculations, graphing algorithms, generalized linear modelling tools). If you'd like to expand functionality of R (and most likely you will eventually have to do this) – you have to use once again the CRAN service – it contains most of available packages, together with their documentations, manuals, etc. Installing them in your R distribution is easy. Let's assume you'd like to have the **ape** package (used for creating and handling phylogenetic data):

```
> install.package("ape")
```

After that this package is installed, but still not loaded. To use it just type

```
> library("ape") #you can omit quoting marks here
```

Now this package is loaded and ready to use. Loading external (non-standard) packages has to be done every time you start up R. To check which packages are loaded, just use

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:ape"
```

How can we know what package we need? Well, first of all – ask others. It's very likely that someone already had similar problem as you and found out how to solve it. Secondly, try using **www.rseek.org** – it's a google based search engine

designed specifically for R users. If you already have a package or function and just want to know what it does – just use R's help commands:

```
> help(seq)
> ?seq
```

If you're not sure what you're looking for – use global help search. R will search everything it has on the local machine and return most useful results. Here we're looking for any idea how to do Spearman correlation test. R suggests the **cor.test** package, which turns out to be of use:

```
> ??spearman
> ?cor.test
```

Using R. The R console. The workspace

When you run R, you'll see white console with some information at the top (such as current version, some basic info how to get help in R, how to cite your use of R, etc.) Under this you'll see the command prompt:

```
>
```

It marks the place where you can enter your commands. E.g., if we enter:

```
> (3+4)*12
```

we'll get

```
[1] 84
```

In other words, R works as command-line software and you cannot use menus and other Windows-like graphic-user-interface features to operate it. You simply write the command, hit Enter – and you get your result (or error message if something is wrong). R is also an interpreted language – you simply write your commands one by one and execute them in real time; it's in clear opposition to compilator-based programming languages (such as C or Java) where you have to write the whole program, and then compile it to make it usable.

It is also essential to realize, that in R everything has its place in the computer's memory (just like in any other programming language). You can name these memory places and assign values to them:

```
> y <- (3+4)*12
```

Now, the place called **y** is a variable, that contains the value of our arithmetical operation. The operator '**<-**' assigns the value on the right-hand side to the variable

on the left-hand side (however, you can use the reverse version). You can call this value by typing the name of the variable:

```
> y
[1] 84
```

There are several rules for effective use of R environment:

- When you want to provide a name (e.g. package name, file name, anything that is not a variable or factor name) – always do this by using quotation marks (" "or ' '). Single apostrophes are 'stronger' – double ones should be nested inside single ones if necessary. Always remember to close quotes once opened, and don't mix different types of quotation marks.
- If you want to find any previously executed command – just use up and down arrows on your keyboard; it allows you to browse through executed commands; if you'd like to stop browsing just hit Esc – it will clear the prompt. Browsing through previous commands is especially useful when you enter a wrong command; correcting it requires just hitting 'up' (it returns last entered command), then you can just correct any spelling errors and hit Enter once again.
- The Tab key can be used to fill any started command, provided R knows the name or command (i.e. the name exists in its environment or required package has bee loaded). Try typing '**cita**' and hitting Tab – R should complete command: '**citation**'; you can than end the command by using brackets. This function returns the proper way of citing R in papers.

```
> cita #hit Tab
> citation
> citation()

To cite R in publications use:

  R Development Core Team (2009). R: A language and environment for
statistical computing. R Foundation for Statistical
  Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-
project.org.

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {R: A Language and Environment for Statistical Computing},
    author = {{R Development Core Team}},
    organization = {R Foundation for Statistical Computing},
```

```
    address = {Vienna, Austria},
    year = {2009},
    note = {{ISBN} 3-900051-07-0},
    url = {http://www.R-project.org},
  }

We have invested a lot of time and effort in creating R, please cite it when
using it for data analysis. See also
'citation("pkgname")' for citing R packages.
```

- If you reach end of the line and don't want the window space to 'shift' following your typing you can break the line by using Enter: just hit Enter after unfinished line (suggesting that something should be added, e.g. further parts of operation, closing brackets, etc.) Broken line is continued below, after leading '+' (instead of '>'; remember that this plus sign does not imply any sum!)

```
> 2+3+4+5*7/9-
+ 2
[1] 10.88889
```

- Two or more commands can be executed at once – just enter one after another, separating them by semicolons:

```
> 4-7; 76/33; "Hellow World"; log(1000)
[1] -3
[1] 2.303030
[1] "Hellow World"
[1] 6.907755
```

- Any spaces are ignored – you can type as many as you want; in general you should use spaces to make your code easier to read – quickly you'll realize it pays off! Also, for clarity try to use informative names.

```
> ff34<-c(3,4,5,3,4,6,3,5,3,5,6,3)
> seomd<-sd(ff34)/sqrt(12)

> data <- c(3,4,5,3,4,6,3,5,3,5,6,3)
> std.error <- sd(data)/sqrt(12) #isn't that clearer?
```

Everything you type or create during the R session is contained in so called workspace, a virtual place in the memory of your machine. Once you close R this

workspace disappears along with everything you've created. To prevent this from happening you might want to save the workspace. In fact, it is a good habit to save the workspace for any particular analysis and load it again later when you go on with you calculations. To save the workspace you'll need to have your working directory, which is a folder on your computer that will contain any saved workspaces. Setting a working directory is simple: first, just create new empty directory on your disk drive (it's better to do this somewhere on top of directory tree to shorten the source path for it).Then type:

```
> setwd("D:/workingdir")
> getwd()
[1] "D:/workingdir"
```

First command sets your working directory (you may choose any valid name for it), second one returns its address. Now, any time you quit R (by typing **`q()`**) – you'll be asked if you want to save your workspace. By clicking Yes, you'll create two files in your working directory: **`.RData`** and **`.Rhistory`**. The first one contains any objects you've created (e.g. **`y`**), the second one – all commands executed. After restarting R it will resume last saved workspace. Should you want to restore any other custom workspace just navigate to the proper working directory (using **`setwd(...)`**) and use below commands to restore objects, commands or both:

```
> load(file=".Rdata")
> loadhistory(file=".Rhistory")
```

If you want, you can remove any object from your current workspace. Just type:

```
> rm(y)

> y
Error: object 'y' not found
```

If you don't remember all the names you've created – just use the **`ls()`** function. It displays all objects in the current workspace (UNIX users should know see that R comes from UNIX). For times of despair you have the following code:

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

but be careful as there will be no warning and all your objects will be irreversibly deleted.

Finally, you don't have to wait until the end of your R session to save your workspace. You can use the **save(file=".Rdata")** and **savehistory(file=".Rhistory")** functions – they will save objects and commands in the current working directory, using the names you provide.

Functions

Everything you use in R to manipulate objects is a function. It means, that everything has this form: **function.name(arguments): { actions }**. In other words, you'll always find a name you can use to call a function. After being called, this function takes arguments you provide, doeas something with them (actions) and returns the result. E.g. you've already seen the function **c()**. It takes as many values (arguments) as you provide and concatenates them creating a vector:

```
> myvector <- c(1,2,3,4,5,6,7,8)
> myvector
[1] 1 2 3 4 5 6 7 8
```

Of course, functions may do more sophisticated things. E.g. if you need a sequence of numbers generated with a specified interval, you should use the **seq()** function. Let's check what it does?

```
> sample <- seq(1,10,length.out=25)
> sample #sequence of 25 numbers
 [1]  1.000  1.375  1.750  2.125  2.500  2.875  3.250
 [8]  3.625  4.000  4.375  4.750  5.125  5.500  5.875
[15]  6.250  6.625  7.000  7.375  7.750  8.125  8.500
[22]  8.875  9.250  9.625 10.000
```

If you'd like to use function that does not exist – that's not a problem! Just write one. Let it be the method for calculating arithmetic mean; let's decide its name is **mean**:

```
> mean <- function(x) {sum(x)/length(x)}
> mean(sample)
[1] 5.5
```

Importantly, when any function is called – anything that happens through this function, happens not in the R global environment, but within this function. As soon as functions finishes its work, any objects and variables required are destroyed as they exist only in the local place, inside this function. In other words – if we had a global

object called x it won't be altered by the fact that our function mean also uses object called x to assign the data to it; this second object exists only inside this function:

```
> x <- c(2,2,2)

> add10 <- function(x) { #we define function that adds 10 to a number
+   x <- x + 10
+   x
+ }

> add10(x)
[1] 12 12 12

> x #should remain unaltered
[1] 2 2 2
```

Part 2 ~ Data structures

Vectors

Vector is the most basic data structure in R. Simply, vector is a collection of variables, e.g. numbers, text variable, etc. Creating vectors is achieved by **`c()`** function. As you noticed before, any output in R begins with **`[1]`**. This is an index showing the position of the element within a vector. In general – you can call any particular element of an vector by giving its index number:

```
> a <- c('a','b','c','d','e')
> a[3]
[1] "c"
```

In R there are no scalars – everything is a vector (or a more complex structure); even single numbers are simply an one-element vectors. The most wonderful thing about vectors is that they largely eliminate the need for iterated operations as known in other languages. If we want to apply some function to every element of an vector, we simply enter this vector as an argument.

```
> myvector <-  seq(1, 10, by=0.76)
> myvector
 [1] 1.00 1.76 2.52 3.28 4.04 4.80 5.56 6.32 7.08 7.84
[11] 8.60 9.36

> sqrt(myvector) #square root
 [1] 1.000000 1.326650 1.587451 1.811077 2.009975
 [6] 2.190890 2.357965 2.513961 2.660827 2.800000
[11] 2.932576 3.059412

> othervector <- c(1,2,3,4)
> myvector + othervector
 [1]  2.00  3.76  5.52  7.28  5.04  6.80  8.56 10.32
 [9]  8.08  9.84 11.60 13.36
```

Second example (sum of two vectors) shows one important feature of using vectors as arguments: if in a function one vector has fewer elements than needed – function uses its elements again from the beginning until it terminates. It's called vector recycling – and you should remember it happens, especially when you forget to care about the length of vectors used as arguments.

Vectors can be handled using many built-in functions. Sometimes their application is quite straightforward and intuitive:

```
> vector1 <- c(1,2,3,4,5,6,7)
> vector2 <- c(2,2,3,3,4,4,5)

> max(vector1) # maximum value
[1] 7


> min(vector2) # minimum value
[1] 2

> sum(vector1) # sum of elements
[1] 28

> mean(vector1)
[1] 4

> median(vector1)
[1] 4

> var(vector1) # variance
[1] 4.666667

> cor(vector1,vector2) # correlation between two vectors
[1] 0.9707253

> sort(vector1) # sorted version
[1] 1 2 3 4 5 6 7

> rev(sort(vector1)) # reversed-sorted vector
[1] 7 6 5 4 3 2 1

> rank(vector2) # ranks of vector elements
[1] 1.5 1.5 3.5 3.5 5.5 5.5 7.0

> quantile(vector2) # minimum, lower quantile, median,
                    # upper q. and max
  0%  25%  50%  75% 100%
 2.0  2.5  3.0  4.0  5.0

> pmax(vector1,vector2) # for every position - maximum value
                        # from all supplied vectors
[1] 2 2 3 4 5 6 7

> cumprod(vector1) # for each element - cumulative product
[1]    1    2    6   24  120  720 5040

> length(vector1)
[1] 7
```

Arithmetic is not the only thing we can do with vectors. We can also apply logical operations – which will be further very useful while handling and filtering our own data. We could want to square root only even numbers in our vector. This can be achieved by the operator of modulo, which either returns an integer part of the division or the reminder:

```
> 23%/%6 #integer division
[1] 3
> 23%%6 #reminder
[1] 5
```

By checking the remainder after diving by 2 we can check if the number is odd (remainder 1) or even (remainder 0):

```
> 6%%2 == 0 # even
[1] TRUE
> 7%%2 == 0 # odd
[1] FALSE
```

Now – let's square-root even number within the range between 0 and 50:

```
> samp <- 0:50
> sqrt(samp[samp%%2 == 0])
 [1] 0.000000 1.414214 2.000000 2.449490 2.828427
 [6] 3.162278 3.464102 3.741657 4.000000 4.242641
[11] 4.472136 4.690416 4.898979 5.099020 5.291503
[16] 5.477226 5.656854 5.830952 6.000000 6.164414
[21] 6.324555 6.480741 6.633250 6.782330 6.928203
[26] 7.071068
```

Now let's translate. First we generated numbers from 0 to 50. Then we checked which of them are even by using modulo:

```
> samp%%2
 [1] 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
[26] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
[51] 0


> samp%%2 == 0
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
 [9]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[17]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[25]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[33]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[41]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
[49]  TRUE FALSE  TRUE
```

Finally we used these logical numbers as indexes to choose only even values (those returning TRUE in our logical test) and square-rooted them:

```
> samp[samp%%2 == 0]
 [1]  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30
[17] 32 34 36 38 40 42 44 46 48 50


> sqrt(samp[samp%%2 == 0])
 [1] 0.000000 1.414214 2.000000 2.449490 2.828427
 [6] 3.162278 3.464102 3.741657 4.000000 4.242641
[11] 4.472136 4.690416 4.898979 5.099020 5.291503
[16] 5.477226 5.656854 5.830952 6.000000 6.164414
[21] 6.324555 6.480741 6.633250 6.782330 6.928203
[26] 7.071068
```

Sometimes using logical operations may be simpler by generic functions, e.g.:

```
> which(samp%%2 == 0)
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31
[17] 33 35 37 39 41 43 45 47 49 51
```

Here we asked simply which elements are even and got the list of indexes for such elements. Of course, in the above functions we might as well use other logical operators; the following are available: **>**, **<**, **==**, **<=**, **>=**, **!=** (not equal to), **|** (logical OR, returns TRUE if either of expressions is TRUE), **&** (logical AND, returns TRUE if both expressions are TRUE).

Other vector functions

There are several useful functions operating on vectors or returning vectors. You've already seen **seq()**:

```
> seq(10, 30, length.out=8)
[1] 10.00000 12.85714 15.71429 18.57143 21.42857
[6] 24.28571 27.14286 30.00000
```

Instead of using output length we could specify the interval (argument by) – which can also be negative (numbers will be in descending order):

```
> seq(30, 10, by=-8)
[1] 30 22 14
```

The function **sample()** can be used to shuffle elements of a vector and generate random samples, either with or without replacement:

```
> sample(vector1)
[1] 6 5 1 4 3 2 7
> sample(vector1,10)
Error in sample(vector1, 10) :
  cannot take a sample larger than the population when 'replace = FALSE'
> sample(vector1,10,replace = T)
 [1] 2 1 1 5 2 5 6 2 2 2
```

Another extremely useful function working on vectors is function for looking for runs of numbers in vectors. Let's generate a vector of 20 random numbers from Poisson distribution and look for runs of numbers in it:

```
> poiss <- rpois(20,0.7)
> poiss
 [1] 0 0 0 1 1 0 0 1 1 1 0 0 0 1 1 2 2 0 1 1
> rle(poiss)
Run Length Encoding
  lengths: int [1:9] 3 2 2 3 3 2 2 1 2
  values : num [1:9] 0 1 0 1 0 1 2 0 1
```

You can see that the longest runs are of three digits and one of them consists of zeros, and the second one of ones.

Finally we can use **rep()** to generate repeats of numbers of different kinds. I'll leave exploring it's usability to you – try **?rep.**

Generic functions

R offers several so-called generic functions. They work with nearly all types of R objects and return outputs that are specific to handled objects. You can check respective summary functions, e.g. **summary.lmer()** is a summary functions designed for **lmer()** output objects. However, the simpler way is to just use **summary(object)**

– in such a case the function related to the type of an "object" will be called. The most important generic functions are as follows: **summary()**, **mode()**, **attributes()**, **class()**. How can we use them?

The **summary()** function return the descriptive summary of an object, e.g. descriptive statistics for data objects, analysis results for model objects or just the list of object's internal attributes. As mentioned before, every R object has its own specific summary function, e.g. for a vector:

```
> vector1 <- sample(1:100, size = 20)
> summary(vector1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00   15.25   52.50   47.15   73.00   97.00
```

The next three functions provide information about the type of information stored in an object and about its attributes. Their usefulness will become clearer when additional data types will be introduce. In case of vectors the only useful is the **mode()** function – it returns the type of stored data (numbers, text, etc.) The **class()** function would return exactly the same, whereas the **attributes()** function will return **NULL**: as the most basic data structures in R, vectors don't have any attributes:

```
> vector2 <- c('A','B','C')
> mode(vector1); mode(vector2)
[1] "numeric"
[1] "character"
```

The most obvious way of using these functions is the logical test to check if our data are of the required type:

```
> if(mode(vector2) == "character") "All OK!"
[1] "All OK!"
```

Other data structures in R

Vectors are the simplest data-types in R – and probably the most widely used. However we should know more complex data structures as they will allow for more sophisticated and less cumbersome manipulation of numbers.

Arrays are used probably as often as vectors. They can be regarded as generalisation of vectors to cases of more than one dimension. Thus, there can be 2-dimensional arrays (a "grid" of cells), 3-dimensional arrays (a "cube" of cells), 4-dimensional tables, and so on. Probably somewhere around the 4^{th} or 5^{th} dimension

our imagination will start failing. Luckily – in R arrays are not displayed in the form of monstrous pseudo-3D things – R uses much simpler, flat representation. All information about the number of arrays' dimensions are stored in their attributes and we can extract them using **`attributes()`**:

```
> months <- c("Jan","Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
+ "Sep", "Oct", "Nov", "Dec")

> array(months, c(3,4)) #2D array
#we use a vector [3,4] to specify dimensions
     [,1]  [,2]  [,3]  [,4]
[1,] "Jan" "Apr" "Jul" "Oct"
[2,] "Feb" "May" "Aug" "Nov"
[3,] "Mar" "Jun" "Sep" "Dec"

> array(months, c(2,6))
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] "Jan" "Mar" "May" "Jul" "Sep" "Nov"
[2,] "Feb" "Apr" "Jun" "Aug" "Oct" "Dec"

> array(months, c(3,2,2)) #3D array
, , 1

     [,1]  [,2]
[1,] "Jan" "Apr"
[2,] "Feb" "May"
[3,] "Mar" "Jun"

, , 2

     [,1]  [,2]
[1,] "Jul" "Oct"
[2,] "Aug" "Nov"
[3,] "Sep" "Dec"

> attributes(array(months, c(4,3)))
$dim
[1] 4 3

> class(array(months, c(4,3)))
[1] "matrix"

> class(array(months, c(2,2,3)))
[1] "array"

> matrix(months, c(2,6))
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] "Jan" "Mar" "May" "Jul" "Sep" "Nov"
[2,] "Feb" "Apr" "Jun" "Aug" "Oct" "Dec"
```

As you could see – a 2-dimensional array is a special data structure in R and is called a matrix, which should not be surprising. After all – matrices are very common

in data analysis. Instead of using **array()** we can use respective **matrix()** function. Both data structures are similar and contain the **$dim** attribute.

```
> attributes(array(months, c(2,2,3)))


$dim
[1] 2 2 3
```

When working with arrays (and matrices) one should remember about some important rules. Every time we call an object within an array, we do it by specifying the row and column numbers, row number first. There's one confusing consequence of this rule – if we look at any of the matrices above, it's apparent that columns are filled from top to bottom. It might seem strange but once we realize that R iterates through rows first, it becomes clear: the filling of a matrix proceeds row by row (as rows are called first) and then – having reached the bottom of a column – R jumps to the next column.

Technically, calling an element from an array means writing it's indexes (as many as there are dimensions) in square brackets, in the order explained above, separated by commas. We may as well use single index – in such a case the whole array will be treated as a single vector (again, with the order of elements determined in a way already described).

```
> mon.arr <- array(months, c(2,2,3))
> mon.arr[2,1,2]
[1] "Jun"
> mon.arr[5]
[1] "May"
```

If we omit any of the dimensions retaining commas – the result will be the whole dimension (the one for which there's no index) extracted:

```
> mon.arr[2,1,]
[1] "Feb" "Jun" "Oct"
> mon.arr[2,,2]
[1] "Jun" "Aug"
> mon.arr
, , 1

     [,1]  [,2]
[1,] "Jan" "Mar"
[2,] "Feb" "Apr"
```

```
, , 2

     [,1]  [,2]
[1,] "May" "Jul"
[2,] "Jun" "Aug"

, , 3

     [,1]  [,2]
[1,] "Sep" "Nov"
[2,] "Oct" "Dec"
```

There are many functions that work only with matrices (and not with arrays), such as: **t()** – which transposes a matrix, "reflecting" it relatively to its diagonal; **ncol()** and **nrow()** – which returns the number of rows or columns.

Another important data structure is a list. Sometimes it's more useful than the vector as it may store objects of different types. Try building a vector made of text, numbers and logical values. Is it possible?

```
> mix.vec <- c(1,2,'a','nie chce mi sie')
> mix.vec
[1] "1"    "2"    "a"    "nie chce mi sie"
```

No! R automatically converts numbers to ASCII characters. We have to use lists in order to be able to store these values unaltered:

```
> mix.list <- list(1,2,'a','nie chce mi sie')
> mix.list
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] "a"

[[4]]
[1] "nie chce mi sie"
```

R puts our variables into the list (note lack of quoting marks around numbers, i.e. they really are numbers). All elements are also given additional indexes within a list, indicated by double square brackets. Each of these sub-elements is in fact a vector. Consequently, if one wish to call an element inside a list, they should use double square brackets around the indexes. It may lead to somehow complex indexing

structures as nesting one list inside the other creates the need for double list indexes (i.e. doubled double square brackets):

```
> mix.list <- list(1,2,'a','nie chce mi sie',
+ c(33,44),list('nested'))
> mix.list
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] "a"

[[4]]
[1] "nie chce mi sie"

[[5]]
[1] 33 44

[[6]]
[[6]][[1]]
[1] "nested"


> mix.list[[5]]
[1] 33 44

> mix.list[[5]][2]
[1] 44


> mix.list[[6]]
[[1]]
[1] "nested"

> mix.list[[6]][[1]]
[1] "nested"
```

Another useful feature of vectors and matrices is the possibility of naming their elements/rows/columns. Having done so we can use these names instead of indexes when calling object's elements:

```
> sunny.days.krakow <- c(10,9,13,14,23,20,29,27,26,18,5,3)
> names(sunny.days.krakow) <- months

> sunny.days.krakow
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
 10   9  13  14  23  20  29  27  26  18   5   3

> sunny.days.krakow["Feb"]
Feb
  9
```

```
> mon.matrix <- matrix(months, c(4,3))
> colnames(mon.matrix) <- c("1st","2nd","3rd")
> rownames(mon.matrix) <- c("top","middle1","middle2","bottom")

> mon.matrix
        1st   2nd   3rd
top     "Jan" "May" "Sep"
middle1 "Feb" "Jun" "Oct"
middle2 "Mar" "Jul" "Nov"
bottom  "Apr" "Aug" "Dec"
```

When the elements, rows or columns have been names, an object gets the **$names** attribute, which contains them:

```
> attributes(sunny.days.krakow)
$names
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

Arrays and lists are not the only complex data types in R. Another important structures are factors. They are used to define categorical variables – such as used in conventional ANOVA. In factors, numbers are not interpreted as numbers but rather as factor levels (categories). Below we use employ factors to define the cloud cover during subsequent hours of the day. Cloud cover is measured using a 8-degrees scale and we might be interested in counting occurrences of a particular degree within our dataset rather than in treating these measurements as continuous. In other words – we might prefer expressing cloud cover in nominal scale, hence – as factors. The **summary()** function called for a factor object returns counts in respective levels instead of descriptive statistics known for continuous variables:

```
> cloud.cover <- c(8, 7, 4, 8, 1, 1)
> names(cloud.cover) <- c(0400, 0800, 1200, 1600, 2000, 0000)
> cloud.cover
 400  800 1200 1600 2000    0
   8    7    4    8    1    1
> summary(cloud.cover)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.750   5.500   4.833   7.750   8.000
> cloud.cover <- factor(cloud.cover)
> summary(cloud.cover)
1 4 7 8
2 1 1 2
> levels(cloud.cover) #may be used for assigning new level names
[1] "1" "4" "7" "8"
```

Class-control functions

R provides several functions for controlling the class of objects. We've seen some of them already – these were functions used for generating respective objects: **`matrix()`**, **`list()`**, **`factor()`**, **`array()`**, **`data.frame()`** (more on data-frames in the next section). Their arguments are usually single values or vectors of values, and sometimes additional parameters modifying the way these functions work. However – their output is "permanent" – a new instance of the respective data objects is created. However, it might as well be useful to *treat* an object as if it was of other type, just for the purposes of the current procedure. To achieve this we can use as-functions:

```
> vector <- c(1,1,2,2,1,3,4)
> as.factor(vector)
[1] 1 1 2 2 1 3 4
Levels: 1 2 3 4 5

> as.matrix(vector)
     [,1]
[1,]    1
[2,]    1
[3,]    2
[4,]    2
[5,]    1
[6,]    3
[7,]    4

> as.data.frame(vector)
  vector
1      1
2      1
3      2
4      2
5      1
6      3
7      4
```

Another important group of functions are is-functions, which allow for testing the type of an object:

```
> is.vector(vector)
[1] TRUE
> is.factor(vector)
[1] FALSE
```

Data-frames

We have reached probably the most essential section on data types in R and that would be the data-frame. They're usually used to store data obtained in regular research. Each data-frame contains several columns called variables, each reflecting

one particular quantity measured. Variables have names, stored in the **$names** attribute. Rows (or subsequent cases/data points) also may be named (**$row.names**), but by default they are numbered consecutively.

Let us enter simple data-frame; it contains data for silk production by silkworms (current production), together with respective variables describing the sex of individuals, number of days it had been producing silk, humidity of the environment, silk production of their parents (previous production). Hence, rows of the table represent individual caterpillars.

```
> silk.production <- data.frame(
+ 'sex' = c(1,2,2,1,1,1,1,2,2,2,1,2,1,2,1),
+'no.days' = c(10,12,8,21,20,13,18,19,20,20,10,7,9,12,12),
+ 'humid' = c(100,100,90,89,88,78,79,90,88,90,87,
+ 90,76,56,78),
+ 'prev.prod'=c(100,340,546,234,432,765,432,318,287,
+ 190,223,210,218,220,431),
+ 'curr.prod'=c(34,45,38,76,54,28,56,98,67,32,22,10,23,43,8))

> silk.production
   sex no.days humid prev.p curr.p
1    1      10   100    100     34
2    2      12   100    340     45
3    2       8    90    546     38
4    1      21    89    234     76
5    1      20    88    432     54
6    1      13    78    765     28
7    1      18    79    432     56
8    2      19    90    318     98
9    2      20    88    287     67
10   2      20    90    190     32
11   1      10    87    223     22
12   2       7    90    210     10
13   1       9    76    218     23
14   2      12    56    220     43
15   1      12    78    431      8

> summary(silk.production)
      sex           no.days          humid            prev.p
 Min.   :1.000   Min.   : 7.00   Min.   : 56.00   Min.   :100.0
 1st Qu.:1.000   1st Qu.:10.00   1st Qu.: 78.50   1st Qu.:219.0
 Median :1.000   Median :12.00   Median : 88.00   Median :287.0
 Mean   :1.467   Mean   :14.07   Mean   : 85.27   Mean   :329.7
 3rd Qu.:2.000   3rd Qu.:19.50   3rd Qu.: 90.00   3rd Qu.:431.5
 Max.   :2.000   Max.   :21.00   Max.   :100.00   Max.   :765.0

     curr.p
 Min.   : 8.00
 1st Qu.:25.50
 Median :38.00
 Mean   :42.27
 3rd Qu.:55.00
 Max.   :98.00
```

It's obvious that the sex variable should be categorical rather than continuous. We can access this variable either by selecting the first column or by using the variable's name. The rule here is the same as in the case of matrices – first we indicate the row, then the column. Omitting one index means that we want to extract the whole row/column.

```
> silk.production[10,1]
[1] 2
> silk.production[,1]
 [1] 1 2 2 1 1 1 1 2 2 2 1 2 1 2 1
> silk.production$sex
 [1] 1 2 2 1 1 1 1 2 2 2 1 2 1 2 1
```

Zmieńmy zatem typ danych w kolumnie z płcią:

```
> silk.production$sex <- as.factor(silk.production$sex)
> summary(silk.production)
 sex      no.days          humid            prev.p          curr.p
 1:8   Min.   : 7.00   Min.   : 56.00   Min.   :100.0   Min.   : 8.00
 2:7   1st Qu.:10.00   1st Qu.: 78.50   1st Qu.:219.0   1st Qu.:25.50
       Median :12.00   Median : 88.00   Median :287.0   Median :38.00
       Mean   :14.07   Mean   : 85.27   Mean   :329.7   Mean   :42.27
       3rd Qu.:19.50   3rd Qu.: 90.00   3rd Qu.:431.5   3rd Qu.:55.00
       Max.   :21.00   Max.   :100.00   Max.   :765.0   Max.   :98.00
```

Applying functions and transformations to whole datasets is as easy as working with single vectors. Here we calculate the mean daily production of silk for each individual and add it as a new variable to the data-frame:

```
> average.p <- silk.production$curr.p / silk.production$no.days
> average.p
 [1] 3.4000000 3.7500000 4.7500000 3.6190476 2.7000000 2.1538462
 [7] 3.1111111 5.1578947 3.3500000 1.6000000 2.2000000 1.4285714
[13] 2.5555556 3.5833333 0.6666667


> cbind(silk.production, average.p)
   sex no.days humid prev.p curr.p average.p
1    1      10   100    100     34 3.4000000
2    2      12   100    340     45 3.7500000
3    2       8    90    546     38 4.7500000
4    1      21    89    234     76 3.6190476
5    1      20    88    432     54 2.7000000
6    1      13    78    765     28 2.1538462
7    1      18    79    432     56 3.1111111
8    2      19    90    318     98 5.1578947
9    2      20    88    287     67 3.3500000
10   2      20    90    190     32 1.6000000
11   1      10    87    223     22 2.2000000
```

```
[...clipped...]
```

We can further extend operations on data-frames to joining several columns or rows together. The **cbind()** function joins two vectors column-wisely whereas the **rbind()** joins the vectors row-wisely. Of course, we can use these on whole datasets to add new cases or variables. It's important that the elements of the vector being added contained elements in the right order!

Data-frames can be filtered in the same way as vectors. E.g. we can extract the data on the production of silk only for females (category "2" in sex):

```
> silk.production[silk.production$sex==2,]
   sex no.days humid prev.p curr.p
2    2      12   100    340     45
3    2       8    90    546     38
8    2      19    90    318     98
9    2      20    88    287     67
10   2      20    90    190     32
12   2       7    90    210     10
14   2      12    56    220     43
```

Note the position of the logical condition – the test is applied to rows as we want to have WHOLE rows that contains appropriate value in the second (sex) column.

Indexing can be used to sample our data-frame randomly, e.g. in randomization methods. We can use the **sample()** function here:

```
> silk.production[sample(1:15,5),]
   sex no.days humid prev.p curr.p
12   2       7    90    210     10
5    1      20    88    432     54
8    2      19    90    318     98
13   1       9    76    218     23
2    2      12   100    340     45
```

Sorting data-frames according to particular variables is also simple. Note that the only thing one have to do is to sort row indexes – that's why sorting expression appears in the place of rows:

```
> silk.production[order(silk.production$no.days),]
   sex no.days humid prev.p curr.p
12   2       7    90    210     10
3    2       8    90    546     38
13   1       9    76    218     23
1    1      10   100    100     34
11   1      10    87    223     22
2    2      12   100    340     45
14   2      12    56    220     43
15   1      12    78    431      8
6    1      13    78    765     28
7    1      18    79    432     56
```

```
8    2      19    90    318     98
5    1      20    88    432     54
9    2      20    88    287     67
10   2      20    90    190     32
4    1      21    89    234     76
```

If one want to remove data from a data-frame it can be done by using negative subscripts: this indicates the numbers of rows that should be deleted. The minus sign may also be applied to the whole logical condition – in such a case rows or columns satisfying the condition will be deleted (note that the same could be achieved by filtering the data-frame with the "not equal to" (!=) operator). Here we remove the 3rd and 4th rows and then all records of females:

```
> silk.production[-(3:4),][-which(silk.production$sex==2),]
   sex no.days humid prev.p curr.p
1    1      10   100    100     34
6    1      13    78    765     28
7    1      18    79    432     56
8    2      19    90    318     98
9    2      20    88    287     67
13   1       9    76    218     23
15   1      12    78    431      8
```

Finally, it might be the case that we want to remove all missing data points (records containing NA's). To do this we use **`na.omit()`** function:

```
> missing <- data.frame('var1'=c(1,2,3,4,5,6,NA),
+ 'var2'=c(2,2,2,NA,NA,2,2)) #sample data with NAs
> missing
  var1 var2
1    1    2
2    2    2
3    3    2
4    4   NA
5    5   NA
6    6    2
7   NA    2
> na.omit(missing)
  var1 var2
1    1    2
2    2    2
3    3    2
6    6    2
```

NaN or Not-A-Number

Sometimes R operates on values that are not numbers, such as missing values or division-by-zero outputs:

```
> 2/0
```

```
[1] Inf
```

Here the infinity (the result of dividing by zero) is a non-numerical value. Such values can seriously hamper our work wince many functions return **NaNs** after encountering even single **NaNs** in the set of values. One possible solution for this problem is using the **na.rm** argument, that is provided by several R functions and lets them ignore non-numerical values:

```
> x <- c(2,3,4,5,6,NA)
> var(x,na.rm=T)
[1] 2.5
> var(x)
[1] NA
```

If for some reason one want to check is the value is **NA** (e.g. to filter out such values from a dataset) do not try to use conventional logical test, i.e. **A!=NA** as it won't work. Instead, use the class-controlling function **is.NA(A)**.

Attach() and with() – a digression

You have probably noted that in data-frames we used the usual way of calling variables, that is using the dollar operator. However the need for providing the whole name of the table may be annoying. That's why you might want to use the **attach()** function. It allows for attaching the whole object to the namespace we are currently working in. Names of all variables become available without the need of using the dollar operator and invoking the name of the table. Compare:

```
> silk.production$prev.p
 [1] 100 340 546 234 432 765 432 318 287 190 223 210 218 220 431
> attach(silk.production)
> prev.p
 [1] 100 340 546 234 432 765 432 318 287 190 223 210 218 220 431
```

There is however one problem. Once the table have been attached – the names of all variables are directly available but we can easily overwrite them by creating a new object with the same name. In such a case it is easy to forget that we've attached an object – and use the name as if it was from the data-frame. However now it refers to something entirely different, which may introduce serious bias in our analyses. The reason for that is the location of the attached table in the hierarchy of the namespace. All newly created objects are placed in the global environment and by default they have higher priority over subsequent sections of the namespace. We can actually see this hierarchy after calling the **search()** function:

```
> search()
 [1] ".GlobalEnv"        "silk.production"   "package:sudoku"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
> prev.p <- 'Iam not correct!'
> prev.p #newly created object replaced the one from the table
[1] "Iam not correct!"
```

Fear not! The data in the data frame remain intact. The only thing that changed is that there's one more object called **prev.p** (you can check this by calling **silk.production$prev.p**).

Of course we can remove attached objects from the namespace using the **detach()** function. It removes all variable names from current namespace and once again we must provide full names with **$** to call variables. Fortunately for those that prefer safety over convenience – there's one more solution. By using the **with()** function we can both call variables using their names (without the name of the table) and avoid any problems. This is because **with()** doesn't attach the object permanently, just allows for temporal in-line use of simple variable names. The choice is yours – but be aware of dangers.

```
> with(silk.production, prev.p)
[1] 100 340 546 234 432 765 432 318 287 190 223 210 218 220 431
```

Data input

The last but maybe most important section on data was skipped until now. You probably wondered how to enter your own data to R. the simplest way of doing that is using the **scan()** function, which reads numbers (data) from the standard computer input (the keyboard). Leaving the last position empty ends the data chain and results in the formation of the data vector:

```
> scan()
1: 3
2: 4
3: 5
4: 6
5:
Read 4 items
[1] 3 4 5 6
```

Most commonly we store our research data in electronic spread-sheets such as Excel. We can load them into R in several ways. First – make sure your dataset meets several requirements of the R environment, most importantly variable names are single words or multiple words joined by dots or underlines (they cannot contain

white spaces). Excel file should be saved as the Unicode text file with columns separated by tabulators or as a csv file. To read such file into R we use the **read.table()** function if our numbers have dot-separated decimals or **read.delim2()** if numbers have comma-separated decimals. By specifying **head=T** we indicate that the first row contains variable names. The **sep** argument specifies the character that separates columns (if that should be the tabulator "**\tab**" it can be omitted; if this is comma – use **sep=","** or employ alternative reading function – **read.csv()**; note that in csv files by default decimals must be separated by dots).

```
> mydata <- read.table(file="cisnienie.txt", head=T, sep="\t")
```

Upon reading to R – all text values are automatically converted to factors. In case we wanted them imported as text values we could add the **as.is=T** argument.

Similarly, we can write the data-frame to disc:

```
> write.table(silk.production, file="silk.dat", sep="\t")
```

Manipulating tables

Tabular datasets are preferred in R and it provides several specific functions designed for handling and summarizing data in tabular form. At first these functions may seem complicated and hard to predict, but once you have started using R on a regular basis they prove to be very useful.

The **tapply()** function applies specified procedure to a vector (or column) of data according to some grouping variables. E.g. let's calculate mean current production of silk by individuals of different sex:

```
> with(silk.production, tapply(curr.p,sex,mean))
       1        2
37.62500 47.57143
```

The result of this function is a table structured according to used grouping variables. If there are two grouping variables – the output table becomes more complex:

```
> with(silk.production, tapply(curr.p,list(sex,humid),mean))
  56 76 78 79 87 88 89   90 100
1 NA 23 18 56 22 54 76   NA  34
2 43 NA NA NA NA 67 NA 44.5  45
```

In case of matrices the specified function can be applied to either columns or rows. Rows form the margin number 1, columns are margin number 2. Here we calculate standard deviation column-wise:

```
> mymat <- matrix(silk.production$curr.p,c(3,5))
> mymat
     [,1] [,2] [,3] [,4] [,5]
[1,]   34   76   56   32   23
[2,]   45   54   98   22   43
[3,]   38   28   67   10    8
> apply(mymat,2,sd)
[1]  5.567764 24.027762 21.779195 11.015141 17.559423
```

Then we subtract obtained SD values column-wise from the values of the original matrix:

```
> sds <- apply(mymat,2,sd)
> sweep(mymat,2,sds)
         [,1]      [,2]     [,3]      [,4]      [,5]
[1,] 28.43224 51.972238 34.22081 20.984859  5.440577
[2,] 39.43224 29.972238 76.22081 10.984859 25.440577
[3,] 32.43224  3.972238 45.22081 -1.015141 -9.559423
```

Similar function for vectors is called **sapply(vector, FUN)** – it applies the FUNction to each element of the vector and returns a list.

Part 3 ~ Graphics and plotting

Graphical devices

As anything else in R – graphics may be more complicated than it should. Unfortunately there's no way around – we'll have to go through the basics first and learn some of the most non-intuitive ways of dealing with graphics.

To start with – let's generate a sample data set. The syntax of the following will be a mystery for you (at least until our next meeting) but just assume you're generating linearly correlated data with two variables, *y* being normally distributed:

```
> x <- runif(20, min=1, max=10)
> y <- x + rnorm(20, mean=0, sd=1)
> plot(x,y)
Waiting to confirm page change...
> abline(lm(y~x))
```

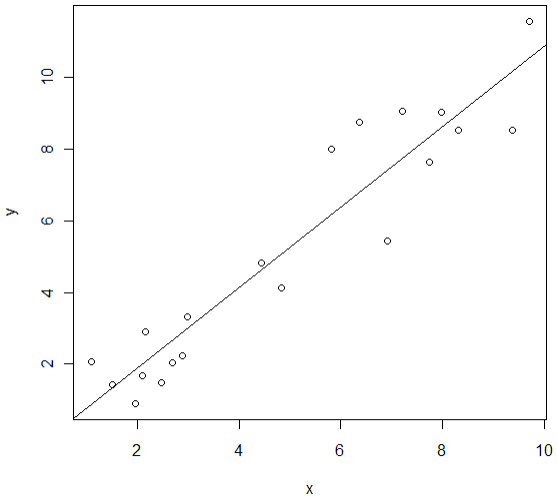


Fig 1: Sample graph

What happened is that we used the function plot to generate the general **`plot`** and then used the function **`abline`** to add the line to it. It's important to remember, that **`abline`** can do its job only when the plot it could draw onto already exists. If not – you'll get an error message. That brings us to the concept of the graphic device in R (Davey, 2009; see also Crawley, 2010). Graphic device is anything that takes your data and prints it in the graphical format. By default, when calling any plotting function in

R, it opens a standard output device – which is simply your screen (that's this window popping up when you call plot). There are other possible devices – some of them allow for writing the graphics to the file so you can save the plot. In windows it's a little bit simpler – having focus on the device window just choose File > Save as > [File format] and you'll be able to save your plot on the hard drive. Below there's a list of the most common graphic devices; they're also names of the functions used for handling them:

Table 1: Graphic devices

windows()	Open MS Windows window
quartz()	Works in MacOS
x11()	UNIX display
pdf()	PDF file (multipage)
postscript()	Postscript file (multipage)
png()	PNG file (one page)
jpeg()	JPG file (one page)
tiff()	TIFF file (one page)

Please notice that some devices allow multiple pages – plotting new graphs will just add the pages to the existing files. Other devices allow just one page and any next page added to the graph will destroy previous content. You can see the list of available devices using **dev.list()**. By using **dev.curr()** you'll get the name of the currently working device. You can open a device by using functions from above table and close it by using **dev.off()**. Finally you can close all open devices by using **graphics.off()**. Let's plot our graph to a PDF file:

```
> pdf("plik.pdf")
> plot(x,y)
> abline(lm(y~x))
> dev.off()
windows
      2
```

In R you can choose the graphing functions from three different packages. The oldest one is **graphics**, which is automatically loaded when R starts. Inside it you can find so called high-level functions for creating graphs of different types – and low-level functions for adding different things to your graphs. But more advanced packages are available that produce more professional-looking and more aesthetical graphs – and these are **ggplot2** and **lattice**. We'll look shortly on the standard graphing functions and then introduce these more advanced packages. Please note that **ggplot2** and **lattice** are not available by default and you'll have to install them manually.

Table 2: High-level plotting functions

assocplot	Association plot
barplot	Bar-plot
boxplot	Box-plot
bxp	Box-plot from summaries
cdplot	Conditional density plot
contour	Contour (map-like) plot
coplot	Conditioning plot
curve	Draw mathematical curve
dotchart	Cleveland Dot plot
filled.contour	Level plot
fourfoldplot	Fourfold plot
hist	Histogram
image	Digital image
matplot	Plot from columns of a matrix
mosaicplot	Mosaic plot
persp	3D plot
pairs	Scatterplot matrix
pie	Pie chart
plot	x-y plot
spineplot	spineplot and spinograms

`stars`	star (spider/radar) plot
`stem`	stem-and-leaf plot

Working with standard graphics

In the **`graphics`** package the way of working is always the same: you create a graph using a high-level function and the you adjust the graph using low-level functions. Below you'll find most widely used high-level functions with brief descriptions of what they do:

Many of these functions take additional arguments for early customization of the plots – look up their help pages for more details. To play with them – let's use some example data set called **`pressure`**. It contains data on temperature and pressure in a data-frame:

```
> plot(pressure)
Waiting to confirm page change...
> plot(pressure$temperature, pressure$pressure)
Waiting to confirm page change...
> plot(pressure~temperature, data = pressure)
Waiting to confirm page change...
```

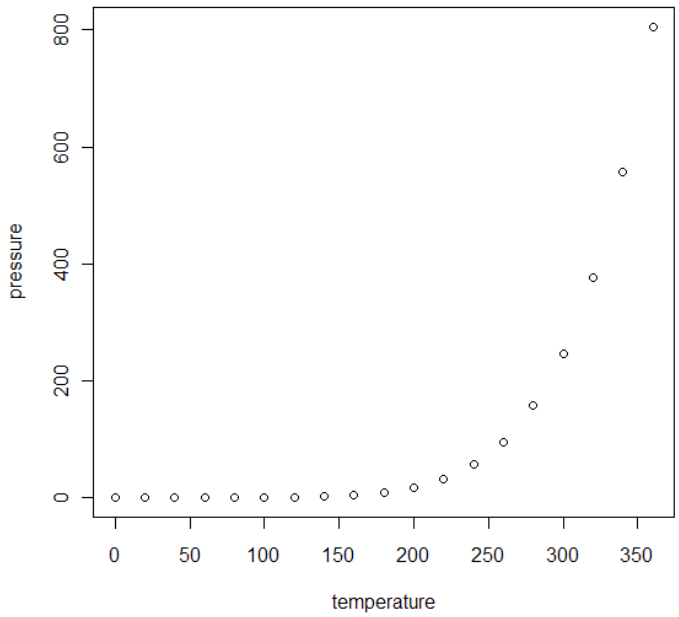


Fig 2: Sample plot

As you can see – all three plots are similar – they just differ in the way the data – and their labels – are assigned to the axes. Actually the third method is the best – it's the most self-explanatory and gives the best axis labels. Other functions use more arguments at the start. E.g. let's generate a histogram of 200 normally distributed random numbers:

```
> hist(rnorm(200))
Waiting to confirm page change...
```

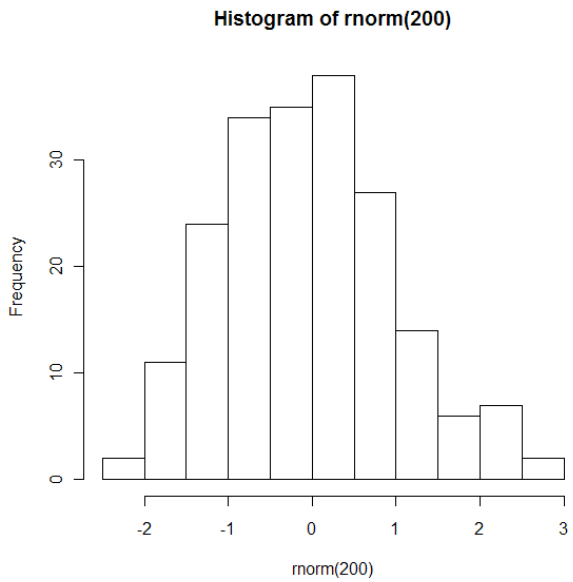


Fig 3: Sample histogram of gaussian data

We might as well adjust shape and position of bars on our graph. Let's create two additional ones (by using **`par()`** which will be explained a bit later). They show two kinds of adjustments we may do to the histogram:

```
> par(mfrow = c(2,1), mex = 0.6)
> hist(rnorm(200), breaks = 20)
Waiting to confirm page change...
> hist(rnorm(200), breaks = c(-4, -3, -1, -0.5, 0, 0.25, 0.5,
+ 0.6, 1, 2, 4))
```

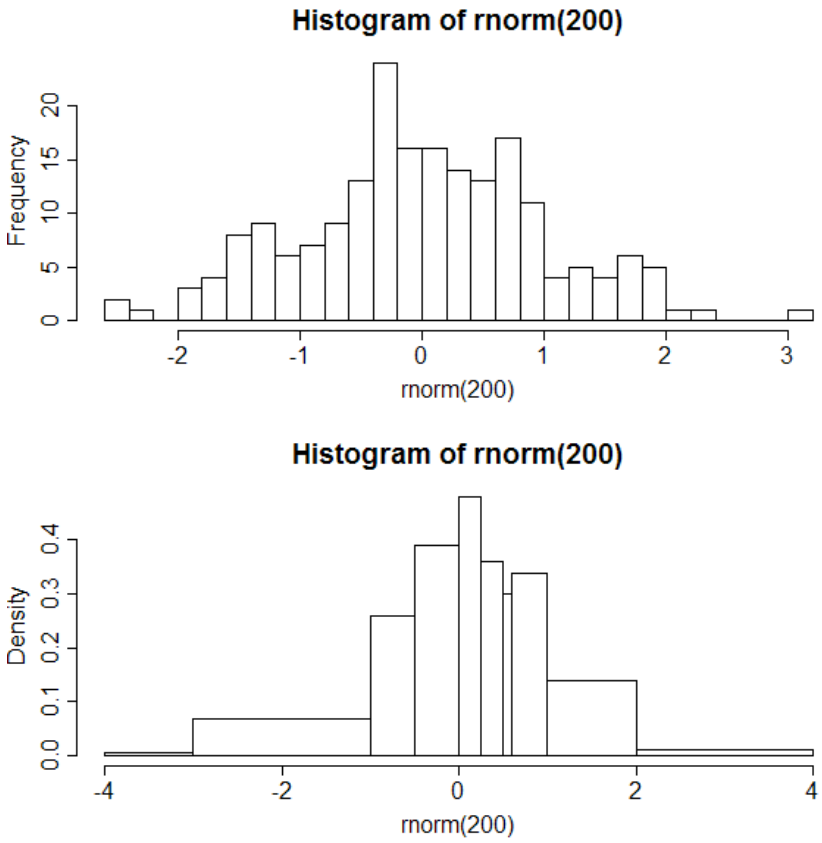


Fig 4: Sample layouts of histograms

Standard X-Y plotting function also allows for some minor adjustments:

```
> y <- rnorm(20)
> par(mfrow = c(2,2), mex = 0.6)
> plot(y, type="l")
Waiting to confirm page change...
> plot(y, type="p")
> plot(y, type="b")
> plot(y, type="h")
```

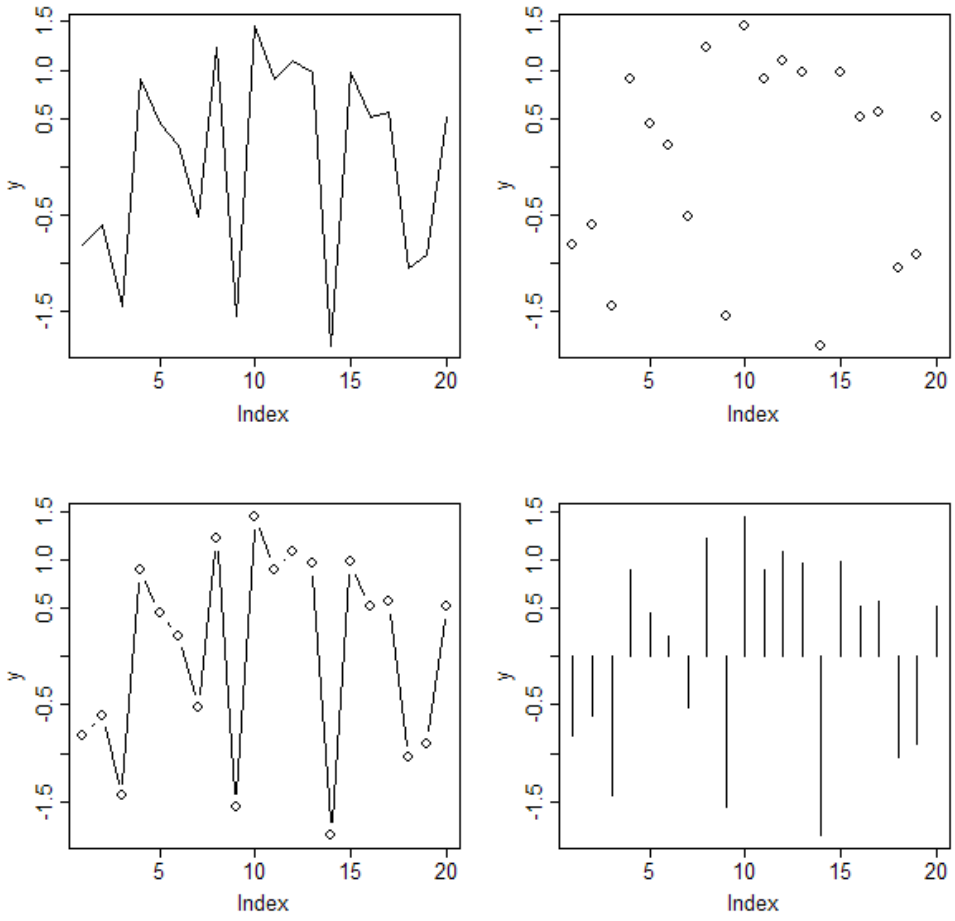


Fig 5: Different types of plots

As you can see – using standard high-level functions is quite straightforward but confusing, especially at the beginning. Problem is that the names of arguments are often quite mysterious and they don't explain themselves so you'll have to remember a lot of them. And that's not over!

Low-level functions

Low-level functions allow for adding elements to existing graphs. Below you'll find the list of common low-level functions and what they do (Table 3):

Let's try to add some elements to our previous graph: we'll add the box around the graph, some text, an arrow and a legend:

```
x <- runif(20, min = 1, max = 10)
> y <- x + rnorm(20, mean = 0, sd = 1)
> plot(x, y, pch = 2, main = "Plot of x and y to show low-level
+ functions")
Waiting to confirm page change...

> lmfit <- lm(y ~ x)
> abline(lmfit)
> box(col = "grey")
> arrows(5, 8, 7, predict(lmfit, data.frame(x = 7)))
> text(5, 8, "Line of best fit", pos = 2)
> legend(7.5, 3, c("x on y"), pch = 2)
```

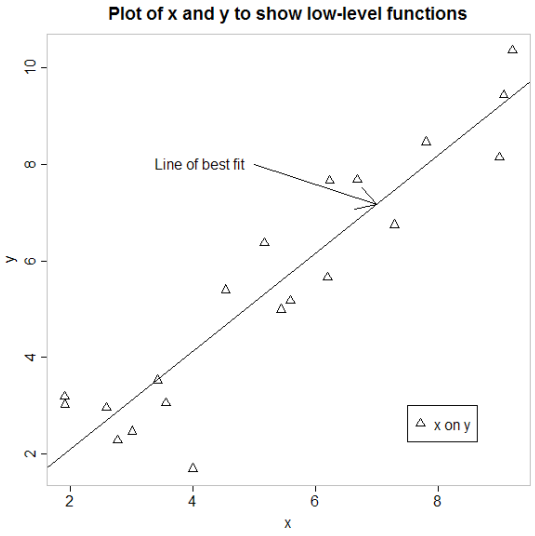


Fig 6: Additional elements as low-level functions

Table 3: Low-level functions

`abline`	Add straight lines to the plot
`arrows`	Add arrows
`axis`	Add axes to the plot
`box`	Draw box around a plot
`grid`	Add grid
`legend`	Add legends
`lines`	Add connected line segments
`mtext`	Write text into the margins of the plot
`panel.smooth`	Simple panel plot
`points`	Add points
`polygon`	Draw polygon
`rect`	Draw rectangles
`rug`	Add rug to the plot
`segments`	Add line segments
`symbols`	Draw symbols
`text`	Add text
`title`	Plot annotation

Graphical states and parameters

So far we've used several different parameters (such as line style) inside the graphing function. When called like this they're applied only to this particular call and don't affect other plots. But we can set these parameters manually to work in the whole R workspace and thus – change default values of these attributes in other R functions. To do this – we should use the **`par()`** function. Everything specified there will change the state of a variable – but you can still override this by specifying your own value in a graphing function call. You can see different types of state variables in the attached table. Some of them are callable from the **`par()`** or in any other graphing function, some can be set only by **`par()`**. Finally – some are read-only and cannot be set by the user. It will become more apparent during the class. It is a good practise to save starting values of these parameters to an object (e.g. **`parameters<-par()`**) in order to be able to restore them if needed (**`par(parameters)`**).

Positioning plots

You've seen the use of **par()** to set multiple plots on one page. By using **mfrow** and **mex** arguments we were able to set the grid of several plots and specify the space between them. Here, we'll do the same for four diagnostic plots from a simple linear regression:

```
> par(mfrow = c(2,2), mex = 0.6)
> plot(lm(y ~ x))
Waiting to confirm page change...
```

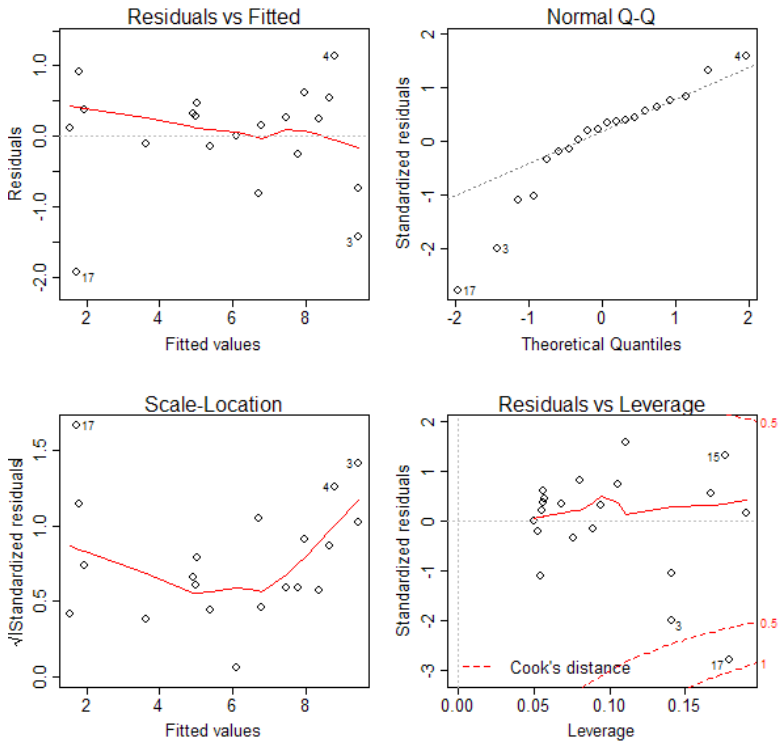


Fig 7: Subdivided plot space

We might also want to adjust the relative heights and widths on our graph. Here we should use layout-manipulating function:

```
> layout(cbind(c(4, 2), c(0,0), c(3,1)), heights = c(2,1),
+ widths = c(2, lcm(2), 1))
> par(mex=0.6)
> plot(lm(y ~ x))
Waiting to confirm page change...
```

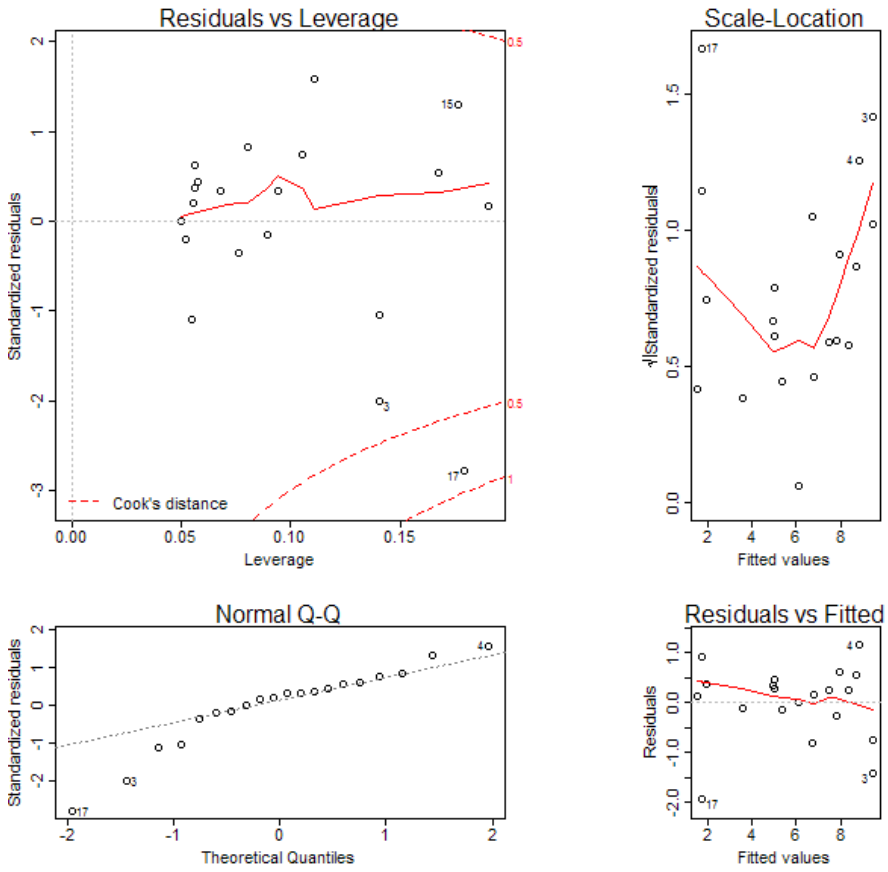


Fig 8: Manipulated layout

On the picture at the end (below the Short Guide to the Most Essential Functions) you'll find an illustration of additional positioning parameters that may influence the layout of your graphs.

GGPLOTS

Do you have the impression that these plotting functions are a little bit to slow and tedious? If yes – you're correct. Standard graphing package of R is old and uses just about 10% of modern computers calculating power. What if we want to have professional, publication-quality plots? I suggest using ggplot2 and lattice. Let's have a brief overview. We'll use some of the example data available with these packages.

First – let's make simple plot for the relationship of body size and brain weight for 83 mammal species:

```
> library(ggplot2)
> attach(msleep)
> plot(bodywt, brainwt, log = "xy")
Waiting to confirm page change...
> abline(lm(log10(brainwt) ~ log10(bodywt)))
```

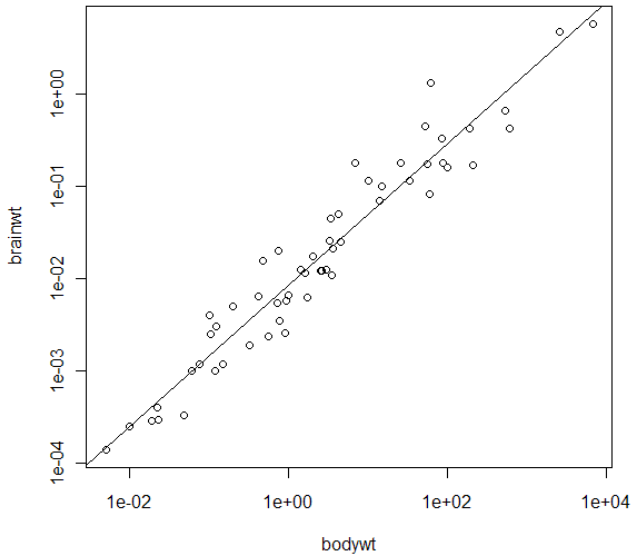


Fig 9: Body weights and brain weights of mammals using graphic package

Now let's try the ggplot2 version (the function is called qplot and mimics the standard plot function):

```
> print(qplot(bodywt, brainwt, log = "xy"))
```

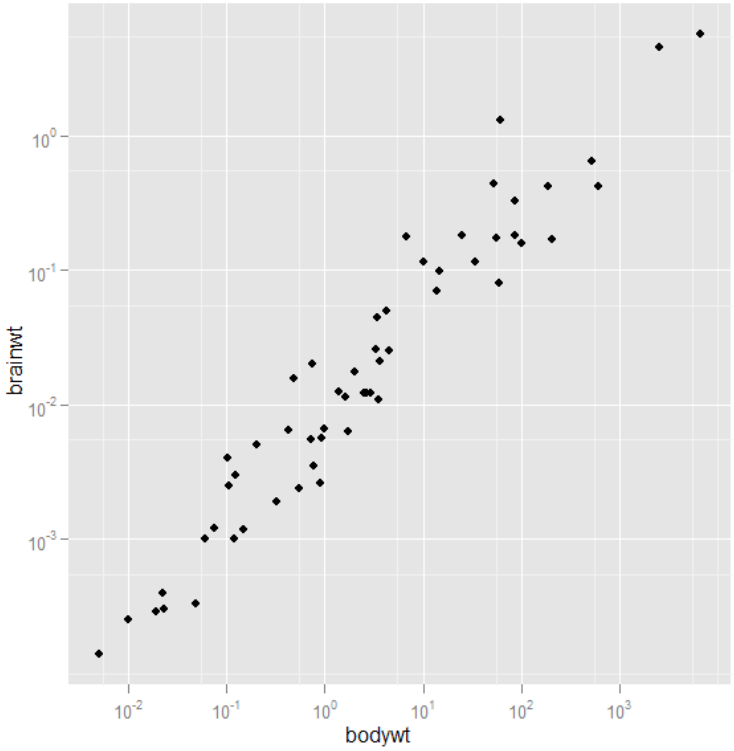


Fig 10: Body and brain weights using qplot

I hope you to think this plot is better and more aesthetical than the previous one. We could also want to add the regression line to our graph. Since elements of the graph in this package are so called *geoms* – we add another geom specifying that we want a regression line:

```
> print(qplot(bodywt, brainwt, log = "xy",
+ geom = c("point", "smooth"), span = 1))
```

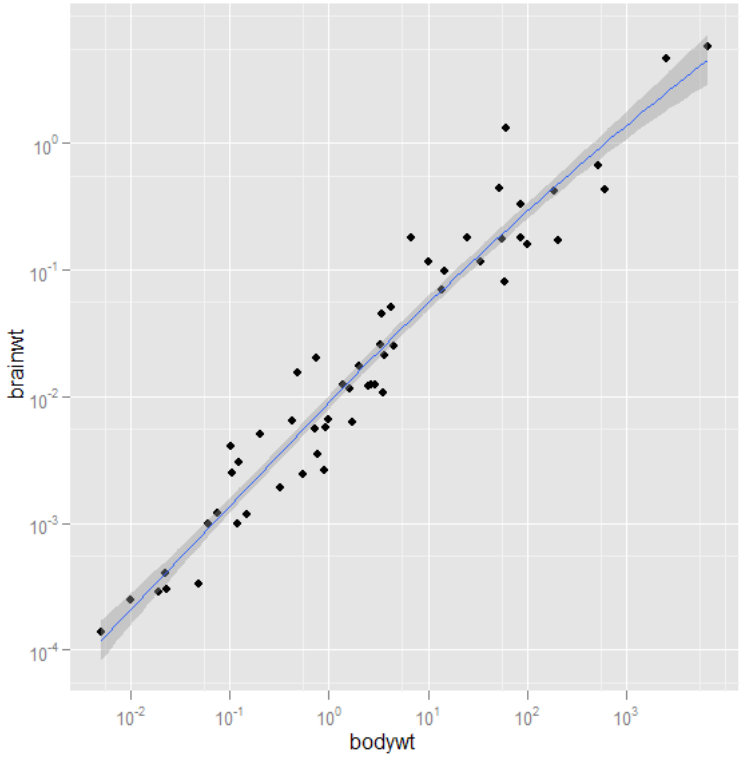


Fig 11: Adding regression line as a smoother

We can also play with the "span" parameter – it regulates the smoothness of the line – or the degree it tries to fit the data correctly.

The data in our dataset are grouped according to the type of food each species takes. We could incorporate this in our traditional plot:

```
> plot(bodywt, brainwt, log = "xy", pch = as.numeric(vore))
Waiting to confirm page change...
> legend(0.1, 4.4, c("carni", "herbi", "insecti", "omni", "NA"),
+ pch = 1:5)
```

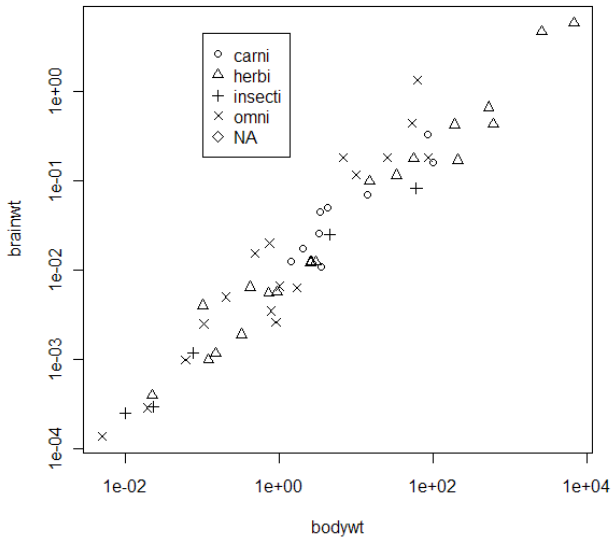


Fig 12: Simple plot with different symbols

Neat? But look at the output of ggplot2:

```
> print(qplot(bodywt, brainwt, log = "xy", shape = vore))
```

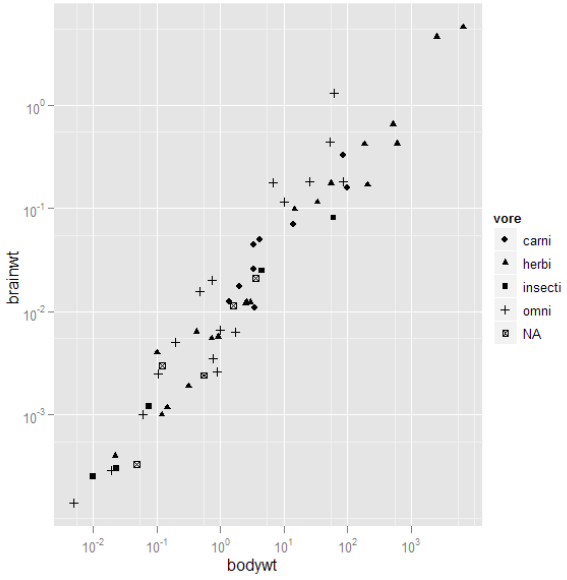


Fig 13: qplot with different symbols

We could also group them by colour:

```
> print(qplot(bodywt, brainwt, log = "xy", shape = vore))
```

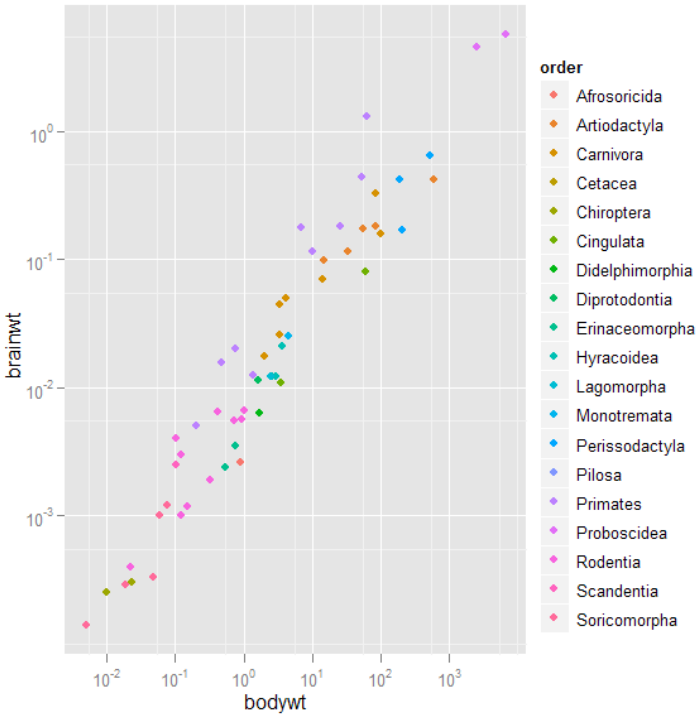


Fig 14: qplot with coloring (see PDF version)

I hope I encouraged you to explore the ggplot2 package. Lattice mentioned earlier is similar – it also produces complex plots of publication-quality, but has superb advantage of plotting complex multivariate plots for random-variable analyses. Feel free to explore their manuals – both are real treasuries of fresh and modern graph types!

Part 4 ~ Hypothesis testing. Simple tests

Distributions

R uses a set of predefined distributions and also provides several functions for obtaining different pieces of information from them. There are four types of distribution-related functions – beginning with the letters **r**, **d**, ,**p** and **q**. They return – respectively: pseudo-random observation from a specified distribution; probability density function (point value) for a given value; cumulative probability for a given value (i.e. *p* of getting value smaller than specified); value from a distribution given specified cumulative probability (i.e. quantile for a given *P*). E.g. for normal distribution:

```
> y <- rnorm(50, mean = 30, sd = 10)

> y
 [1] 18.38744 35.70038 22.02886 26.38781 35.08229 21.40059 22.39499
 [8] 56.68030 34.22761 21.95484 14.69376 38.31176 13.21152 44.17276
[15] 41.42520 35.73230 37.05871 32.00607 60.06589 33.17033 44.65350
[22] 35.79607 26.31287 17.77483 50.62435 40.11203 36.65872 32.58801
[29] 15.16906 13.63348 40.38250 39.48611 46.92860  5.16183 49.33278
[36] 21.83284 33.02274 22.92437 31.50554 30.56412 30.33825 33.05177
[43] 20.17136 40.76705 30.61425 31.85321 22.95693 32.19884 38.83665
[50] 15.75178

> dnorm(30, mean = 30, sd = 10)
[1] 0.03989423
> pnorm(30, mean = 30, sd = 10)
[1] 0.5
> qnorm(0.025, mean = 30, sd = 10)
[1] 10.40036
```

Importantly, these values are much more useful than those drawn from statistical tables – as scientists did before the era of modern computers. They're calculated from a scratch – allowing for precise calculation of errors and other probability-related values in statistics. Other distributions included in R (used in exactly the same way – by prefixing them with an appropriate letter (p, q, d, r)) are presented in Table 4 (adapted from Davey, 2009).

Most of them can be used without any arguments (they take default arguments in such a case) – but look up their help pages to see specific modes of use and detailed information about possible arguments.

Table 4: Distributions available in basic distribution of R

Beta	**beta**	Multinomial	**multinom**
Binomial	**binom**	Negative binomial	**nbinom**
Cauchy	**cauchy**	Normal	**norm**
Chi-Squared	**chisq**	Poisson	**pois**
Exponential	**exp**	Student t	**t**
F	**f**	Studentised range	**tukey**
Gamma	**gamma**	Uniform	**unif**
Geometric	**geom**	Weibull	**weibull**
Hypergeometric	**hyper**	Wilcoxon Rank Sum	**wilcox**
Logistic	**logis**	Wilcoxon Sign Rank	**signrank**
Lognormal	**lnorm**		

Data diagnostics

Before doing any tests it's a good practice to explore our data and look for possible violations of mostly accepted assumptions that should be met in popular tests. Checking your data is simple and in most cases requires some diagnostic plots – but beginning with simple **summary** is always the best way:

```
> summary(leaf$y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.904   2.241   2.414   2.419   2.568   2.984
> leaf <- read.table(file="leaf.txt", head=T)
> par(mfrow=c(2,2))
> plot(leaf$y)
> boxplot(leaf$y)
> hist(leaf$y,main="")
> plot(c(leaf$y,30))
```

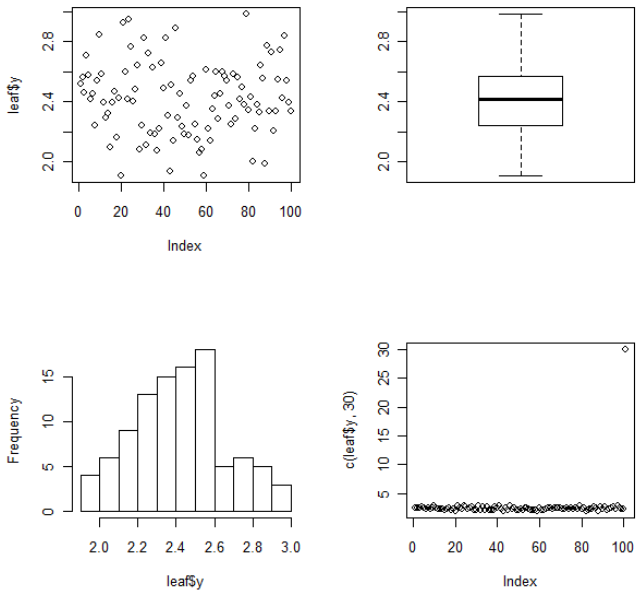


Fig 15: Graphs for data exploration

One quick look on the graphs should be enough to spot possible outliers and deviations form general variation. Outliers can be also spotted with ordinary boxplot:

```
> boxplot(c(leaf$y,4,3,3.5,3.49,3.41), outline=T)
```

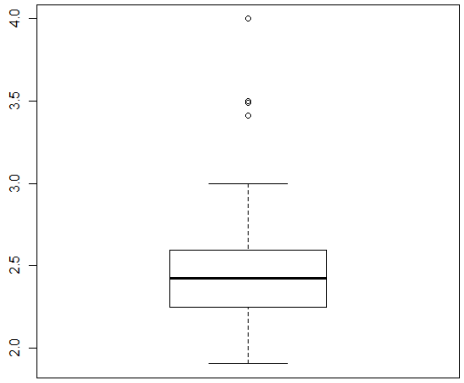


Fig 16: Checking for outliers

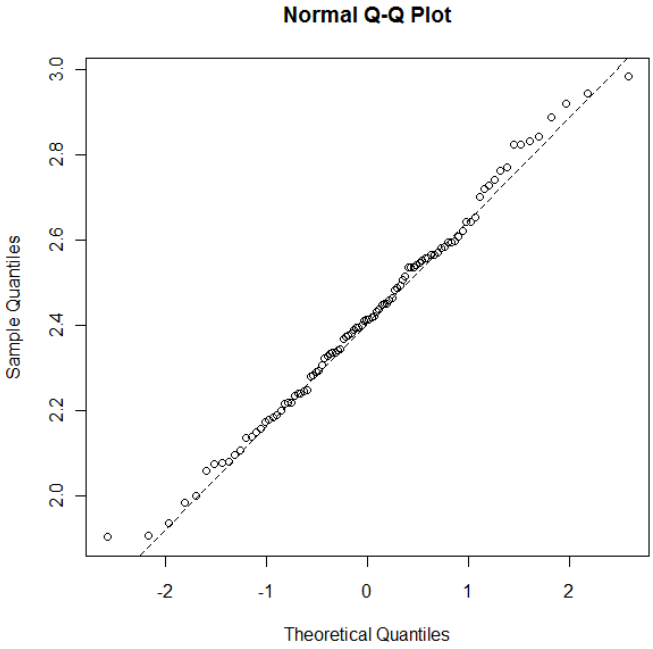


Fig 17: Checking for normality

Another important assumption that has to be met in parametric tests is a normality assumption. It can be easily tested with a quantile - quantile plot, that plots ranked quantiles from our sample against quantiles from the normal distribution. Any severe deviations form a *x = y* line indicate departures from normality. As you can see (Fig. 17) – in our data this is not an issue (it's also evident from near equality of median and mean):

```
> qqnorm(leaf$y)
> qqline(leaf$y,lty=2)
```

Of course, we could use some more formal way of testing normality – e.g. with a Shapiro-Wilk test:

```
> shapiro.test(rpois(n=30,lambda=0.1))

        Shapiro-Wilk normality test

data:  rpois(n = 30, lambda = 0.1)
W = 0.2754, p-value = 4.402e-11
```

```
> shapiro.test(exp(rnorm(100)))

        Shapiro-Wilk normality test

data:  exp(rnorm(100))
W = 0.5563, p-value = 6.854e-16
```

In the second case we have used an obviously non-normal distribution (lognormal) and got a significant result, indicating significant deviation from normality. This is the usual way of using statistical tests in R – after providing something like **`name.test`** you enter any arguments necessary and put in you data. The result is usually some form of summary table with statistics values, *P* values and additional information. It is important to note that any hypothesis test produces a new kind of R object of class **`htest`**. These objects provide several attributes allowing to access some particular "parts" and values in our test object:

```
> sometest <- shapiro.test(exp(rnorm(100)))
> sometest

        Shapiro-Wilk normality test

data:  exp(rnorm(100))
W = 0.6932, p-value = 3.762e-13

> attributes(sometest)
$names
[1] "statistic" "p.value"   "method"    "data.name"

$class
[1] "htest"
```

As any other object – you can assign the result of your test to a variable for further manipulation or to simply store it's result in one place.

Some popular tests

R is a real treasury of statistical tests – just have a look at the Table 5 (adapted from Crawley, 2010):

As always it's good to look through the help pages for these tests to look for specific uses and lists of available arguments. As examples of handling test objects and extracting information from them – we'll deal with some of the most popular test statistics.

If our attempt is to check the difference between two means our first guess would be a Student's *t*-Test. Let's check if the growth rate of daphnias in two different rivers differ significantly:

```
> daphnia <- read.table(file="daphnia.txt", sep="\t", head=T)
> with(daphnia, t.test(Growth.rate~Water))

        Welch Two Sample t-test

data:  Growth.rate by Water
t = -1.0984, df = 63.189, p-value = 0.2762
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.9362193  0.2720467
sample estimates:
mean in group Tyne mean in group Wear
          3.685862           4.017948
```

Table 5: Tests available in the basic distribution of R

Ansari-Bradley test	**ansari.test**	Cochran-Mantel-Haenszel Chi-Sq test	**mantelhaen.test**
Bartlett for homogeneity of V	**bartlett.test**	Mauchly's test of sphericity	**mauchly.test**
Exact binomial test	**binom.test**	Oneway ANOVA	**oneway.test**
Box-Pierce test	**box.test**	Test for trend in proportions	**prop.trend.test**
Chi-squared test	**chisq.test**	Quade test	**quade.test**
Correlation test	**cor.test**	Shapiro-Wilk test	**Shapiro.test**
Fisher exact	**fisher.test**	Student's t-Test	**t.test**
Fligner-Killeen for homogeneity of V	**fligner.test**	F test for comparing variances	**var.test**
Friedman Rank sum test	**friedman.test**	Wilcoxon Rank Sum and Signed Ranks t.	**Wilcox.test**
Kruskal-Wallis Rank Sum test	**kruskal.test**	AND OTHERS…	
Kolmogorov-Smirnov test	**ks.test**		

It appears they don't differ. Displaying attributes of the object created after using *t*-test shows some useful values we could extract in our future analyses:

```
> daph <- with(daphnia, t.test(Growth.rate~Water))
> attributes(daph)
$names
[1] "statistic"   "parameter"   "p.value"     "conf.int"
[5] "estimate"    "null.value"  "alternative" "method"
[9] "data.name"

$class
[1] "htest"
```

Importantly – our data could be aligned in two different ways. Daphnias' data were entered in a way every statistician would do this, but we could as well enter single-variable measures column-wisely and thus creating several columns of the same response. E.g.:

```
> beetles <- read.table(file="beetle.txt", head=T, sep="\t")
> beetles
   beetleX beetleZ
1        3       5
2        4       5
3        4       6
4        3       7
5        2       4
6        3       4
7        1       3
8        3       5
9        5       6
10       2       5
> with(beetles, t.test(beetleX, beetleZ))

        Welch Two Sample t-test

data:  beetleX and beetleZ
t = -3.873, df = 18, p-value = 0.001115
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.0849115 -0.9150885
sample estimates:
mean of x mean of y
        3         5

> parasites <- c(beetles$beetleX, beetles$beetleZ)
> label <- factor(c(rep("X",10),rep("Z",10)))
> plot(parasites~label,notch=T,xlab="Beetle",ylab="Parasites")
```

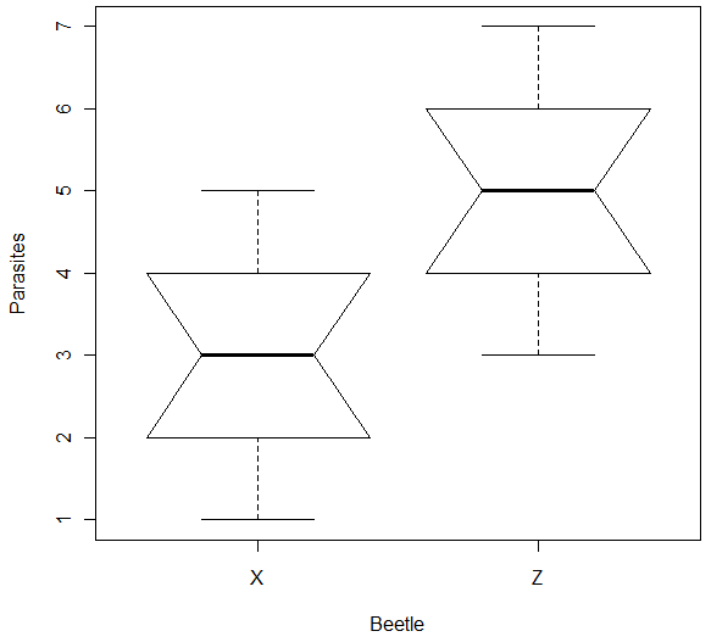


Fig 18: Box-notch graph for beetles example

Here we've tested if the amount of parasites in a beetle differed between two treatments subjecting the insects to different kinds of meds (X and Z). Both the test and the notch-plot indicate that there's a significant difference between the levels of parasite abundance. Here, the test was specified using the treatment groups and not the formula-like expression.

Since our data are counts of parasites we might want to use a non-parametric alternative of the Student's t-test:

```
> with(beetles, wilcox.test(beetleX, beetleZ))

        Wilcoxon rank sum test with continuity correction

data:  beetleX and beetleZ
W = 11, p-value = 0.002988
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(beetleX, beetleZ) :
  cannot compute exact p-value with ties
```

Here we've also found a significant difference – although the test is less powerful. Additionally, we have a warning message that our *P* value is rather an approximation than a strict value (due to ties found in ranked values of our observations).

When doing most of the tests we often would like to determine the nature of our test (one- or two-tailed) and specify if our data constitute two independent samples or paired measures. Let's try a clinical trial example – the following data show the response of the GABA level to some kind of psychotherapy, measured on the same individual pre- and post-therapy. Is there any difference between these levels – specifically, is there any increase in these values, as predicted by theory?

```
> with(gaba, t.test(before, after))

        Welch Two Sample t-test

data:  before and after
t = -0.4088, df = 29.755, p-value = 0.6856
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -5.248256  3.498256
sample estimates:
mean of x mean of y
   12.500    13.375

> with(gaba, t.test(before, after, paired=T,alternative="less"))

        Paired t-test

data:  before and after
t = -3.0502, df = 15, p-value = 0.00405
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
       -Inf -0.3721108
sample estimates:
mean of the differences
                 -0.875
```

Clearly – when an appropriate (and justified!) type of test is used – the differences are apparent. Here, again you should notice that we're more like experimenting with R than learning pure statistics. We should always check our assumptions before performing any test. For the t-test we could check if the variances are equal:

```
> var.test(gaba$before, gaba$after)

        F test to compare two variances

data:  gaba$before and gaba$after
F = 0.8337, num df = 15, denom df = 15, p-value = 0.7292
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
```

```
 0.2912836 2.3860713
sample estimates:
ratio of variances
         0.8336807
```

But perhaps the most important "statistic" we could run is:

```
> length(gaba$before)
[1] 16
```

In other words, if we don't have enough data – we should be extremely cautious in our inferences. How to know how much data is required?

Power calculation

Power analysis in R is easy. There are 3 main functions for doing this, matched with 3 most important statistical techniques you might use: **power.t.test**, **power.prop.test** and **power.anova.test**. In any of these functions you provide several parameters and get the result telling you how you should plan your experiment in order to be able to detect an effect of desired strength. Arguments you provide to these functions are: **n** - the number of observations per group, **delta** – the difference in means you'd like to detect, **sd** – SD of the sample, **sig.level** – default is 5%, **power** – inverse of the II-type error, default is 80%, **type** of the test (independent samples or paired), **alternative** (one-tailed, two-tailed). You should leave one of these as **NULL** – this will become your result. E.g. if the mean of your response is around 20 and you'd like to detect the difference of 10% (i.e. delta = 2) you could call:

```
> power.t.test(delta=2, sd=3.5, power=0.8)

     Two-sample t test power calculation

              n = 49.05349
          delta = 2
             sd = 3.5
      sig.level = 0.05
          power = 0.8
    alternative = two.sided
 NOTE: n is number in *each* group
```

All other parameters are left default (it's for a two-sample two-tailed test). It indicates that for detecting an effect of this magnitude you'd need at least 50 replicates in each group. Check what you'd be able to examine if you could afford a sample of only 60 objects (30 a group).

Bootstrapping

Sometimes you're unable to run classical tests – e.g. distributions look odd. Recall the classical experiment of Michelson (1880) with measuring the speed of light. We'll use his data (as departures from the value of 299000, to make the numbers shorter) and check if this speed is different from 299990 – a value widely accepted in XIX c.

```
> light <- read.table(file="light.txt", sep="\t", head=T)
> summary(light)
     speed
 Min.   : 650
 1st Qu.: 850
 Median : 940
 Mean   : 909
 3rd Qu.: 980
 Max.   :1070
```

As you can see – the mean is very different from median, suggesting strong skewness in our data. It's also apparent from some diagnostic plots:

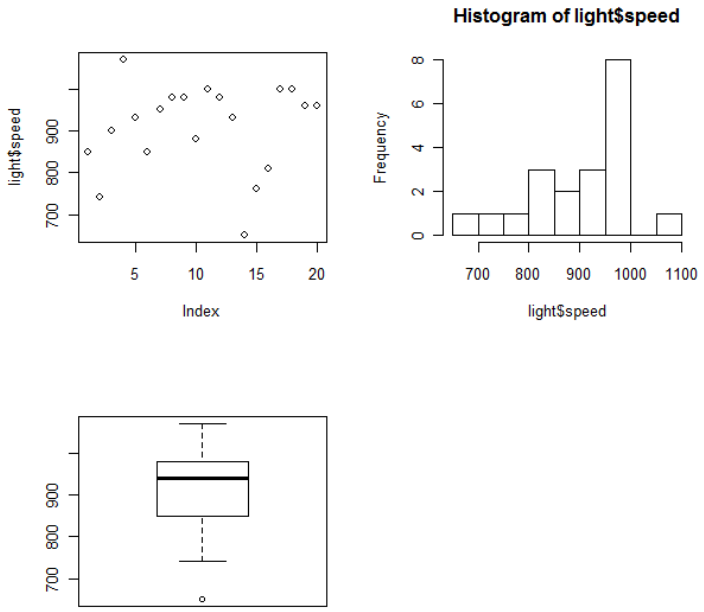


Fig 19: Diagnostic plots for speed-of-light data

Classical tests would probably give biased results. We could use some non-parametric tests, e.g.:

```
> wilcox.test(light$speed, mu=990)

        Wilcoxon signed rank test with continuity correction

data:  light$speed
V = 22.5, p-value = 0.00213
alternative hypothesis: true location is not equal to 990

Warning message:
In wilcox.test.default(light$speed, mu = 990) :
  cannot compute exact p-value with ties
```

But we could use randomization as well – here most probably in the form of bootstrapping. Basically, we have to sample our observations to obtain new samples and based on these samples – we generate the distribution of our sample statistics (in this case – the mean). Since our sample mean is 909 – the distribution based on the null hypothesis will have such mean. Then we should ask – how likely it is to obtain from this distribution the value of 990 (the value we're testing our sample against). If this probability will be low enough (to say – lower than 5%) we'll reject the hypothesis that our sample comes from the population with the mean of 990. More specifically – based on this simulated distribution we should be able to specify the confidence interval for our mean.

```
> a <- numeric(10000) # here we'll store our simulated means (10000)
> for (i in 1:10000) { #we iterate 10000 times...
+
+   a[i] <- mean(sample(light$speed, replace=T))
              # ...every time calculating the mean
+
+ }
> par(mfrow=c(1,1))
> hist(a, main="")
> quantile(a, c(0.025,0.975))
    2.5%    97.5%
863.0000 950.5125
```

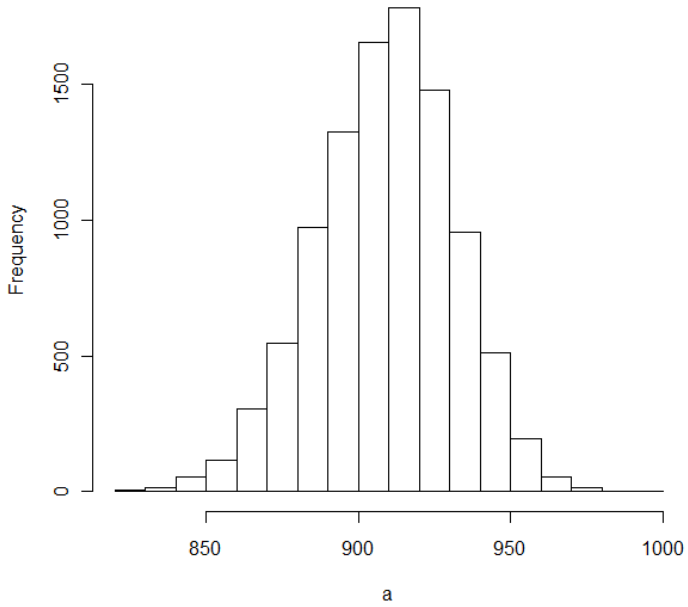


Fig 20: Histogram from bootstrapping

Alternatively we could use an external library boot to the same. Here it's sometimes tricky to specify the statistic's function used for resampling, but once we've done that – the rest is simple:

```
> library(boot)
> mymean <- function(val,i) mean(val[i])
> myboot<-boot(light$speed,mymean,10000)
> myboot

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = light$speed, statistic = mymean, R = 10000)


Bootstrap Statistics :
    original  bias    std. error
t1*      909 -0.0853    23.03312

> boot.ci(myboot)
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
```

```
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = myboot)

Intervals :
Level      Normal              Basic
95%   (863.9, 954.2 )   (866.0, 956.0 )

Level     Percentile            BCa
95%   (862.0, 952.0 )   (856.0, 948.5 )
Calculations and Intervals on Original Scale
Warning message:
In boot.ci(myboot) : bootstrap variances needed for studentized intervals
```

Part 5 ~ Statistical modeling

The outline

Statistical modeling constitutes the essence of contemporary statistical analysis. It stems from one simple idea: try to fit some arbitrary relationship to your data and then estimate the goodness of fit with respect to all parameters included. Several functions allow you to fit statistical models to your data –depending on the form of your model, types of your variables and whether they contain fixed or random effects.

In general the model is defined using so called formula: it's always of the same form: **`response~dependent_variable1+dpenedent_variable2+etc`**. Operators in formulae work in the following way: + means include a variable; – means exclude a variable; **:** forms an interaction between explanatory variables; * includes both single variables and their (all possible) interactions; / means nesting; | conditions (response as a function of a variable given the condition). Below you'll find formulae for most popular types of models (adapted and modified after Crawley, 2010).

As you can see, assigning the formula to a particular type of model is rather arbitrary – all these are just different types of additive (linear) models. Functions that deal with LMs are listed in Table 7 below – your choice will depend on the type of variables in your model (random/fixed) and the distribution your data follow (adapted and modified after Crawley, 2010).

Table 6: Formula syntax

`y ~ 1`	Null model
`y ~ x`	Linear regression – x is continuous
`y ~ factor`	One-way ANOVA – factor is a categorical variable
`y ~ factor1 + factor2`	Two-way ANOVA without interaction
`y ~ A*B*C`	Factorial ANOVA with all possible interactions
`y ~ A*B*C – A:B:C`	3-wayANOVA without the 3rd-order interaction
`y ~ a/b/c`	Factor a nested in b which is nested in c (rarely used in GLMM)
`y ~ factor + x`	ANCOVA – common slope but two intercepts
`y ~ factor*x`	ANCOVA – two slopes and two intercepts

`y ~ x + z`	Multiple regression
`y ~ x + z`	multiple regression with interaction

Table 7: Modeling functions in R

`lm`	`stats`	Linear model – simple regression
`aov`	`stats`	ANOVA and ANCOVA
`anova`	`stats`	Compares two model objects (most often with different fixed-effects structures) and compares their goodness of fit (residual sum of squares)
`glm`	`stats`	Generalised Linear Model (non-gaussian data)
`nls`	`stats`	Nonlinear least squares
`loess`	`stats`	Local polynomial regression (form of smoothing)
`gam`	`gam`	Generalized additive model
`lme`	`nlme*`	Linear mixed effects model (only gaussian data)
`nlme`	`nlme*`	Nonlinear mixed effect model
`lmer`	`lme4*`	Linear, generalized linear and nonlinear models (GLMM) using restricted maximum likelihood (REML)
`MCMCglmm`	`MCMCglmm*`	GLMM using Bayesian statistics (Markov Chains); allows also for fitting random regression models and meta-analyses
`glmmPQL`	`MASS*`	GLMM using penalized quasi-likelihood

It's important to note once again that in fact majority of the above functions would give the same results, at least for similar types of analyses (e.g. for Gaussian data with categorical explanatory variables each of the following – lm, glm, anova – would yield the same conclusions, either qualitatively and quantitatively; everything else being equal – both LM and ANOVA are just different kinds of linear models and differ in the way the design matrix for fixed effects is defined). However, it's always reasonable to choose the most suitable function, both for speeding up our analyses and for obtaining the most suitable and appropriate type of output.

Simple linear models

To illustrate the way statistical modeling works in R let's try to fit simple linear regression to some data. The data below are for the association between mortality

(continuous Gaussian response) and smoking (continuous explanatory variable) – i.e. the best alternative from the above will be **`lm()`**:

```
> smoke <- read.table(file="smoke.txt", sep="\t", head = T, skip = 1)
> smoke.lm <- with(smoke, lm(Mortality~Smoking))
> summary(smoke.lm)

Call:
lm(formula = Mortality ~ Smoking)

Residuals:
    Min      1Q  Median      3Q     Max
-30.107 -17.892   3.145  14.132  31.732

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -2.8853    23.0337  -0.125    0.901
Smoking       1.0875     0.2209   4.922 5.66e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 18.62 on 23 degrees of freedom
Multiple R-squared: 0.513,      Adjusted R-squared: 0.4918
F-statistic: 24.23 on 1 and 23 DF,  p-value: 5.658e-05


> attributes(smoke.lm)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"

> predict(smoke.lm)
       1        2        3        4        5        6        7
 80.85467 146.10660 124.35596  99.34271 123.26842 108.04297 117.83076
       8        9       10       11       12       13       14
 98.25518  92.81752 108.04297  96.08012 110.21804 113.48063 118.91829
      15       16       17       18       19       20       21
120.00583 116.74323 133.05621 141.75647 122.18089 111.30557  91.72999
      22       23       24       25
 96.08012 105.86791  79.76713  68.89181


> resid(smoke.lm)
          1           2           3           4           5
  3.1453349 -30.1066007  -1.3559555  28.6572865  31.7315768
          6           7           8           9          10
 -7.0429716   0.1692381  14.7448187  11.1824800 -20.0429716
         11          12          13          14          15
  7.9198832  18.7819639 -27.4806329 -22.9182942  23.9941735
         16          17          18          19          20
 22.2567703 -20.0562136   4.2435283   5.8191090   3.6944316
         21          22          23          24          25
-12.7299877 -11.0801168  14.1320929 -19.7671329 -17.8918103
```

Last three commands show the attributes of a **lm** object and also provide two functions we can use to access the values predicted by the model and the residuals. You could easily check that the residuals are actually true – of course, if you're still having some doubts R is actually doing what you want it to do:

```
> smoke$Mortality[4:6] - predict(smoke.lm)[4:6]
        4         5         6
28.657286 31.731577 -7.042972
> resid(smoke.lm)[4:6]
        4         5         6
28.657286 31.731577 -7.042972
```

One look at the graph is enough to see that in fact the relationship is strong and significant. Here we generate the x-y plot for the raw data, and then use predicted values for some specified x values (in the range between 60 and 140) to generate fitted line and confidence bands for the regression. **matlines()** adds these three lines to the plot:

```
> with(smoke,plot(Mortality~Smoking,xlab="Smoking [months]",
+ ylab="Mortality [year^-1]", las = 1))
> x2 <- seq(60,140,2)
> y2 <- predict(smoke.lm, list(Smoking=x2),int="c")
> matlines(x2,y2,lty=c(1,2,2),col=c("black","red","red"))
> #adds confdence band
```

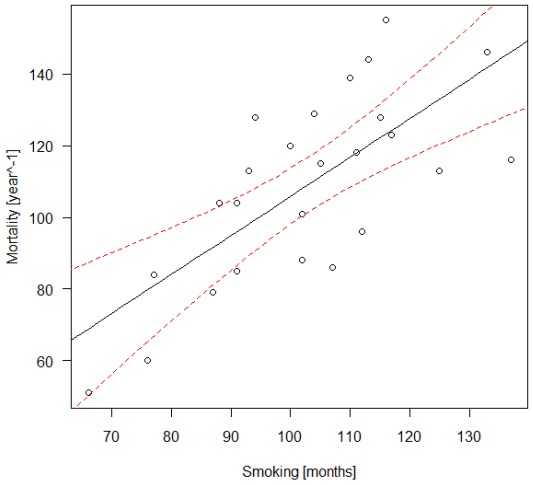


Fig 21: Simple plot with confidence band

Sometimes, especially with complex models, you'd like to re-run the analysis without the need of specifying the model again. Fortunately, all functions making use of formulae (listed in the table above) allow for updating the formulae (which turns out to be useful when doing stepwise simplification of the model):

```
> x <- runif(100, min=10, max=30) # independent variable
> y <- x + rnorm(100, mean=25, sd=11) # correlated response
> z <- runif(100, min=9, max=31) # 2nd independent, not correlated

> model1 <- lm(y~x+z)
> summary(model1)

Call:
lm(formula = y ~ x + z)

Residuals:
    Min      1Q  Median      3Q     Max
-28.773  -9.299  -1.866   9.132  32.705

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.48872    5.79090   5.092 1.74e-06 ***
x            0.85799    0.21555   3.980 0.000133 ***
z            0.01784    0.18239   0.098 0.922288
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.75 on 97 degrees of freedom
Multiple R-squared: 0.1405,     Adjusted R-squared: 0.1227
F-statistic: 7.926 on 2 and 97 DF,  p-value: 0.0006483
```

```
> model2 <- update(model1, ~.-z)
> summary(model2)


Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-28.743  -9.303  -1.785   9.120  32.592

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  29.8598     4.3530    6.86 6.25e-10 ***
x             0.8570     0.2142    4.00 0.000123 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.69 on 98 degrees of freedom
Multiple R-squared: 0.1404,     Adjusted R-squared: 0.1316
F-statistic:    16 on 1 and 98 DF,  p-value: 0.0001228
```

Is the second model really better? Let's check – we'll employ the **anova** function, which (a bit counter-intuitively) does not run any kind of ANOVA (you have **aov** for that), but compares the goodness of fit of two models comparing their residual sums of squares:

```
> anova(model1, model2)
Analysis of Variance Table

Model 1: y ~ x + z
Model 2: y ~ x
  Res.Df   RSS Df Sum of Sq      F Pr(>F)
1     97 13388
2     98 13389 -1   -1.3203 0.0096 0.9223
```

Both models provide similar fit – RSS is a bit lower for the first (more complex) model but the difference is not significant. Having no evidence for significant differences we should obviously choose simpler model.

In fitting any linear model it's important to check assumptions. Two are essential: normality of errors and homogeneity of variance. The easiest way of spotting any violations of assumptions is using diagnostic plots.

```
> par(mfrow=c(2,2))
> plot(smoke.lm)
> par(mfrow=c(1,1))
```

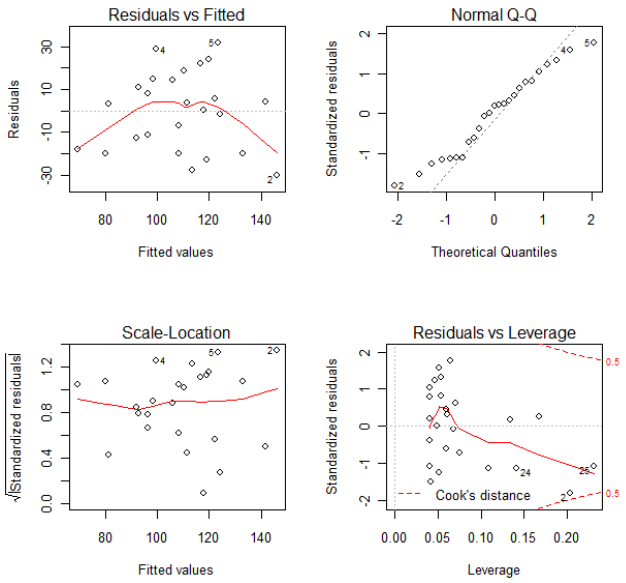


Fig 22: Diagnostic plots for checking assumptions

One quick look is enough to see that residuals are fairly normally distributed and our data only weakly deviate from the normal distribution (plots 1 and 2). We can see (plot 4) that point number 2 (116, 137) has a great influence on our fit (it lies close to the red dashed lines designating Cook's influence statistics critical values). We might try removing this point and looking at the fit again to check if statistical significance is not the result of this one point influencing our analysis:

```
> smoke.lm2 <- with(smoke, update(smoke.lm, subset=(Mortality!=116)))
> summary(smoke.lm2)

Call:
lm(formula = Mortality ~ Smoking, subset = (Mortality != 116))

Residuals:
     Min       1Q   Median       3Q      Max
-29.7425 -11.6920  -0.4745  13.6141  28.7587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -20.0755    23.5798  -0.851    0.404
Smoking       1.2693     0.2297   5.526 1.49e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 17.62 on 22 degrees of freedom
Multiple R-squared: 0.5813,     Adjusted R-squared: 0.5622
F-statistic: 30.54 on 1 and 22 DF,  p-value: 1.488e-05
```

Finally, sometimes – especially in more complex GLM models, we might want to explore the degree of autocorrelation in the residuals of our model. We could use this using **durbinWatsonTest** from **car** library:

```
> install.packages("car")
> library(car)
> durbinWatsonTest(smoke.lm)
 lag Autocorrelation D-W Statistic p-value
   1       0.01874995      1.921095   0.714
 Alternative hypothesis: rho != 0
> with(smoke,dataEllipse(Smoking, Mortality))
```

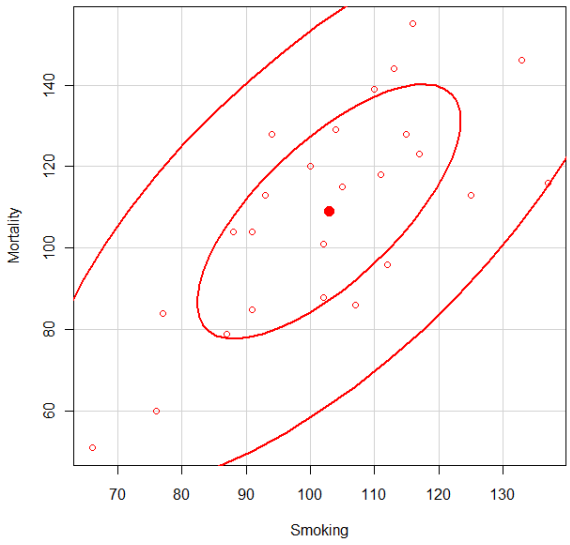


Fig 23: Elipses of normality

Here – there's no evidence for autocorrelation in the residuals (P>0.05). Additionally, we've applied the 'ellipses' function to visualise how good our data reflect a bivariate normal distribution (by default, the 50th and 90th percentiles of the normal distribution are plotted) (Fig 23).

Going multiple – basic tools for models with more than one *x*

Defining a multiple regression model is as straightforward as seen in case of simple regression. The only thing that arises may be the case of nonlinearity (it's more likely we'd have to deal with it when the nr of explanatory variables increases) and the

decision as to which interactions of explanatory variables should we keep. Here we'll see a very simple case of multiple regression, where the response (subjective taste of a cheese) is correlated with three variables objectively describing a cheese (lactic acid concentration, hydrosulphide content and acetic acid concentration). We begin with familiarizing ourselves with the data by plotting simple pairwise regressions for all possible pairs of variables:

```
> ser <- read.table(file="ser.txt", head=T, sep="\t")
> ser <- ser[,-1]
> pairs(ser, panel=panel.smooth)
```

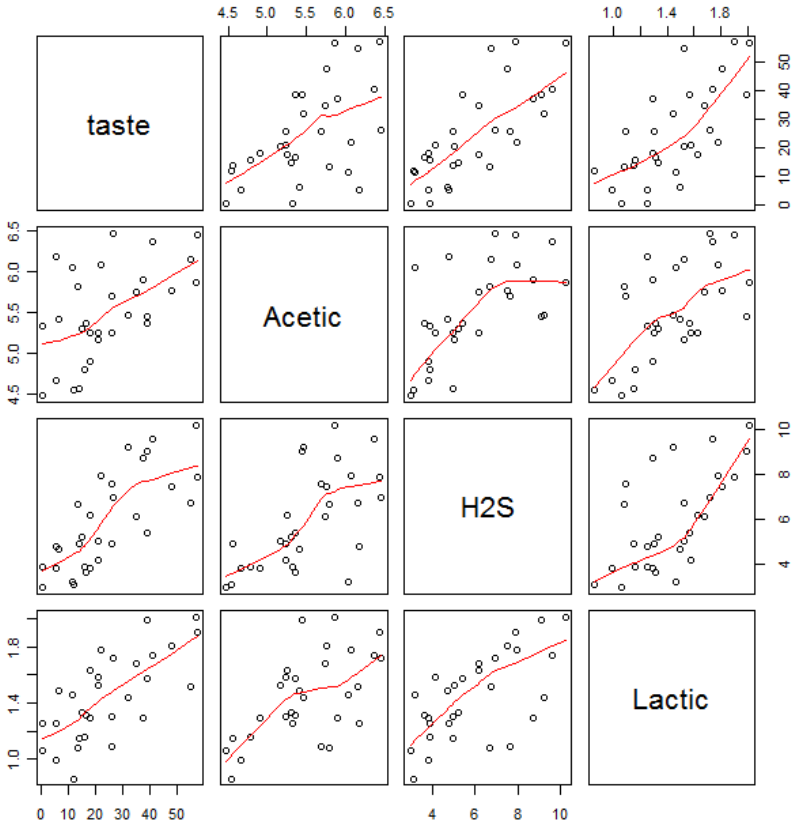


Fig 24: Pair-wise plot of all variables

It seems that taste is positively correlated with all three descriptors of cheese chemistry. Further – could we expect any curvilinear relationships? The pairwise plots are not definitive – it's good to employ general additive models which don't assume

anything about linearity of the relationship (in fact – the allow for non-linearity) and they employ non-parametric smoothing techniques to fit the most appropriate form of relationship:

```
> library(mgcv)
This is mgcv 1.6-2. For overview type 'help("mgcv-package")'.
> par(mfrow=c(2,2))
> modelgam <- gam(taste~s(Acetic)+s(H2S)+s(Lactic), data=ser)
> plot(modelgam)
> par(mfrow=c(1,1))
```

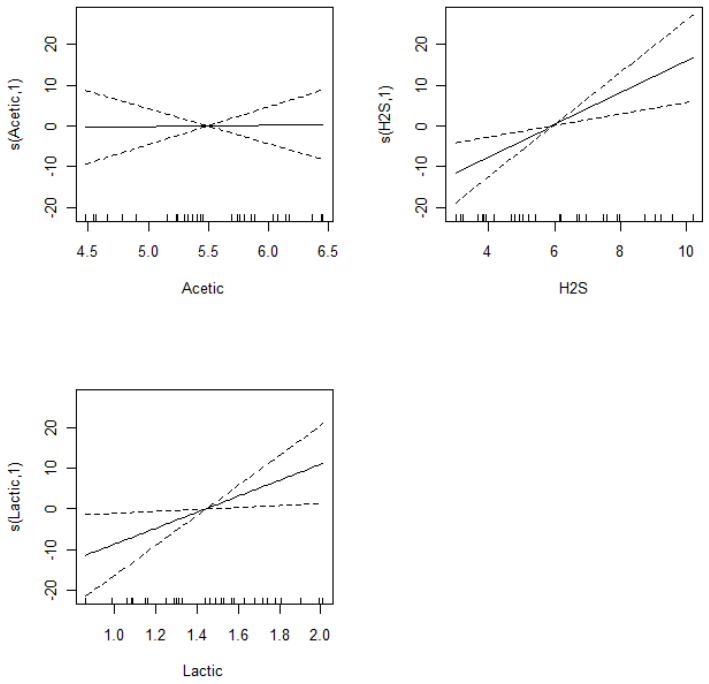


Fig 25: Plots from GAM function

All three plots (as you probably guessed – they show relationships together with confidence intervals) show no evidence for non-linearity – we can than omit all quadratic and higher order terms in our subsequent models. How should we deal with non-linearity once spotted? The easiest way is fitting higher order terms – i.e. quadratic, cubic and higher by using **`I()`** function (see below) or by forming polynomials (expressions approximating our relationship by using power series of an appropriate order; function **`poly()`**). The decision which to choose is rather formal – the results would be the same. Let's look at this example; here you'll generate simple data with curvilinear pattern (by imposing a quadratic function). As you can see,

simple linear regression yields poor fit. Adding one quadratic term of the explanatory variable makes the fit much better:

```
> x <- runif(20,1,20) #explanatory variable
> y <- x^2 + rnorm(20,5,15) #parabola-shaped response
> plot(x,y, las=1) #las makes y labels horizontal
> linear <- lm(y~x) #linear predictor
> abline(linear, col="red", lty=2) #lty sets the dashed line
> quadratic <- lm(y~x+I(x^2)) #parabolic predictor
> range <- seq(1,20,1) #range for parabole x values
> yp <- predict(quadratic, list(x=range)) #parabole y values
> lines(range,yp,col="blue",lty=2) #adds parabole to the plot
> anova(linear,quadratic) #compares two models using ANOVA
Analysis of Variance Table

Model 1: y ~ x
Model 2: y ~ x + I(x^2)
  Res.Df     RSS Df Sum of Sq      F    Pr(>F)
1     18 21530.1
2     17  3053.5  1     18477 102.87 1.258e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see – the parabola fits our data much better than a straight line. ANOVA comparing these two models confirms the result: with quadratic term (which also is statistically significant in the summary of **`lm()`**) our model yields much lower RSS.

Let's go back to our cheese. We know quadratic or higher terms won't be necessary. Should we include any interactions? The tree regression analysis should give us quick answer. This is still rarely used method, but highly informative and also speeds up analyses by employing by-eye inspection. In tree regression all effects from our model form tree-like structure; the most influential variables form the longest branches. In case of interactions, we should note unequal distribution of variables on different branches of the tree, implying differential influence of these variables depending on the value of another variable (located higher in the tree hierarchy). Let's look at the example:

```
> library(tree)
> modeltree <- tree(taste~., data = ser)
> plot(modeltree)
> text(modeltree)
```

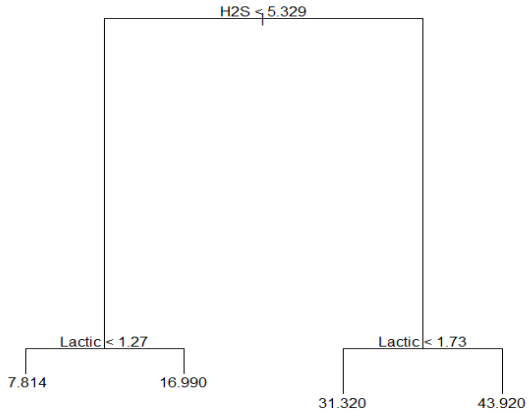


Fig 26: Example of the regression tree

The longest branches reflect the influence of H2S variable. Second most influential is Lactic –lengths of their branches in both groups of H2S are equal indicating lack of any interaction. In line with GAM analysis – Acetic seems to have no influence on the response. We've decided to fit simple multiple regression, with no curvilinear terms. We'll include second-order interaction just to illustrate the way we deal with multiple term models but obviously as no interactions seems to be involved – we'll omit 3rd order interaction of all three variables.

```
> model3 <- lm(taste ~ Acetic+Lactic+H2S+Acetic*Lactic*H2S-
+ Acetic:Lactic:H2S, data = ser)
> summary(model3)

Call:
lm(formula = taste ~ Acetic + Lactic + H2S + Acetic * Lactic *
    H2S - Acetic:Lactic:H2S, data = ser)

Residuals:
    Min      1Q  Median      3Q     Max
-19.675  -5.273  -1.011   5.785  25.072

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   104.5479   102.2068   1.023    0.317
Acetic        -30.7974    23.9574  -1.286    0.211
Lactic        -60.1029   116.5393  -0.516    0.611
H2S             6.1573    20.9537   0.294    0.772
Acetic:Lactic  19.0531    22.6367   0.842    0.409
Acetic:H2S      0.5734     3.5246   0.163    0.872
Lactic:H2S     -3.6493     4.4944  -0.812    0.425

Residual standard error: 10.33 on 23 degrees of freedom
Multiple R-squared: 0.6795,     Adjusted R-squared: 0.5959
F-statistic: 8.127 on 6 and 23 DF,  p-value: 8.718e-05
```

Oops. It seems nothing noticeable is happening in our dataset?! Maybe it's because we've included these interactions without any firm evidence they should influence the response. We could follow a step by step simplification of our models, based on P values and goodness-of-fit parameters – but currently a widely accepted procedure is the application of AIC (Akaike's Information Criterion). R provides an automated engine for selecting the best model based on AIC values (a reminder – the lower the AIC the better the model's fit to our data):

```
> summary(step(model3))
Start:  AIC=146.15
taste ~ Acetic + Lactic + H2S + Acetic * Lactic * H2S - Acetic:Lactic:H2S

               Df Sum of Sq    RSS    AIC
- Acetic:H2S    1     2.826 2458.9 144.19
- Lactic:H2S    1    70.402 2526.4 145.00
- Acetic:Lactic 1    75.651 2531.7 145.06
<none>                      2456.0 146.15

Step:  AIC=144.19
taste ~ Acetic + Lactic + H2S + Acetic:Lactic + Lactic:H2S

               Df Sum of Sq    RSS    AIC
- Lactic:H2S    1    69.427 2528.3 143.02
<none>                      2458.9 144.19
- Acetic:Lactic 1   204.750 2663.6 144.59

Step:  AIC=143.02
taste ~ Acetic + Lactic + H2S + Acetic:Lactic

               Df Sum of Sq    RSS    AIC
- Acetic:Lactic 1    140.13 2668.4 142.64
<none>                      2528.3 143.02
- H2S           1    992.67 3521.0 150.96

Step:  AIC=142.64
taste ~ Acetic + Lactic + H2S

         Df Sum of Sq    RSS    AIC
- Acetic  1      0.55 2669.0 140.65
<none>                2668.4 142.64
- Lactic  1    533.32 3201.7 146.11
- H2S     1   1007.66 3676.1 150.25

Step:  AIC=140.65
taste ~ Lactic + H2S

         Df Sum of Sq    RSS    AIC
<none>                2669.0 140.65
- Lactic  1    617.18 3286.1 144.89
- H2S     1   1193.52 3862.5 149.74

Call:
lm(formula = taste ~ Lactic + H2S, data = ser)

Residuals:
    Min      1Q  Median      3Q     Max
-17.343  -6.530  -1.164   4.844  25.618

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)  -27.592      8.982  -3.072  0.00481 **
Lactic        19.887      7.959   2.499  0.01885 *
H2S            3.946      1.136   3.475  0.00174 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.942 on 27 degrees of freedom
Multiple R-squared: 0.6517,     Adjusted R-squared: 0.6259
F-statistic: 25.26 on 2 and 27 DF,  p-value: 6.551e-07
```

At the end we're left we the best-fitting model and indeed – this model shows significant trends in variables indicated by **`tree()`**. Remember that using any of automated methods in R does not release you from being critical and conservative. I also suggest reading some literature on AIC and information-based approaches as they have as much advantages and followers as drawbacks. You should always use both software-driven tests and rational biological thinking.

Robust regression

As we've seen in case of our first diagnostic analysis for **`smoke.lm`** – sometimes single points in our datasets have a great influence on the fit of our model. Often these points are also outliers – and if we don't have any reason to discard these points as erroneous – we have to deal with them in some other way. Robust regression is what brings us closer to the solution. In robust regression the algorithms estimate parameters of our model minimizing not sum of squared residuals (as in classical OLS estimation) but e.g. absolute values of residuals (as in least absolute deviations – LAD) or in methods employing M-estimators (which use minimizing of other functions of residuals; the OLS and LAD are just special cases of the latter). To use robust regression you should download and install the **`MASS`** package. The main function of interest would be **`rlm()`**. I'll leave exploring these possibilities to those interested:

```
> install.packages("MASS")
> library(MASS)
> ?rlm
```

ANOVA and the use of contrasts

Linear regression implies that your explanatory variables are continuous. But the framework of linear modeling can easily be extended to cases where the variables are categorical. In the sense of linear model specification it means estimating differences between the overall mean and means associated with specific factor levels rather than slopes for continuous variables. It should become clearer when we use the example. Here we'll be analysing the data (adapted from Crawley, 2010) on the clipping

experiment carried out to show how cutting shoots or roots of adjacent plants influences the production of biomass of the focal individual and hence – the competition among them.

```
> clipping <- read.table(file="competition.txt", sep="\t", head=T)
> summary(clipping)
    biomass         clipping
 Min.   :415.0   control:6
 1st Qu.:508.8   n25    :6
 Median :568.0   n50    :6
 Mean   :561.8   r10    :6
 3rd Qu.:631.8   r5     :6
 Max.   :731.0
> clipanova <- aov(biomass~clipping, data=clipping)
> summary.aov(clipanova)
            Df Sum Sq Mean Sq F value   Pr(>F)
clipping     4  85356 21339.1  4.3015 0.008752 **
Residuals   25 124020  4960.8
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1


> summary.lm(clipanova)

Call:
aov(formula = biomass ~ clipping, data = clipping)

Residuals:
     Min       1Q   Median       3Q      Max
-103.333  -49.667    3.417   43.375  177.667

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   465.17      28.75  16.177  9.4e-15 ***
clippingn25    88.17      40.66   2.168  0.03987 *
clippingn50   104.17      40.66   2.562  0.01683 *
clippingr10   145.50      40.66   3.578  0.00145 **
clippingr5    145.33      40.66   3.574  0.00147 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 70.43 on 25 degrees of freedom
Multiple R-squared: 0.4077,     Adjusted R-squared: 0.3129
F-statistic: 4.302 on 4 and 25 DF,  p-value: 0.008752
```

First of all – we see two types of output available here. In case of regression the output was simple – we just got the intercept (the overall mean for all explanatory variables equal to zero) and estimates of slope(s). Here – it's different. In the **summary.aov** table we get familiar ANOVA table listing all relevant sources of variation and partitioning of this variation allowing for the F-test. **summary.lm** is different: here we have some estimates for all treatment levels. What do they mean? By default, ANOVA in R fits so called treatment contrasts that compare the intercept (which is the mean of the first – alphabetically or numerically – factor level; control in

this case) with the means for other treatments. As you could expect – these contrasts may be a little confusing – here it looks like we'd have to retain all parameters in our model - which in fact is not the case! Note that e.g. the estimates for **clippingr10** and **clippingr5** are not statistically different (the difference is less than ~2SE) so they could be merged together freeing some degrees of freedom and simplifying our ANOVA. Moreover – these contrasts are not orthogonal – they're not independent one from another simply because comparing **Intercept** with **clipping25** and **clipping50** employs multiple comparison of the same means. Could we specify our own contrasts? Yes – and actually it's quite easy. The only thing we should learn is the syntax of contrast specification. How it works? Contrasts are specified by assigning positive and negative numbers to the opposing groups of means (or single means) that have to be compared. E.g. if we wanted to compare our control with four treatment means, we could assign some negative value to the control, and positive values to the remaining means. The trick is that these + / - coefficient should add up to zero. For our experiment we could specify the following contrasts: control vs. treatment means; shoot clipping means vs. root clipping means; means for different shoot clipping levels; means for different root clipping means. The matrix specifying these contrasts would be as follows:

```
> comparis <- cbind(c(4,-1,-1,-1,-1),c(0,1,1,-1,-1),
+ c(0,0,0,-1,1),c(0,1,-1,0,0))
> rownames(comparis) <- c("control", "clippingn25",
+ "clippingn50", "clippingr5", "clippingr10")
> comparis
           [,1] [,2] [,3] [,4]
control       4    0    0    0
clippingn25  -1    1    0    1
clippingn50  -1    1    0   -1
clippingr5   -1   -1   -1    0
clippingr10  -1   -1    1    0
```

Here each column stands for a single comparison, rows stand for treatment levels. Every time zero appears in the matrix, it means that a particular effect is excluded from the comparison. You can see that all columns add up to zeros. Moreover, pairwise products of any of two columns add to zero, which indicates that our contrasts are orthogonal (independent comparisons).

Assigning contrasts works by simply modifying the contrasts attribute of our data frame. The we can rerun our ANOVA. Be careful of the double naming **clipping$clipping**, resulting from the fact that both our data and the treatment on our data are called **clipping**:

```
> contrasts(clipping$clipping) <- comparis
> clipanova <- aov(biomass~clipping, data=clipping)
```

```
> summary.lm(clipanova)

Call:
aov(formula = biomass ~ clipping, data = clipping)

Residuals:
     Min       1Q   Median       3Q      Max
-103.333  -49.667    3.417   43.375  177.667

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 561.80000   12.85926  43.688  < 2e-16 ***
clipping1   -24.15833    6.42963  -3.757 0.000921 ***
clipping2   -24.62500   14.37708  -1.713 0.099128 .
clipping3    -0.08333   20.33227  -0.004 0.996762
clipping4    -8.00000   20.33227  -0.393 0.697313
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 70.43 on 25 degrees of freedom
Multiple R-squared: 0.4077,     Adjusted R-squared: 0.3129
F-statistic: 4.302 on 4 and 25 DF,  p-value: 0.008752
```

How should we interpret these results? For every contrast we have one **clippingX** label. They are respective to the contrasts we've specified. It's clear that the only parameter we need in our model reflects comparison of the control with all four treatments. Both shoot vs. root pruning and shoot/root pruning themselves don't generate any statistically significant differences. It's important to note that – although all **clippingX's** estimate differences between means specified in contrasts – the intercept is no longer for the control-treatment mean. Now it estimates the overall mean (all treatments and control pooled together).

Generalized Linear Models

Sometimes – or better said – quite often (in biology) our data do not conform with the assumption of error normality. If our data are in form of counts, proportions, binaries or time-to-death – we are likely to observe diverse kinds of relationships between mean ad variance. This violates one of the most important assumptions of classical ANOVA or linear regression, i.e. the assumption of homogeneity of variances. One possible solution is to transform our data (using standard methods such as logs, roots – or by using more advanced approaches, like the **boxcox()** function from the **MASS** library, applying the Box-Cox transformation). Alternatively we can use linear modeling by applying an appropriate link function that normalizes our response rather than using pure response data. The latter approach is equivalent with using generalized linear models (generalized stands for "allowing non-normal error structures"). To do this we employ **glm()** function in R.

Let's see how it works by using a simulated dataset. We will define our theoretical generalized linear equation (i.e. we'll know the exact form of the linear predictor) and then we'll see if an appropriate GLM model yields correct values. Here we'll use both normally and Poisson distributed responses, with the intercept of 1 (μ = 1), regression coefficients β_2 = 0 and β_3 = 1 and residual variance 2; thus our full linear predictor (it's parametrical form in the population) would be y = 0.x + 1.z and we'll fit the following model: $y \sim \mu + \beta_2 + \beta_3$ + error. We aim at estimating the values of the intercept and two regression coefficients, plus the residual variance.

```
> x <- runif(1000, 0, 1) #uniform random variable
> z <- rnorm(1000, 0, sqrt(1.5)) #normal random variable
> lin <- 1 + 0*x + 1*z #desired linear predictor
> y <- rnorm(1000, lin, sqrt(2)) #added residuals form the response
> yp <- rpois(1000, exp(lin)) #alternative Poisson response
> glmdata <- data.frame(yn = y, yp = yp, x = x, z = z)
> glmnorm <- glm(yn ~ x + z, data=glmdata)
> summary(glmnorm)

Call:
glm(formula = yn ~ x + z, data = glmdata)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-4.58847  -0.87353   0.04924   0.85829   3.66437

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.02776    0.09139  11.246   <2e-16 ***
x            0.08314    0.15163   0.548    0.584
z            1.04039    0.03553  29.284   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 1.932124)

    Null deviance: 3585.4  on 999  degrees of freedom
Residual deviance: 1926.3  on 997  degrees of freedom
AIC: 3501.5

Number of Fisher Scoring iterations: 2
```

The results of this fit (of course every time they'll be slightly different due to random nature of our input data) are satisfactory; they correspond to what we've simulated. Particularly, the values of the regression coefficients are correct. Now see what would happen if you tried fitting the Poisson response using normal error structure:

```
> glmnorm2 <- glm(yp ~ x + z, data = glmdata)
> summary(glmnorm2)

Call:
glm(formula = yp ~ x + z, data = glmdata)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
 -9.656   -3.996   -1.318    1.795  115.922

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.9117     0.5119  11.548   <2e-16 ***
x            -0.4318     0.8493  -0.508    0.611
z             5.6205     0.1990  28.242   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 60.6248)

    Null deviance: 108799  on 999  degrees of freedom
Residual deviance:  60443  on 997  degrees of freedom
AIC: 6947.6

Number of Fisher Scoring iterations: 2
```

Due to severe violation of assumptions (in Poisson data the variance exhibits a linear relationship with the mean) the estimates of our parameters are severely biased. Removing this bias is possible by using an appropriate link function (i.e. by specifying the correct error structure):

```
> glmpois <- glm(yp ~ x + z, data=glmdata, family = poisson)
> summary(glmpois)

Call:
glm(formula = yp ~ x + z, family = poisson, data = glmdata)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.0094  -0.9051  -0.1908   0.6208   3.3223

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.98370    0.03228  30.475   <2e-16 ***
x            0.01154    0.04551   0.254      0.8
z            0.99788    0.01089  91.647   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 9590.9  on 999  degrees of freedom
Residual deviance: 1048.3  on 997  degrees of freedom
AIC: 3741.9
```

```
Number of Fisher Scoring iterations: 5
```

Here our estimates are correct because we applied the correct error structure, in accordance with the distribution of the response in our data. That should emphasize the importance of careful inspection of your data. Unfortunately, sometimes using the proper distribution is not enough to fit an appropriate model to your data. Here we have the luxury of knowing exactly what is the true form of the linear predictor that was used to generate the data. In reality we rarely have this opportunity and we have to 'guess' the best composition of our final model. In case of Poisson or binomial distributions failing to include some effects in our model introduces additional residual variation. Unfortunately, in case of such data this extra variation violates the most basic assumption about e.g. Poisson distribution and this is the equality of the mean and variance. We call this overdispersion and it's quite common in the real data. We can show how it influences by simply intentionally omitting one important part of our predictor:

```
> glmpois2 <- glm(yp ~ x, data=glmdata, family = poisson)
> summary(glmpois2)

Call:
glm(formula = yp ~ x, family = poisson, data = glmdata)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-3.39770  -2.45693  -1.27163   0.09496  25.49511

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.7530490  0.0273656  64.060   <2e-16 ***
x           -0.0003854  0.0454007  -0.008    0.993
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 9591  on 999  degrees of freedom
Residual deviance: 9591  on 998  degrees of freedom
AIC: 12283

Number of Fisher Scoring iterations: 6
```

Overdispersion in this analysis (predictors account only for some proportion of variance and thus the residual variance in higher than expected from the Poisson process) causes severe bias in the estimates. We can have the idea about the degree of overdispersion by looking at the residual deviance to residual *df* ratio: in non-overdispersed data it should be equal to one. Here it's nearly 10 times higher! Roughly

speaking, our data contain 10 times as much variation as they should according to the Poisson process.

We could try fitting this model with alternative to Poisson called quasiPoisson, which was designed to deal with this additional variation:

```
> glmqpois <- glm(yp ~ x, family=quasipoisson, data=glmdata)
> summary(glmqpois)

Call:
glm(formula = yp ~ x, family = quasipoisson, data = glmdata)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-3.39770  -2.45693  -1.27163   0.09496  25.49511

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.7530490  0.1189384  14.739   <2e-16 ***
x           -0.0003854  0.1973240  -0.002    0.998
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for quasipoisson family taken to be 18.8901)

    Null deviance: 9591  on 999  degrees of freedom
Residual deviance: 9591  on 998  degrees of freedom
AIC: NA

Number of Fisher Scoring iterations: 6
```

As you can see – the significance of *x* went down and SE increased, but nothing else really has changed. Also, we can see that the dispersion parameter for Poisson distribution was simulated rather than fixed to one. Quasipoisson works out the exact value of the ratio of residual deviance to residual *df* and adjusts the estimates using this value. Still, however, the estimates are far from being correct. The reason for this is that quasipoisson uses multiplicative overdispersion model and it's obvious that – given the nature of our simulated linear predictor – omitted *z* had an additive influence on the response (we added *z* rather than multiplied by *z*; see Hadfield (2010a) for more details). Since this is the case in most biological data – we should try applying additive overdispersion models. A recently emerged alternative is to use Markov Chains (Bayesian methodology) rather than frequentists' approach of Maximum Likelihood. MC based methods are sometimes slower and cumbersome, but run long enough (with large enough number of Monte Carlo randomizations) they provide accurate and unbiased estimates, especially in case of "weird" distributions such as gamma-distributed responses (common in survival analysis) or zero-inflated Poisson responses. Let's try the power of MCMC in our case:

```
> install.packages("MCMCglmm")
> library(MCMCglmm)
> glmmcmc <- MCMCglmm(yp ~ x, family = "poisson", data = glmdata,
+ verbose=F)
> summary(glmmcmc)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 4489.607

 R-structure:  ~units

      post.mean l-95% CI u-95% CI eff.samp
units     1.462    1.260    1.628    878.9

 Location effects: yp ~ x

            post.mean l-95% CI u-95% CI eff.samp  pMCMC
(Intercept)   0.97338  0.79778  1.14919    879.8 <0.001 ***
x             0.08298 -0.20720  0.35077   1000.0  0.574
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As expected - our estimates are much closer to their theoretical values. Moreover – **MCMCglmm** does not assume or simulate any value for residual variance. **MCMCglmm** knows, that it should fix the residual variance as it cannot be estimated in case of Poisson distribution. So why there's some value next to **units** (which stands for residuals in **MCMCglmm**)? As a matter of fact – looking into the object containing estimates of variances (**$VCV**) for random effects (here it's only residual variance) allows for analysing this value more thoroughly. Here we'll use two functions – one uses mode value of the posterior distribution as an estimate of variance; the second uses highest posterior density to calculate desired percentile range, by default it is 95% credible interval:

```
> posterior.mode(glmmcmc$VCV)
   units
1.459504
> HPDinterval(glmmcmc$VCV)
         lower    upper
units 1.259746 1.628112
attr(,"Probability")
[1] 0.95
```

The value of residual variance looks strange, but when you think where did the extra variation come from – you should quickly find the reason for that. Extra variation in our data comes from not including the *z* explanatory variable. It was generated using 1.5 as it's variance – and our residual variance estimate corresponds to that value.

Overview of modelling generic functions and attributes

In one of his books Crawley went as far as to say "Fitting models to data is the central function of R". It's hard not to agree with him when you realize the extent of functions making statistical modelling in R so easy and straightforward. The following table contains an overview of functions you can apply to model objects as well as the most important arguments used together with formulae in nearly all modelling functions. Functions are listed first, than arguments.

Table 8: Functions and arguments used in statistical modeling

`fitted()`	Provides fitted values for predictors provided to the modelin gfunction
`resid()`	Returns residuals from the model
`predict(0`	Predicts new values for provided x values
`AIC()`	Returns AIC score (only for ML/REML methods; not applicable for quasilikelihood and Bayesian methods)
`plot()`	Diagnostic plots
`update()`	Updates the model object
`coef()`	Returns estimated coefficients of the model
`anova()`	Compares two model objects (some modelling pckages are not compatible in this sense, e.g. one cannot compare lmer and glm objects)
`summary.aov()`	Returns ANOVA table for model fitting
`summary.lm()`	Returns regression coefficients for model fitting
`subset(data_set, LOGICAL)`	Returns subset of the data_set satisfying the LOGICAL condition
`na.action`	What should be done with `NAs`? Default is `na.fail` (returns error); other possibilities are `na.omit` or `na.exclude` (affects the way df are calculated for RSS – see chapter 3); does not work with MCMCglmm (use `data=na.omit(your_data)` instead)
`weight`	Provides vector object for weighting the residuals; here what is minimized in OLS is weighted sum of squared residuals
`data`	Defines the data object
`family`	Defines the error distribution (in generalized models)
`prior`	In MCMCglmm – defines prior distributions of random effects

`mev`	In MCMCglmm – provides measurements errors for meta-analysis
`ped`	In MCMCglmm – provides pedigree for animal model
`random`	In MCMCglmm – formula for random effects
`saveX, saveZ`	In MCMCglmm – savesthe design matrices for random fixed/random effects, respectively

You should also remember that you can always invoke type-control functions, such as **`attributes(model_object)`** to see what type of data are contained within such item. To get to some particular "piece" of model object we have fitted just use the dollar operator: **`model_object$attribute_name`**.

Part 6 ~ GLMM

Overview

In case of (G)LMs we are treated all model terms as fixed effects. However in biology we often fail to specify any particular effect with our treatment/grouping factors. Things such as populations, plots, nests are assumed to generate variability in our data through the simple fact that they're drawn at random from some large population. In case of fixed effects any change in the response variable comes from the data associated with a specified, particular level of this effect whereas in random effects some information comes from data in a particular level, but we also weight the outcome using data from other levels of random effect, taking into account also likelihood this effect could take other values. In terms of mathematical implementation we simply treat variance associated with fixed effects as very large (infinite) and are interested in the overall effect of a treatment/factor on the mean of our response. In case of random effects we estimate this variance precisely and are interested not in the means associated with particular levels but in the overall variability introduced in our data by these levels. A good rule of thumb is to treat effects with informative levels as fixed and those with non-informative levels (such as grouping factors, replication units, blocks etc.) as random (Crawley 2010).

In R there are several possibilities of fitting mixed models (implementing both fixed and random effects). These methods vary according to their performance, output format, time required to complete the analysis and data they accept. In the majority of the cases we will use either **`lmer`** from **`lme4`** package or **`MCMCglmm`** from **`MCMCglmm`** package (Hadfield, 2010a). Both fit generalized (possible non-normal response) and random effects, they just differ in the way they achieve that. The former uses classical REML method, whereas the latter allows for incorporating additional information in the Bayesian framework and uses Monte Carlo simulations and Markov Chains. Is there any difference in their reliability or effectiveness? Well – there is. First of all – **`lmer`** performs well only for Gaussian data. If you'd like to analyse non-normal data, especially using distributions where variance should not be estimated (it equals the mean; e.g. (zero-inflated)Poisson, binomial), choose **`MCMCglmm`**. Real data are often overdispersed (see discussion in the previous chapter) due to effects we didn't account for and `quasi` distributions in **`lmer`** – that should deal with overdispersion – somehow fail to do this correctly. Maybe it's because of the 'under development' status `lme4` is still having. Secondly – when fitting categorical random interactions, the

better choice is **MCMCglmm** because it allows for fitting factor-specific residual variances; if not accounted for and present, such differences in residual variance would be likely to be confounded with the variance of factor. Third - comparing models with non-normal response using likelihood ratio tests or AIC should be treated with caution – in such cases likelihood is not calculated but approximated. **MCMCglmm** provides simple DIC-based system for selecting competing models. Of course – **MCMCglmm** has its drawbacks: for large data and complex models calculations may be slow and tedious – nevertheless they provide reliable estimates, especially for "weird" distributions.

Simple mixed model and why it should be mixed?

To see mixed effects in action let's simulate simple data. Imagine you've measured tail lengths in birds from 20 plots in the forest, 5 birds per plot (after Davey 2009). You'd like to know the effect these plots have on your data.

```
> plot <- as.factor(LETTERS[rep(1:20,5)]) #generate plots
> ploteff <- rnorm(20, mean=0, sd=sqrt(50)) #effects of the plots
> lin <- 100 + ploteff[plot] #form predictor with intercept of 100
> tail <- rnorm(100, mean=lin, sd=sqrt(50)) #generate data from normal
> mydata <- data.frame(plot = plot, tail = tail)
```

If we treated plots as fixed effect – here the best estimate of this effects would be simply the mean for each plot:

```
> tapply(tail, plot, mean) #apply some function to data by groups
       A        B        C        D        E        F
 94.88551 104.19274  95.84284 104.09997 105.07422  88.30977
       G        H        I        J        K        L
 87.14866 115.38389  98.40378  92.86162 116.21911  89.03037
       M        N        O        P        Q        R
 86.91065 100.27507 108.72091 103.11933 111.70116  92.39619
       S        T
 99.91879 109.81192
```

This should give the same results as fitting a linear model:

```
> prfixed <- glm(tail ~ plot, data = mydata)$coef
> prfixed
(Intercept)       plotB       plotC       plotD       plotE       plotF
 96.2846690  -1.6781011   6.2437841  11.8258226   0.5314776   6.4617477
      plotG       plotH       plotI       plotJ       plotK       plotL
-11.8751458  -9.9479240  -1.1035683  17.4540470   2.0472615   4.4722584
      plotM       plotN       plotO       plotP       plotQ       plotR
  2.6331821  -6.2494063  -1.4984548  20.6336267  -6.0521353   8.6533025
      plotS       plotT
 -4.4806945   7.2699813
> prfixed <- glm(tail ~ plot-1, data = mydata)$coef
```

```
> prfixed
    plotA     plotB     plotC     plotD     plotE     plotF     plotG
 96.28467  94.60657 102.52845 108.11049  96.81615 102.74642  84.40952
    plotH     plotI     plotJ     plotK     plotL     plotM     plotN
 86.33674  95.18110 113.73872  98.33193 100.75693  98.91785  90.03526
    plotO     plotP     plotQ     plotR     plotS     plotT
 94.78621 116.91830  90.23253 104.93797  91.80397 103.55465
```

In the first case we forgot that in modelling functions, for categorical effects we'll get the Intercept and deviations from it; thus we've subtracted 1 from the formula to get the means. How it compares to the situation in which we treat plots as random effect? We'll use **`lmer`** (note that in lmer fixed effects are defined in a standard way whereas random effects enter through a | operator and in parentheses – the meaning of this will be explained later):

```
> install.packages("lme4")
> library("lme4")
> rantail <- lmer(tail~1+(1|plot), data=mydata)
> summary(rantail)
Linear mixed model fit by REML
Formula: tail ~ 1 + (1 | plot)
   Data: mydata
   AIC   BIC logLik deviance REMLdev
 706.5 714.3 -350.3    703.6   700.5
Random effects:
 Groups   Name        Variance Std.Dev.
 plot     (Intercept) 62.146   7.8833
 Residual             44.362   6.6605
Number of obs: 100, groups: plot, 20

Fixed effects:
            Estimate Std. Error t value
(Intercept)   98.552      1.884    52.3
```

Note that here we use `@` instead of `$` to access specific information from our model (here we extract fixed and random effects coefficients for values fitted by our model):

```
> prrand <- rantail@ranef+rantail@fixef
> prrand
 [1] 96.56778  95.09925 102.03183 106.91678  97.03289 102.22258  86.17562
 [8] 87.86217  95.60203 111.84214  98.35938 100.48154  98.87213  91.09881
[15] 95.25646 114.62465  91.27145 104.14045  92.64664 102.92988
```

What's essential here is that the variance in our estimates is much higher when we treat these effects as fixed:

```
> var(prfixed)
```

```
[1] 71.01838
> var(prrand)
[1] 54.38815
```

Also, this variance for random version of our model is closer to the value (50) we've simulated at the beginning.

Let's move to something more realistic. We'll employ data on blue tits for tarsus lengths and back colours. Tits were cross-fostered and dam indicates real mother, whereas fosternest specifies nest of rearing. We're interested in the effects of dams and nest of rearing.

```
> data(BTdata)
> sikor <- lmer(tarsus~sex+(1|dam)+(1|fosternest),data=BTdata)
> summary(sikor)
Linear mixed model fit by REML
Formula: tarsus ~ sex + (1 | dam) + (1 | fosternest)
   Data: BTdata
  AIC  BIC logLik deviance REMLdev
 2087 2115  -1038     2065    2075
Random effects:
 Groups     Name        Variance Std.Dev.
 dam        (Intercept) 0.220259 0.46932
 fosternest (Intercept) 0.069204 0.26307
 Residual               0.567919 0.75360
Number of obs: 828, groups: dam, 106; fosternest, 104

Fixed effects:
            Estimate Std. Error t value
(Intercept) -0.40566    0.06706  -6.049
sexMale      0.76879    0.05714  13.455
sexUNK       0.21043    0.12670   1.661

Correlation of Fixed Effects:
        (Intr) sexMal
sexMale -0.449
sexUNK  -0.210  0.224
```

It's clear that both fosternest and dam explain significant proportion of variance in our data. Is that true? In case of mixed models tests using typical Wald-type statistics (such as *t*-Student test) are not proper. It's recommended to use likelihood ratio test. Fortunately, the **anova()** function appears to be handy here:

```
> sikor2 <- lmer(tarsus~sex+(1|dam),data=BTdata)
> anova(sikor,sikor2)
Data: BTdata
Models:
sikor2: tarsus ~ sex + (1 | dam)
sikor: tarsus ~ sex + (1 | dam) + (1 | fosternest)
       Df    AIC    BIC  logLik  Chisq Chi Df Pr(>Chisq)
sikor2  5 2086.7 2110.2 -1038.3
sikor   6 2077.1 2105.4 -1032.6 11.518      1  0.0006893 ***
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

R knows that here simple RSS (as in **lm()** and **aov()**) is not enough. As we can see both logLik and AIC are lower for the 2nd model (with both effects) – we can than conclude that the fosternest effect is significant here. Unfortunately, **lmer()** does not allow for models with no random effects (so that we could test dam vs. model with just an intercept). Here other functions such as **MCMCglmm** are much more useful.

Adding to our confusion, you actually cannot see the explicit tests for fixed effects in the model above. Unfortunately, the only thing **lmer** provides are the *t* values – or *F* values if one apply the **anova()** function. We can try testing our fixed effects using these values but remember to use proper degrees of freedom (Hadfield 2009). Probably most conservative *df* for residuals in our case would be the number of observations minus the number of levels for dams minus the number of levels for fosternest.

```
> tv<-summary(sikor2)@coefs[,3][2]
> df<-dim(BTdata)[1]-nlevels(BTdata$dam)-
+ nlevels(BTdata$fosternest)
> 2*(1-pt(tv,df))
sexMale
      0
> Fv<-anova(sikor2)[,4][1]
> 1-pf(Fv,2,df)
[1] 0
```

Important and so far unexplained thing is this mysterious **1|** in the specification of random effects. Vertical line means simply "random interaction". But what's interacting with what here? One stands for intercept – so you should read this definition as "see how much variation in intercept is introduced by the right-hand term (here dams or fosternests)". It's quite straightforward to introduce another fixed effect in our formula on the left side. Say, you want to see if dam effect is sex specific – i.e. the variance resulting from dams is sex-specific and maybe there's some significant covariance in dam effects between sexes. To see if that' the case, enter the following:

```
> siksex <- lmer(tarsus~sex+(sex-1|dam)+(1|fosternest),data=BTdata)
> summary(siksex)
Linear mixed model fit by REML
Formula: tarsus ~ sex + (sex - 1 | dam) + (1 | fosternest)
   Data: BTdata
  AIC  BIC logLik deviance REMLdev
 2097 2149  -1037     2065    2075
Random effects:
 Groups     Name        Variance Std.Dev. Corr
 dam        sexFem      0.227304 0.47676
            sexMale     0.209747 0.45798  1.000
            sexUNK      0.296946 0.54493  1.000 1.000
```

```
 fosternest (Intercept) 0.066462 0.25780
 Residual               0.568052 0.75369
Number of obs: 828, groups: dam, 106; fosternest, 104

Fixed effects:
            Estimate Std. Error t value
(Intercept) -0.40616    0.06738  -6.028
sexMale      0.77018    0.05714  13.479
sexUNK       0.19936    0.12878   1.548

Correlation of Fixed Effects:
        (Intr) sexMal
sexMale -0.470
sexUNK  -0.171  0.218
```

Now it's sex on the left hand of | to tell R that we want sex-specific effects of dams (or – how much variation in sexes is caused by dam effects; note that in some cases sex is unknown – UNK – but we did not remove such data from the dataset). We removed the intercept so that estimates will appear as sex-specific variances rather than deviations from variance for females (alphabetically first level of sex). Resulting (co)variance matrix is below; the order of columns and rows is alphabetical (females, males, UNK).

```
> matrix(VarCorr(siksex)$dam,3,3)
          [,1]      [,2]      [,3]
[1,] 0.2273043 0.2183493 0.2598021
[2,] 0.2183493 0.2097471 0.2495668
[3,] 0.2598021 0.2495668 0.2969462
```

It seems that the degree of sex-specific variation is not large but formal test should be used here. We could specify alternative (co)variance structures and compare models using likelihood ratio test (**`anova()`**) – I'll leave exploration to you. The only thing worth noting is that sometimes (co)variance definitions are not straightforward; the following table gives you most common ones with the way you could specify them in either **`lmer`** or **`MCMCglmm`**.

You can see that fitting random interactions may more complicated than writing **`A:B`**. Some of these interactions will become clearer in the next section, where we'll deal with multivariate mixed models. Also, note that in the left-hand side of | you can have not only categorical variables but also continuous ones; in such a case resulting (co)variance matrix will contain variances for random-effect specific intercepts and slopes and possibly also covariances between them. For more details please see the Advanced Issues section.

Speaking of **`MCMCglmm`** – fitting our blue tit model in **`MCMCglmm`** is a little bit tricky. **`MCMCglmm`** is Bayesian-based and thus it requires some additional information in form a prior. Prior is simply anything we know about our parameters (based on

experience or previous research) and we can use this information in our analyses. Sometimes researchers are concerned that in Bayesian statistics we incorporate some a priori information, possibly subjective. But fortunately we can use so-called non-informative priors, where we assume that we know little a priori and want (nearly)all the information to come from our data.

Table 9: Possible covariance structures in lmer and MCMCglmm (adapted from Hadfield 2010a)

`(1|dam)`	`dam`	$\begin{bmatrix} V & V & V \\ V & V & V \\ V & V & V \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
`(sex-1|dam)`	`us(sex):dam`	$\begin{bmatrix} V_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,1} & V_{2,2} & C_{2,3} \\ C_{3,1} & C_{3,2} & V_{3,3} \end{bmatrix}$	$\begin{bmatrix} 1 & r_{1,2} & r_{1,3} \\ r_{2,1} & 1 & r_{2,3} \\ r_{3,1} & r_{3,2} & 1 \end{bmatrix}$
`(1|sex:dam)`	`sex:dam`	$\begin{bmatrix} V & 0 & 0 \\ 0 & V & 0 \\ 0 & 0 & V \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
`(1|dam)+` `(1|sex:dam)`	`dam+` `sex:dam`	$\begin{bmatrix} V_1+V_2 & V_1 & V_1 \\ V_1 & V_1+V_2 & 0V_1 \\ V_1 & V_1 & V_1+V_2 \end{bmatrix}$	$\begin{bmatrix} 1 & r & r \\ r & 1 & r \\ r & r & 1 \end{bmatrix}$
–	`idh(sex):dam`	$\begin{bmatrix} V_{1,1} & 0 & 0 \\ 0 & V_{2,2} & 0 \\ 0 & 0 & V_{3,3} \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
–	`corh(sex):dam`	$\begin{bmatrix} V_{1,1} & rV_{1,1}V_{2,2} & rV_{1,1}V_{3,3} \\ rV_{1,1}V_{2,2} & V_{2,2} & rV_{3,3}V_{2,2} \\ rV_{1,1}V_{3,3} & rV_{3,3}V_{2,2} & V_{3,3} \end{bmatrix}$	$\begin{bmatrix} 1 & r & r \\ r & 1 & r \\ r & r & 1 \end{bmatrix}$
–	`cor(sex):dam`	$\begin{bmatrix} 1 & r_{1,2} & r_{1,3} \\ r_{2,1} & 1 & r_{2,3} \\ r_{3,1} & r_{3,2} & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & r_{1,2} & r_{1,3} \\ r_{2,1} & 1 & r_{2,3} \\ r_{3,1} & r_{3,2} & 1 \end{bmatrix}$

Priors or random are specified as lists, where there's one element called `R` (for residual variance) and (optionally, if needed), one or more elements called `G1`, **`G2`, `Gn`** pooled inside a **`G`** element. G-elements provide priors for other random effects. Inside every element we enter **`V`** (variance for our prior) and **`nu`** (so called belief parameter, often written as simple **`n`**); it's a good practise to set this parameter as **`(m-1)+0.001`** to obtain non-informative prior, where **`m`** is the number of response variables and

hence the dimension of (co)variance matrix; the greater the **n** value, the more we believe in the value **V** provided in the greater will be its influence on our results; negative priors are possible, however they result in so called improper prior distributions (essentially, not summing up to one) and should be used with care – I'll recommend reading Hadfield's notes from **MCMCglmm's** CRAN page for more detail both on improper priors and on ways of setting the).

```
> prsik <- list(R=list(V=1,n=0.001),G=list(G1=list(V=1,n=0.001),
+ G2=list(V=1,n=0.001)))
> sikormc <- MCMCglmm(tarsus~sex, random=~fosternest+dam,
+ data=BTdata, prior=prsik, verbose=F)
> summary(sikormc)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 1991.724

 G-structure:  ~fosternest

           post.mean l-95% CI u-95% CI eff.samp
fosternest   0.06684 0.002443   0.1208    381.7

              ~dam

    post.mean l-95% CI u-95% CI eff.samp
dam    0.2251   0.1408   0.3245     1074

 R-structure:  ~units

      post.mean l-95% CI u-95% CI eff.samp
units    0.5716   0.5151   0.6342    765.7

 Location effects: tarsus ~ sex

            post.mean l-95% CI u-95% CI eff.samp  pMCMC
(Intercept)   -0.4037  -0.5233  -0.2650   1000.0 <0.001 ***
sexMale        0.7705   0.6582   0.8846   1000.0 <0.001 ***
sexUNK         0.2133  -0.0189   0.4858    895.1    0.1
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Estimates of dam, fosternest and residual (here called **units**) variances are with good agreement with those from **lmer** (up to Monte Carlo error). Some practise is needed to interpret results from **MCMCglmm. summary()** provides confidence intervals for variance estimates (G-structure and R-structure) and **pMCMC** values for fixed (location) effects (interpreted as P values). We can visualise posterior distributions for fixed and random effects:

```
> plot(sikormc)
```

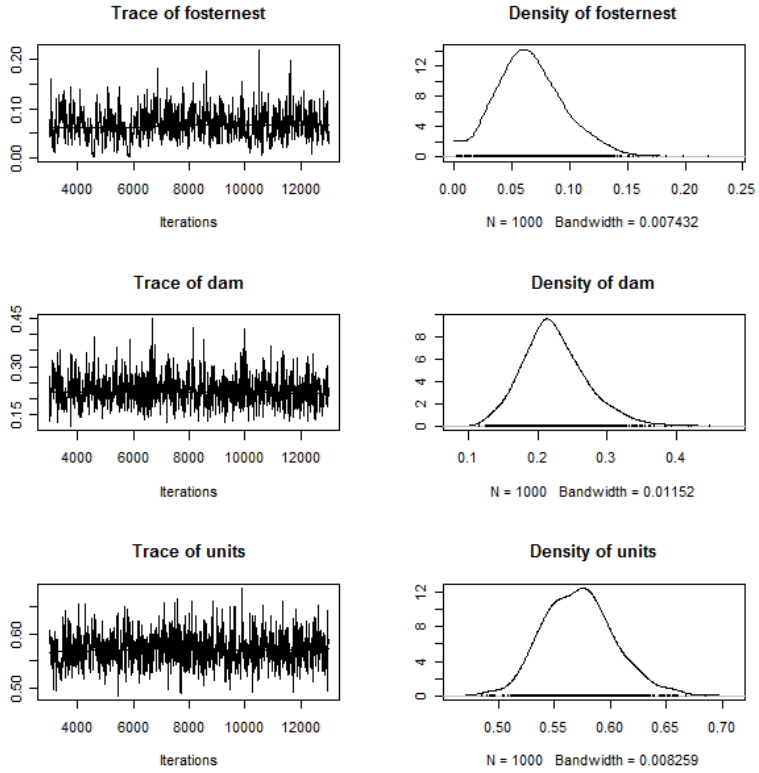


Fig 27: Posterior distributions of random effects

Here I show only the last page of plots. Each left-hand side plot shows the trace of time series (nr of iteration on x-axis and parameter values and y-axis) for random Monte Carlo samplings from the posterior distribution, and right-hand side plots show these estimated posterior distributions (as smoothed histograms). For all estimated parameters.

You can also access specifically fixed effects (**`sikormc$Sol`** object) and random variances (**`sikormc$VCV`**). Obtaining exact values of the expectations (mode

of the posterior distribution) and confidence intervals requires using special functions:

```
> posterior.mode(sikormc$VCV)
fosternest        dam      units
0.05505678 0.21926275 0.57147922
> HPDinterval(sikormc$VCV)
                lower     upper
fosternest 0.002442576 0.1207558
dam        0.140842844 0.3244714
units      0.515111574 0.6341997
attr(,"Probability")
[1] 0.95
```

One of great advantages of MCMC-based GLMMs is simplicity of using posterior distributions. If we e.g. wanted to calculate proportion of variance explained by dam effect – we can simply use whole distributions. Also – confidence intervals will be calculated that way, which circumvents many difficulties and approximation issues when calculating confidence interval for (co)variance ratios (e.g. heritabilities) in REML-based algorithms. Here, the only thing we have to know is the exact position of each random effect in VCV object (you can infer this from the above output; effects are in columns ordered in the same way as their modes in **posterior.mode** output, i.e. dam is in the second column):

```
> propdam <- sikormc$VCV[,2]/(sikormc$VCV[,1]+
+ sikormc$VCV[,2]+sikormc$VCV[,3]) #var for dams over sum of all vars
> posterior.mode(propdam)
     var1
0.2670148
> HPDinterval(propdam)
         lower     upper
var1 0.1731281 0.3452775
attr(,"Probability")
[1] 0.95
```

Try to specify (using table with different random interaction definitions) the same interactions we've examined in **lmer()**. Also, see if you can extract all the relevant information (variances and covariances) from **$vcv** object.

Animal model

In a simple mixed model each random effect has it's associated variance structure. Generally we assume that different levels of the random effect are independent from each other, and data within each level are identically distributed, implying homoscedascity of variances (i.e. random effects are i.i.d. – independent and

identically distributed). In terms of a linear model parameters, random effects are assumed to come from a multivariate normal distribution with mean 0 and defined (co)variance structure, i.e. $\sigma_R^2\mathbf{I}$ (**I** is the identity matrix: I[*i*,*j*]=0 for $i \neq j$ and 1 gdy $i=j$). However, in some cases we cannot assume independence of different levels of a random effect, i.e. if we're dealing with genotypes – the way they influence variability in the response may depend on the extent of their genetic relatedness. Thus, we might won't substitute zeros in **I** using some measures if dependence; in the simplest case we might use coefficients of relatedness. Thus we define such (co)variance structure as $\sigma_A^2\mathbf{A}$, where **A** is a square matrix with first two columns/rows for parents, the remaining *i*-2 rows/*j*-2 columns for their offspring; A[*i*,*j*] is the relatedness coefficient between *i*-th and *j*-th individual. This type of variance structure allows for estimating additive genetic variance and narrow-sense haritabilities. In R – there's currently one well tested possibility of fitting animal models – the **MCMCglmm** package – and given its versatility I wouldn't suggest any other. Both **glmmPQL** and **lmer** (see Statistical Modelling for details) can fit animal models, but it requires some additional steps with defining (co)variance structures – which you can avoid in **MCMCglmm** by simply providing your pedigree structure to the function. So unless you want use REML or PQL for some special reasons, I recommend using **MCMCglmm** – especially in case of non-gaussian data.

We'll begin with simple, normally distributed data: tarsus length of blue tits.

```
> data(BTdata)
> data(BTped)
```

Look carefully at the structure of `BTped` file:

```
> head(BTped)
   animal  dam sire
1 R187557 <NA> <NA>
2 R187559 <NA> <NA>
3 R187568 <NA> <NA>
4 R187518 <NA> <NA>
5 R187528 <NA> <NA>
6 R187945 <NA> <NA>
```

Important thing is that it contains **all** individuals (parents and offspring) in the first columns, their mothers in the second column and its fathers in the third one. As you can see – first individuals have no parents assigned – which is obvious as we have some clearly defined first generation.

Inside the **MCMCglmm** function we incorporate pedigree data using `ped` argument. The list of random effects must also contain a special effect, with restricted name animal:

```
> BTgen <- MCMCglmm(tarsus~sex, random=~animal+fosternest,
+ ped=BTped, data=BTdata, verbose=F)
Warning message:
In MCMCglmm(tarsus ~ sex, random = ~animal + fosternest, ped = BTped,  :
  some combinations in animal do not exist and 212 missing records have been generated
```

Note that we did not use any priors – our data are well structured, with no apparent imbalance and **MCMCglmm** can handle this analysis with its internally default priors. Let's see the results for random effects, mainly the genetic effect:

```
> posterior.mode(BTgen$VCV)
    animal fosternest      units
 0.4542497  0.0824894  0.3676182
> HPDinterval(BTgen$VCV)
                lower     upper
animal     0.286807024 0.6328057
fosternest 0.008939443 0.1284861
units      0.232745745 0.4435488
attr(,"Probability")
[1] 0.95
> plot(BTgen$VCV)
```

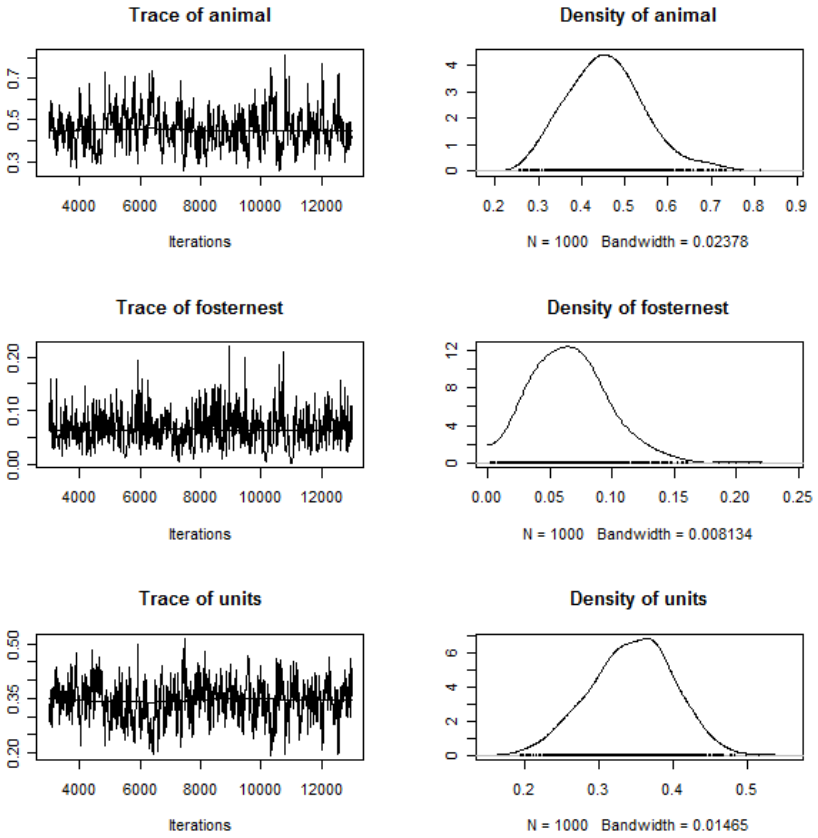


Fig 28: Random effects form an animal model

As you can see – animal explains a significant proportion of variance: its posterior distribution lies well over the zero value. But the strict test of its significance can be obtained only by using DIC:

```
> BTgen$DIC
[1] 1840.246
> BTnogen$DIC
[1] 2122.019
```

The DIC value for the genetic model is lower and thus – animal explains a significant proportion of variance in tarsi. With this information we can easily proceed to calculation of heritabilities. It simply requires adequate manipulation of posterior distributions. Note, that here it's important to provide correct column numbers for successive random effects:

```
> h2tar <- BTgen$VCV[,1]/(BTgen$VCV[,1]+BTgen$VCV[,2]+BTgen$VCV[,3])
> posterior.mode(h2tar)
     var1
0.5641558
> HPDinterval(h2tar)
         lower    upper
var1 0.3594623 0.680278
attr(,"Probability")
[1] 0.95
```

Another genetic parameter of interest is genetic correlation. We can easily estimate it using the same pedigree data – the only change is the use of appropriate (co)variance functions and extension of our univariate model to a bivariate one. If – for univariate model – the (co)variance structure is $\sigma_R^2\mathbf{I}$, in the bivariate case it will be $\mathbf{V}\otimes\mathbf{I}$ where $\mathbf{V}$ is a $m\times m$ square matrix (with m being the number of response variables):

$$\mathbf{V} = \begin{bmatrix} \sigma_1^2 & \sigma_{1,2}^2 \\ \sigma_{1,2}^2 & \sigma_2^2 \end{bmatrix}$$

Once again, for the estimation of genetic correlations we define an additive genetic effect $\mathbf{a}\sim\mathcal{N}(0,\mathbf{V}\otimes\mathbf{A})$. Here essential is that default priors are univariate. Thus, to proceed with the analysis we have to define m-variate priors (most often it will be just a square diagonal matrix, hence providing variances equal to one and covariances equal to zero). Remember to set the belief parameter to m-1+0.001 to obtain least informative but still proper prior (but see Hadfield (2010b) on more specific indication as to how to use belief parameters in complex covariance structures):

```
> prBT <- list(R=list(V=diag(2),n=1.001),
+ G=list(G1=list(V=diag(2),n=1.001),
+ G2=list(V=diag(2),n=1.001)))
```

Note, that we should have two elements in the G-structure because we have two random factors beside the residual (R-structure).

Calling the **MCMCglmm** – remember to deal with several things: (i) for each random effect decide whether to fix trait-specific variances or just single variance for pooled trait values; most often the former would be biologically more relevant and correct; (ii) decide whether to fit random effect covariances between traits or fix them to zeros; here you should use either **idh()** or **us()** variance functions, the former assuming zero covariance, the latter estimating its precise value(s) – see below; (iii) remember to use **rcov** argument to define appropriate residual (co)variance

structure; remember that if each individual was measured only in one trait – residual covariance cannot be estimated and should be fixed at zero (which is common when estimatin cross-sex r_G or analysis genotype-by-environment interaction); still – you should fit residuals as two trait-specific variances to allow for differences in σ_e^2 between traits; (iv) finally, use **`cbind()`** to pass both response variables to **`MCMCglmm`**. Below we'll look at the genetic correlation between tarsus length and back colour in blue tits:

```
> BTgenr <- MCMCglmm(cbind(tarsus,back)~sex,
+ random=~us(trait):animal+us(trait):fosternest,
+ rcov=~us(trait):units, data=BTdata, ped=BTped,
+ verbose=F, nitt=50000, thin=100, burnin=10000,
+ prior=prBT, family=c('gaussian','gaussian'))
Warning message:
In MCMCglmm(cbind(tarsus, back) ~ sex, random = ~us(trait):animal +  :
  some combinations in us(trait):animal do not exist and 212 missing records have been
generated
```

Since genetic covariance isn't constrained by zero lower boundary it's test using confidence interval will be valid. Alternatively me might use DIC to compare our model with the in which we define animal as **`idh(trait):animal`**, hence preventing covariance from being estimated. Here we'll proceed with confidence-based test:

```
> HPDinterval(BTgenr$VCV)
                              lower      upper
tarsus:tarsus.animal      0.28878224 0.66357759
back:tarsus.animal       -0.20459648 0.01906264
tarsus:back.animal       -0.20459648 0.01906264
back:back.animal          0.09297079 0.31168575
tarsus:tarsus.fosternest  0.07671264 0.20046810
back:tarsus.fosternest   -0.00864343 0.09907748
tarsus:back.fosternest   -0.00864343 0.09907748
back:back.fosternest      0.08682076 0.22635957
tarsus:tarsus.units       0.25140748 0.47979075
back:tarsus.units        -0.10522497 0.05454153
tarsus:back.units        -0.10522497 0.05454153
back:back.units           0.63802316 0.85200243
attr(,"Probability")
[1] 0.95
> posterior.mode(BTgenr$VCV)
    tarsus:tarsus.animal       back:tarsus.animal
              0.41552815              -0.05264475
      tarsus:back.animal         back:back.animal
             -0.05264475               0.17940797
tarsus:tarsus.fosternest   back:tarsus.fosternest
              0.11540324               0.04211067
  tarsus:back.fosternest     back:back.fosternest
              0.04211067               0.13240551
     tarsus:tarsus.units        back:tarsus.units
              0.34843877              -0.03127438
       tarsus:back.units          back:back.units
```

```
-0.03127438                0.74120317
```

Genetic covariance for analysed traits (**tarsus:back.animal** or **back:tarsus.animal**) is negative and its confidence interval does overlap zero. Thus, we conclude that it is not significantly different from zero and we might as well use **idh()** covariance structure. Check that it's the case also for other random effects.

Calculation of genetic correlation is as straightforward as it was in case of heritability. Putting appropriate (co)variance parameters into the formula for r_G (covariance divided by square root of variances' product) allows for its calculation. We obtain a complete posterior distribution of r_G allowing for easy construction of confidence intervals and hypothesis testing:

```
> posterior.mode(rG)
     var1
-0.232888
```

```
> HPDinterval(rG)
          lower      upper
var1 -0.6142401 0.06382528
attr(,"Probability")
[1] 0.95
```

Once again, although tarsus and back colour seem to be negatively correlated at the genetic level, this correlation is not significantly different from zero.

Here important thing should be emphasized. In case of different traits, the H_0 should be that r_G=0; also, we should always fit bot variances, i.e. use either **us()** or **idh()**. However, when testing for r_G in one trait but in different sexes or environments (hence, testing for genotype-by-sex or –environment interactions) we should rather test two different hypotheses: H_{01}: r_G=1 (as could be expected due to shared genetic background of our individuals) and H_{02}: $\sigma_1^2 = \sigma_2^2$. Here, when the former fails to be rejected, we could use **idh()** or **us()**, but when both are accepted – simple univariate specification should be used, assuming equal variances and covariance one. Importantly – there's no simple way of constraining our (co)variance structure so that genetic correlations where equal to one; it's possible by use of **sir()** in so called simultaneity and recursion analysis in **MCMCglmm** but I'll not cover this here (the problem is due to the character of r_G – if it's equal to zero, covariance is also fixed at zero; but for r_G=1 – there's no unique value of covariance satisfying this equality and rather it depends on the values of variances).

Correlations in non-gaussian data

Real power of **MCMCglmm** pops out when dealing with non-normal data. MCMC deals with such data much better than REML – both simulations and mathematical considerations confirm that estimates of random effects are much more reliable in reasonably designed MCMC analyses. As an example we'll go through estimating genetic effects in two traits: oxidase activity (one of immune system branches) and virus resistance (measured as probability of pupation). Data come from an experiment on Indian meal moth (*Plodia interpunctella*) exposed to its specific granulosis virus, PiGV (Tidbury & Boots, unpublished data). Several difficulties arise in such data (Hadfield 2010b): (i) resistance is measured by probability of pupation (i.e. data points are 1 for pupated larvae and 0 for unpupated larvae) and hence follow binomial distribution in which residual variance cannot be estimated – it's a quadratic function of the mean; as pointed in the previous chapter, overdispersion often arises in such data; (ii) both traits have different distributions (binomial and Gaussian) which is difficult to handle in most modelling packages if we'd like to estimate r_G; (iii) measuring oxidase activity kills an individual thus preventing it from pupating – so none can be measured in the same trait (residual covariance cannot be estimated).

Measuring genetic effects in our data set won't require pedigree file – here, each individual has assigned family from which it comes from. In this case individuals are full-sibs, sharing 50% of their genes. Multiplying estimates of proportion of family variance in the overall variance should thus give us approximate estimation of heritability – although it will be broad-sense heritability since we cannot be sure that – excluding additive genetic effects - no other factors influence variability among families. However, since in binomial data residual variance cannot be estimated, family effects will constitute the only source of variation in random effects. Described way of calculating H^2 will be possible only for oxidase-trait. Try fitting such model by yourself.

We can however look at some approximation of genetic effects, as expressed by family effect. Let's start with simple estimation of family effect in pupated/not-pupated trait:

```
> data(PlodiaPO)
> data(PlodiaR)
> data(PlodiaPR)
> plobin <- MCMCglmm(cbind(Pupated,Infected)~1,
+ family="multinomial2", data=PlodiaR, verbose=F)
```

As mentioned before – residual variance cannot be estimated here. Moreover – since data are counts in families, family effects would be confounded with residuals. Thus we've decided to exclude **FSfamily** from analyses. Obviously (recall our models in the previous chapter for Poisson data) this will cause overdispersion (it will be put into units) – and since we know the source of this overdispersion, we can simply take residual variance as the estimate of family effect. Plotting this variance shows it's considerable:

```
> plot(plobin$VCV)
```

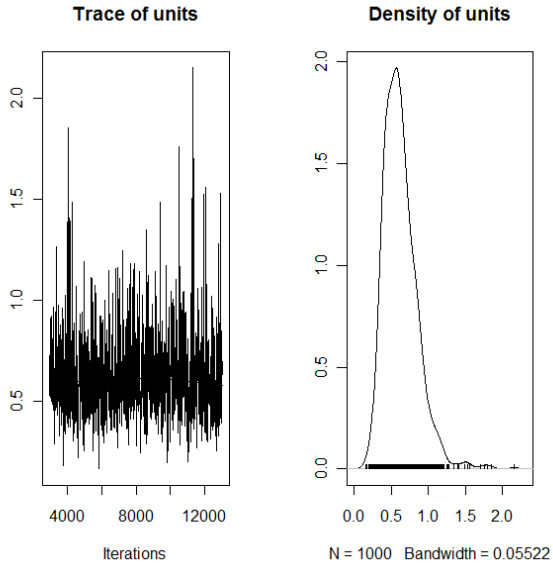


Fig 29: Residual variance from simple Plodia model

Here we've used **multinomial2** as the distribution family. However, we could express our data not as counts in families but as 0/1 data for single individuals. In such case we would have a special case of binomial data, where only one "object" is sampled in every trial (as in PlodiaRB). For such data, family **categorical** is appropriate:

```
> prplob <- list(R=list(V=1,n=0,fix=1),G=list(G1=list(V=1,n=0.001)))
> plocat <- MCMCglmm(Pupated~1, random=~FSfamily,
+ family="categorical", data=PlodiaRB, prior=prplob,verbose=F)
```

Here we can estimate separately **FSfamily** – now it won't be confounded with residuals (we have several individuals in each family). However, we still cannot estimate residual variance. Thus, we are fixing this variance using **fix** in our prior. Try exploring the results on your own.

As for family-level correlation between two traits we've considered so far – it should now be quite simple. We should remember to use proper distribution. First however, we should merge the data on oxidase activity and pupation together by generating for all individuals that were measured in one trait NA – remember they cannot be measured in both traits:

```
> PlodiaPO$ID <- 1:dim(PlodiaPO)[1]
> PlodiaRB$ID <- dim(PlodiaPO)[1]+1:dim(PlodiaRB)[1]
> PlodiaRBPO <- merge(PlodiaPO,PlodiaRB,all=T)
```

We should use prior that will be bivariate (2×2 square matrices) and that will fix proper residual variances (Hadfield 2010b):

```
> prplor <- list(R=list(V=diag(2),n=0,fix=1),
+ G=list(G1=list(V=diag(2),n=1.001)))
> plorg <- MCMCglmm(cbind(Pupated,PO)~trait-1,
+ random=~us(trait):FSfamily, rcov=~idh(trait):units,
+ family=c("categorical","gaussian"),data=PlodiaRBPO,
+ prior=prplor, verbose=F)
> rG <- plorg$VCV[,2]/sqrt(plorg$VCV[,1]*plorg$VCV[,4])
> plot(rG)
```

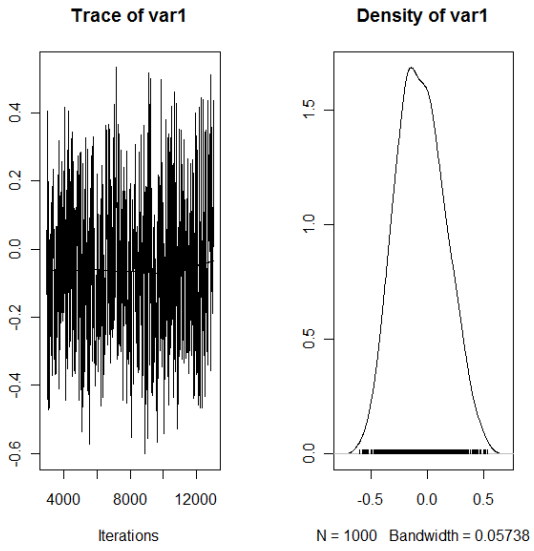


Fig 30: Family-level correlation in Plodia analysis

It seems that – at the family level – both traits are not correlated.

Part 7 ~ (Very brief) introduction to multivariate methods

Multivariate statistics serves as a tool for exploring structures in your data. It does not test any particular hypotheses – it rather lets you look into your data for patterns and structuring. It is important to use multivariate statistical methods described here only when you know exactly what you're doing. It is tempting to put your data into some complex algorithms and get some tricky outputs. But only when you know the aim of your analyses you will be able to extract the most of them.

Principal Component Analysis

PCA is a way of seeking for correlations in explanatory variables. It is designed to look for a set of standardized orthogonal (independent) linear combinations of the variables that explain all the variation in our data set. In other words, all variation in our predictors is turned into a set of *n* (*n* – number of variables) principal components. Most often first two-three components explain most of the variation. Here we'll try to extract variation components from an experiment where on 89 plots 54 plants were quantitatively examined (see Crawley 2010). Before doing PCA we must remove several continuous variables not associated with these species measurements:

```
> gatdane <- read.table("pgfull.txt",head=T)
> names(gatdane)
 [1] "AC"      "AE"      "AM"      "AO"      "AP"      "AR"      "AS"      "AU"
 [9] "BH"      "BM"      "CC"      "CF"      "CM"      "CN"      "CX"      "CY"
[17] "DC"      "DG"      "ER"      "FM"      "FP"      "FR"      "GV"      "HI"
[25] "HL"      "HP"      "HS"      "HR"      "KA"      "LA"      "LC"      "LH"
[33] "LM"      "LO"      "LP"      "OR"      "PL"      "PP"      "PS"      "PT"
[41] "QR"      "RA"      "RB"      "RC"      "SG"      "SM"      "SO"      "TF"
[49] "TG"      "TO"      "TP"      "TR"      "VC"      "VK"      "plot"    "lime"
[57] "species" "hay"     "pH"
> gat<-gatdane[,1:54]
```

We then use the whole dataset as an input to PCA. Here we have to scale our data to equalize variances in different predictors:

```
> pcagat <- prcomp(gat,scale=T)
> summary(pcagat)
Importance of components:
                          PC1    PC2     PC3    PC4     PC5     PC6
```

```
Standard deviation     3.005 2.336 1.9317 1.786 1.7330 1.5119
Proportion of Variance 0.167 0.101 0.0691 0.059 0.0556 0.0423
Cumulative Proportion  0.167 0.268 0.3373 0.396 0.4520 0.4943
                          PC7    PC8    PC9  PC10   PC11   PC12
Standard deviation     1.5088 1.3759 1.3244 1.273 1.2195 1.1979
Proportion of Variance 0.0422 0.0351 0.0325 0.030 0.0275 0.0266
Cumulative Proportion  0.5365 0.5716 0.6040 0.634 0.6616 0.6882
                         PC13   PC14   PC15   PC16   PC17   PC18
Standard deviation     1.1723 1.1355 1.0931 1.0678 1.0057 0.9550
Proportion of Variance 0.0254 0.0239 0.0221 0.0211 0.0187 0.0169
Cumulative Proportion  0.7136 0.7375 0.7596 0.7807 0.7995 0.8164
                         PC19   PC20   PC21   PC22   PC23   PC24
Standard deviation     0.9185 0.8947 0.8644 0.8497 0.7690 0.7513
Proportion of Variance 0.0156 0.0148 0.0138 0.0134 0.0109 0.0104
Cumulative Proportion  0.8320 0.8468 0.8606 0.8740 0.8850 0.8954
                         PC25    PC26    PC27    PC28    PC29
Standard deviation     0.7419 0.70653 0.69475 0.67325 0.62565
Proportion of Variance 0.0102 0.00924 0.00894 0.00839 0.00725
Cumulative Proportion  0.9056 0.91485 0.92379 0.93218 0.93943
                          PC30    PC31    PC32    PC33    PC34
Standard deviation     0.56800 0.56269 0.53857 0.52670 0.49524
Proportion of Variance 0.00597 0.00586 0.00537 0.00514 0.00454
Cumulative Proportion  0.94540 0.95127 0.95664 0.96177 0.96632
                          PC35    PC36    PC37  PC38    PC39
Standard deviation     0.48706 0.46638 0.44471 0.402 0.37661
Proportion of Variance 0.00439 0.00403 0.00366 0.003 0.00263
Cumulative Proportion  0.97071 0.97474 0.97840 0.981 0.98403
                          PC40    PC41    PC42    PC43    PC44
Standard deviation     0.35794 0.34381 0.31452 0.29639 0.26849
Proportion of Variance 0.00237 0.00219 0.00183 0.00163 0.00133
Cumulative Proportion  0.98640 0.98859 0.99042 0.99205 0.99338
[clipped…]
```

First PC explains over 16% of total variability in our predictors. Next one explains over 10%. To see relative contributions you can use so called cliff-plot (consecutive bars indicate percentages of total variance explained by respective PCs):

```
> plot(pcagat)
```

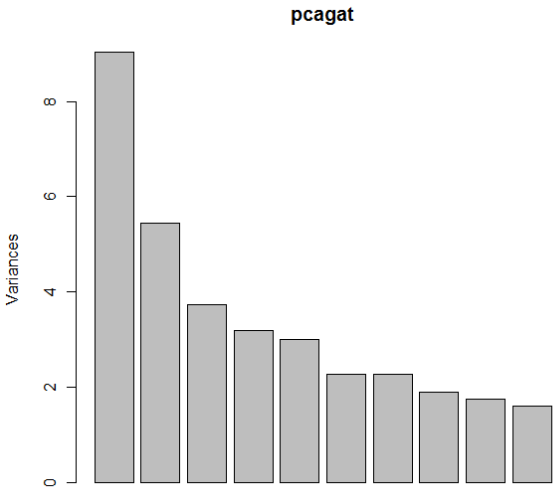


Fig 31: Cliff-plot

An usual way of expressing PCA analysis is a biplot. Here, each explanatory variables is represented by an arrow on the plane (or in the space) built on several (usually two – PC1 and PC2 – and thus plane) components. Lengths and directions of arrows indicate the magnitude and sign of contribution of each predictor (so called load) to the considered PCs:

```
> biplot(pcagat)
```

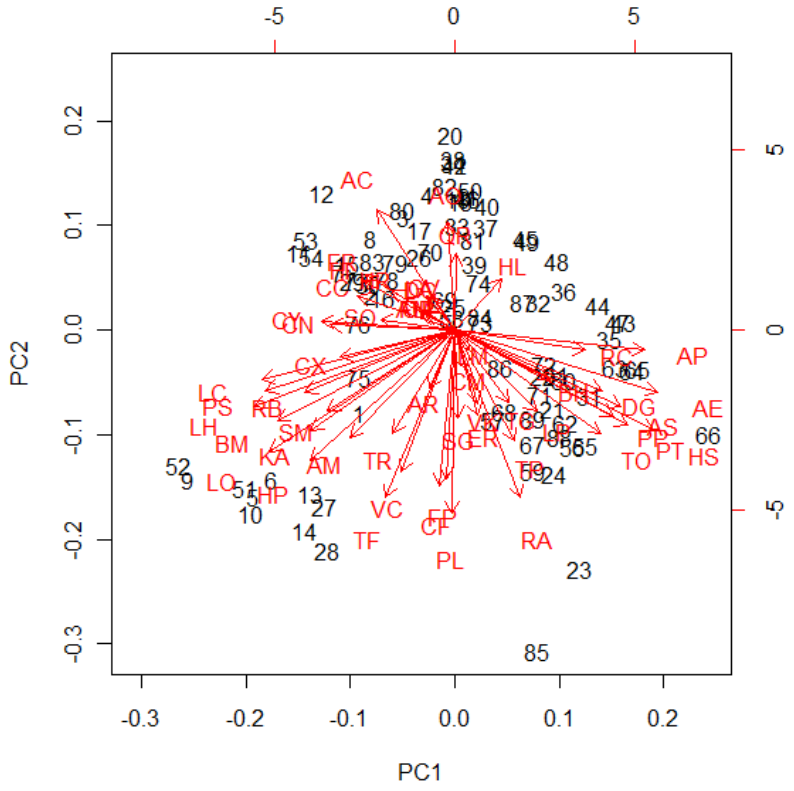


Fig 32: A biplot

Here, for instance, species AP, AE and HS have large, positive influence on the first (x-axis) PC. You can actually see how this "new" predictor (PC1) correlates with our response variables, e.g. with hay mass:

```
> plot(predict(pcagat)[,1],gatdane$hay,xlab="PC1",ylab="hay")
```

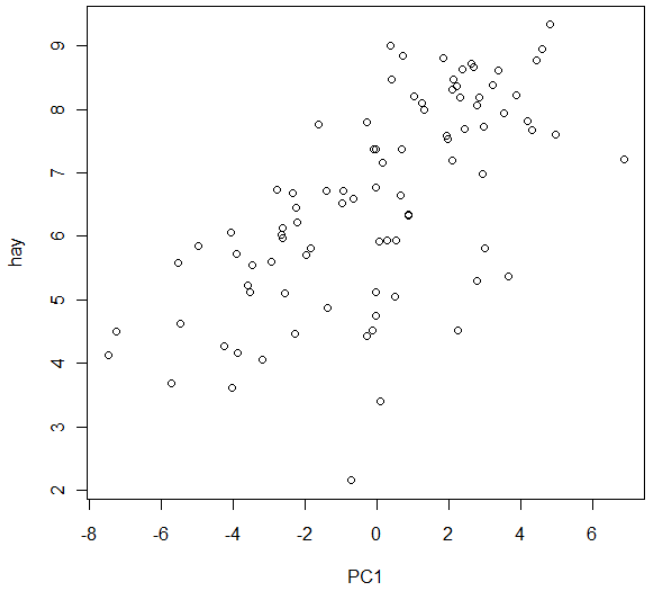


Fig 33: Using PC in further analyses

Factor analysis

Sometimes you're not interested in building some artificial variables but rather want to see how your measured variables contribute to some broader, unmeasured (or unmeasurable) variables, such as intelligence, fitness etc. Here you specify how many compound variables you'd like to have (and these are called factors). Let's analyse the same dataset, this time we'll aim at searching for eight factor variables:

```
> factanal(gat,8)

Call:
factanal(x = gat, factors = 8)

Uniquenesses:
   AC    AE    AM    AO    AP    AR    AS    AU    BH    BM    CC
0.638 0.086 0.641 0.796 0.197 0.938 0.374 0.005 0.852 0.266 0.056
   CF    CM    CN    CX    CY    DC    DG    ER    FM    FP    FR
0.574 0.786 0.579 0.549 0.733 0.837 0.408 0.072 0.956 0.371 0.815
   GV    HI    HL    HP    HS    HR    KA    LA    LC    LH    LM
0.971 0.827 0.921 0.218 0.332 0.915 0.319 0.305 0.349 0.333 0.927
   LO    LP    OR    PL    PP    PS    PT    QR    RA    RB    RC
```

```
0.121 0.403 0.005 0.286 0.606 0.336 0.401 0.913 0.491 0.005 0.754
   SG    SM    SO    TF    TG    TO    TP    TR    VC    VK
0.341 0.212 0.825 0.428 0.476 0.469 0.309 0.611 0.651 0.170

Loadings:
   Factor1 Factor2 Factor3 Factor4 Factor5 Factor6 Factor7 Factor8
AC -0.512  -0.268                          0.121
AE  0.925  -0.107          -0.146          -0.118
AM -0.206   0.413   0.213           0.163   0.115   0.153   0.186
AO -0.312  -0.196  -0.151  -0.105          -0.148  -0.102
AP  0.827  -0.173  -0.195  -0.167          -0.123
AR          0.150           0.111                  0.127
AS  0.778
AU                                                         0.996
BH  0.380
BM -0.116   0.292           0.695                  0.380
CC -0.152                   0.159           0.943
CF          0.539                   0.342
CM                  0.434  -0.110
CN -0.276   0.143                           0.541   0.147
CX                          0.628           0.169   0.146
CY -0.211          -0.162   0.340                  0.270
DC         -0.125                           0.372
DG  0.738                  -0.127           0.145
ER                                  0.960
FM -0.108                                   0.133
FP  0.245   0.226           0.478   0.493          -0.176
FR -0.386          -0.144
GV -0.134
HI -0.202  -0.129  -0.163   0.182                  0.216
HL         -0.157          -0.127          -0.139
HP -0.155   0.832                                  0.240
HS  0.746  -0.102   0.257  -0.152
HR -0.155  -0.107  -0.122   0.101                  0.150
KA -0.167   0.774  -0.169   0.139
LA                                          0.829
LC -0.306   0.378  -0.125   0.529                          0.328
LH -0.256   0.556  -0.132   0.421           0.223   0.195
LM                                  0.112   0.221
LO -0.129   0.432           0.781                  0.251
LP  0.115           0.745
OR                                                         0.996
PL          0.369   0.675           0.337
PP  0.527           0.226  -0.167          -0.175
PS -0.212   0.301  -0.130   0.681           0.150   0.158
PT  0.741                  -0.100   0.150  -0.105
QR -0.194  -0.135
RA  0.195   0.227   0.578           0.205  -0.166  -0.107
RB -0.122   0.158           0.272                  0.934
RC  0.361                  -0.198          -0.176  -0.152
SG                                  0.806
SM          0.388                                  0.787
SO                 -0.100   0.386
TF          0.702   0.260
TG  0.141           0.583  -0.110           0.367   0.107
TO  0.418           0.567  -0.158
TP                  0.818
TR          0.141   0.306   0.238                  0.458
VC          0.403   0.246   0.309          -0.169
```

```
VK                                0.909

                Factor1 Factor2 Factor3 Factor4 Factor5 Factor6
SS loadings       5.840   3.991   3.577   3.540   3.028   2.644
Proportion Var    0.108   0.074   0.066   0.066   0.056   0.049
Cumulative Var    0.108   0.182   0.248   0.314   0.370   0.419
                Factor7 Factor8
SS loadings       2.427   2.198
Proportion Var    0.045   0.041
Cumulative Var    0.464   0.505

Test of the hypothesis that 8 factors are sufficient.
The chi square statistic is 1675.57 on 1027 degrees of freedom.
The p-value is 5.92e-34
```

Although this output seems complex, you can easily see that, for example, AE, AP, AS have positive contribution to the first factor, and AC, AO, FR have negative contributions; take character of these species, the first factor serves as a measure for the type of grassland (Crawley, 2010). Other factors also have clear biological interpretations (see commentary during the class). Remember that the number of parameters estimated (loadings for factors from different predictors) is 54 in factor analysis (the number of variables) and not 89 (the number of cases) as in PCA.

Cluster analysis

In cluster analysis the task is too look for any structuring in our data. We can look for clusters of similar objects using several methods – mainly, we can either build our structure starting from single individuals or divide all observations among groups. The simplest method works by fitting specified number of groups in our data in such way that the sum of squared Euclidean distances from the centres of these groups is minimized inside and across all groups. Well analyse a sample data where we have two variables and wish to see if they're structured in their 2-dimendional data space. As comparison, we'll look at actual structuring of these data indicated in the **group** column:

```
> klastr<-read.table("kmeansdata.txt", sep="\t", head=T)
> par(mfrow=c(2,2))
> with(klastr,plot(x,y,pch=16))
> with(klastr,plot(x,y,pch=16,col=group))
> klmod <- kmeans(klastr[,-3],6)
> with(klastr,plot(x,y,pch=16,col=klmod[[1]]))
> klmod <- kmeans(klastr[,-3],4)
> with(klastr,plot(x,y,pch=16,col=klmod[[1]]))
> par(mfrow=c(1,1))
```

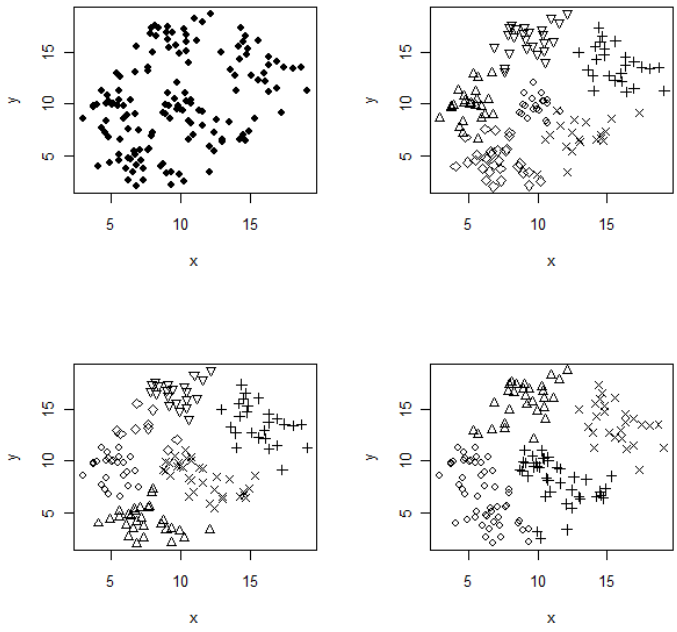


Fig 34: Raw data (upper left), correct clustering (upper right)
and two estimated clustering patterns (lower)

Here, we've plotted two plots for original data, one monochromatic, and one with *a priori* groups indicated. Two other plots are based on structuring as inferred from **kmeans** function for 4 and 6 clusters predefined. As you can see, structuring is apparent but several mistakes have been made. We can actually look at the precision by comparing structuring we know with this inferred in R:

```
> klmod <- kmeans(klastr[,-3],6)
> table(klmod[[1]],klastr$group)

    1  2  3  4  5  6
1   0  8  0  0  5  0
2   0  0  0  0  0 24
3  20  0  0 17  0  0
4   0  0 25  1  0  0
5   0  0  0  2 25  0
6   0 17  0  0  0  1
```

Now it's obvious that there are no mistakes in the first cluster (all 20 objects were correctly classified in it by classifying function) but for instance 5 objects from original 5th cluster were assigned to the 1st cluster (look at the 5th column). In general, this

method may not be precise, especially when one does not have any hints as to how many clusters to specify.

It's common to use clustering in taxonomic analyses. Here we'll try to assess usefulness of several morphological features in separating measured individuals in several taxa (Crawley, 2010). Here, we're in a convenient situation since we know we should have four taxa – hence, we know how many clusters we should specify.

```
> taxa<-read.table("taxon.txt",head=T)
> names(taxa)
[1] "Petals"    "Internode" "Sepal"     "Bract"     "Petiole"
[6] "Leaf"      "Fruit"
> pairs(taxa)
```

Looking at the pairwise correlation matrix (Fig. 34) indicates that at least two traits (Sepals and Petioles) should be useful in separating taxa. Let's see how clustering works here. We know, that in our data-frame there are 30 taxa and they're arranged one over another so we have consecutive 30s of species' data. It will allow us to look at the precision of this method:

```
> taxakl <- kmeans(taxa,4)
> taxateo <- rep(c(1,2,3,4),each=30)
> taxateo
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2
 [32] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3
 [63] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4
 [94] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
> table(taxakl[[1]],taxateo)
   taxateo
     1  2  3  4
  1  9  1  8 10
  2  0 13  5  0
  3  2 11 10 19
  4 19  5  7  1
```

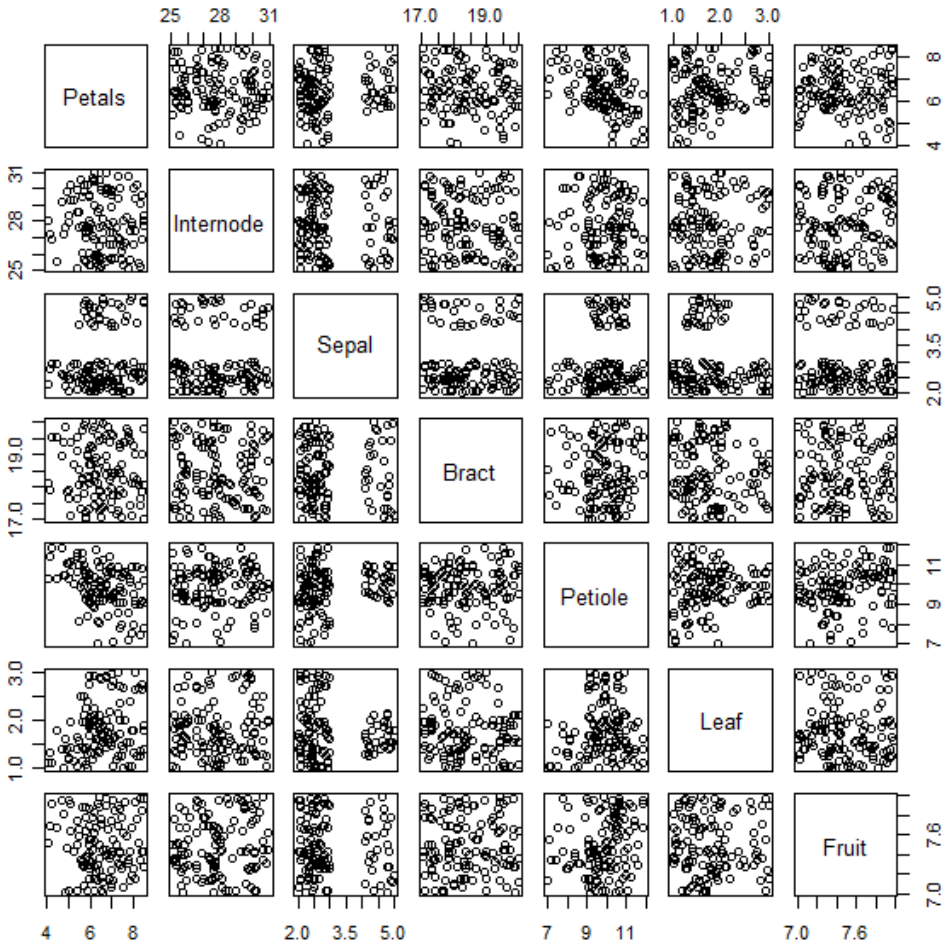


Fig 35: Pair-wise scatter plots for morphological traits

Although clustering works, it is fairly accurate. This would look even worse if we did not have the number of candidate taxa and used some other guesses. Is it any way around? Actually – there is and it's called regression trees. Let's look at this from the opposite end – it is when we have the right classification of our taxa (Crawley 2010; we'll use alternative dataset here):

```
> taxa<-read.table("taxonomy.txt",head=T)
> modkey1 <- tree(Taxon~.,taxa)
> plot(modkey1)
> text(modkey1)
```

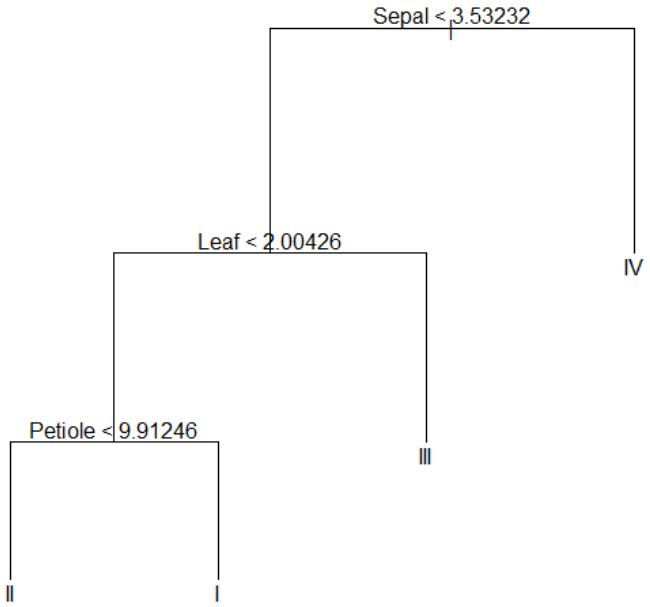


Fig 36: Regression tree for morphological clustering off species

As you can see – our 4 taxa can be fairly separated using several morphological features – and more important, we could build a simple dichotomous key for these plants. Actually, the raw form of such key can be obtained using built-in printing function:

```
> print(modkey1)
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

1) root 120 332.70 I ( 0.2500 0.2500 0.2500 0.2500 )
  2) Sepal < 3.53232 90 197.80 I ( 0.3333 0.3333 0.3333 0.0000 )
    4) Leaf < 2.00426 60  83.18 I ( 0.5000 0.5000 0.0000 0.0000 )
      8) Petiole < 9.91246 30   0.00 II ( 0.0000 1.0000 0.0000 0.0000 ) *
      9) Petiole > 9.91246 30   0.00 I ( 1.0000 0.0000 0.0000 0.0000 ) *
    5) Leaf > 2.00426 30   0.00 III ( 0.0000 0.0000 1.0000 0.0000 ) *
  3) Sepal > 3.53232 30   0.00 IV ( 0.0000 0.0000 0.0000 1.0000 ) *
```

We can also use some useful feature of tree models to visualize partitioning of our traits in some phase-space of candidate traits (adapted from Crawley, 2010). Here we'll depict our taxa in the 2D phase-space of two traits (most important ones) – Sepal and Leaf:

```
> modkey2 <- tree(Taxon~Sepal+Leaf,taxa)
> partition.tree(modkey2)
> attach(taxa)
> label<-ifelse(Taxon=="I","a",ifelse(Taxon=="II","b",
+ ifelse(Taxon=="II","c","d")))
> text(Sepal,Leaf,label)
```

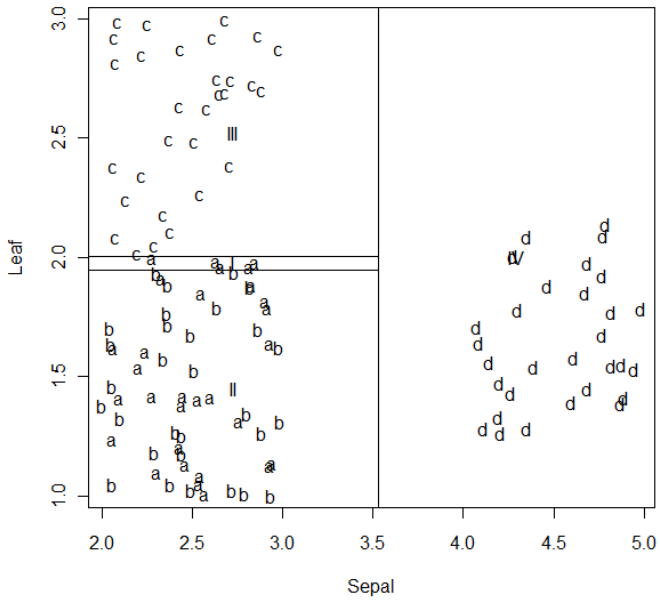


Fig 37: Phase-space for two most efficient separating traits

What we did here is we built a tree-model in a two-dimensional space of two trait-values and depicted our data in this space by "clusters" for each taxon (Crawley 2010). Additionally, we labelled each point with a letter to visualize how well our data fit this clustering. You can see that it is almost perfect separation of 3rd and 4th taxa, but 1st and 2nd are mixed. It's reasonable since they're separated by the Petiole lengths. Now – imagine running the same tree model in reverse, without prior knowledge of taxa. It would of course require some more attention and inspection of

results but eventually you would find your way to correct traits that separate your data into taxa.

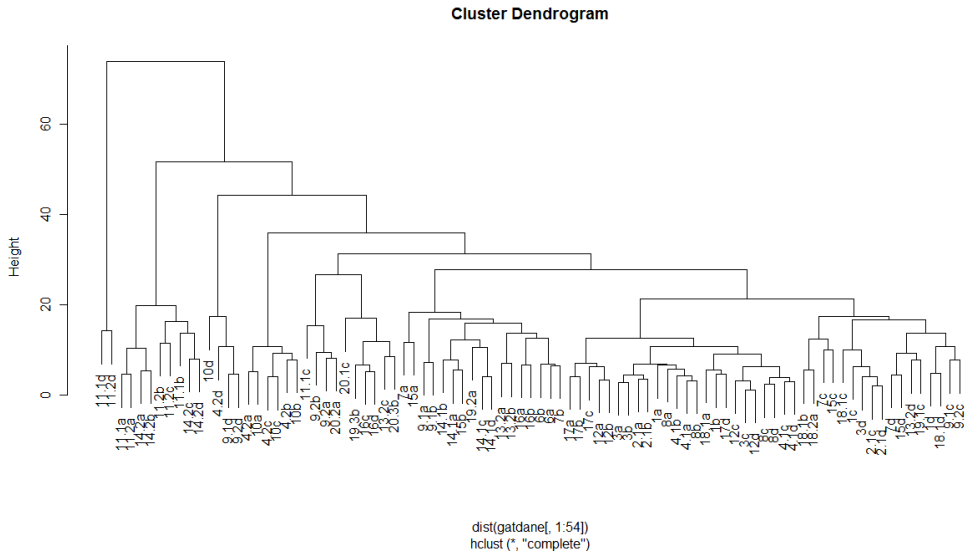
Hierarchical clustering

You've probably encountered pretty trees in hundreds of scientific papers. They depicted different kinds of hierarchical dependencies – and had in common one thing: they depicted in a hierarchical manner growing degree of similarity between some units as we climb up the hierarchy. Hierarchical clustering works by using some measures of geometric distance to cluster data units (individuals, species, etc.) into some more general units. Generalising up and up we get tree-like structure describing patterning in our data.

In R there are several "treeing" functions. Here we'll use the simplest one and analyse the data on plant communities you've encountered in PCA and FA (first two subchapters). To be able to effectively analyse those data we will first extract some labels identifying all units (plots) uniquely. Then, using our 54 predictors variables, we'll calculate pairwise distances between all possible cases (plots).

```
> etyk <- paste(gatdane$plot,letters[gatdane$lime],sep="")
> etyk
 [1] "1a"    "1b"    "1c"    "1d"    "2.1a"  "2.1b"  "2.1c"
 [8] "2.1d"  "3a"    "3b"    "3c"    "3d"    "4.1a"  "4.1b"
[15] "4.1c"  "4.1d"  "4.2a"  "4.2b"  "4.2c"  "4.2d"  "6a"
[22] "6b"    "7a"    "7b"    "7c"    "7d"    "8a"    "8b"
[29] "8c"    "8d"    "9.1a"  "9.1b"  "9.1c"  "9.1d"  "9.2a"
[36] "9.2b"  "9.2c"  "9.2d"  "10a"   "10b"   "10c"   "10d"
[43] "11.1a" "11.1b" "11.1c" "11.1d" "11.2a" "11.2b" "11.2c"
[50] "11.2d" "12a"   "12b"   "12c"   "12d"   "13.2a" "13.2b"
[57] "13.2c" "13.2d" "14.1a" "14.1b" "14.1c" "14.1d" "14.2a"
[64] "14.2b" "14.2c" "14.2d" "15a"   "15b"   "15c"   "15d"
[71] "16a"   "16b"   "16c"   "16d"   "17a"   "17b"   "17c"
[78] "17d"   "18.1a" "18.1b" "18.1c" "18.1d" "18.2a" "19.1c"
[85] "19.2a" "19.3b" "20.1c" "20.2a" "20.3b"
> drzewo <- hclust(dist(gatdane[,1:54]))
> plot(drzewo,labels=etyk)
```

Looking carefully through the tree (next page) and the data – you can see that this analysis clustered our data in such a way that similar plant communities were placed together, as explained in more detail during the class.


Cluster Dendrogram
Height
0
20
40
60
dist(gatdane[, 1:54])
hclust (*, "complete")

Advanced Issues

Szymon Drobniak

Jarrod Hadfield

Note

Topics presented below are selected and adapted from the workshop on advanced methods in mixed models implemented in R, that took place in January 2011 in the Evolutionary Biology Centre, University of Uppsala. Complete data files and complete scripts containing relevant R code are available on the website of this book (see Preface). Please note, that these subjects are more advanced than problems in the first part of this book. Also, the style of the remaining sections is somehow different, with less off-code narration and more inline comments (indicated by the leading # sign). In order to facilitate using these notes code blocks are identified with their unique numbers, which are the same as blocks' IDs in the R scripts available online.

In case you have executed **`library(NAME)`** and received error message saying that you have not the required package, remember to (re)install it using **`install.packages(NAME)`**.

Part A ~ Likelihood and Bayesian statistics

Seeing the invisible – likelihoods and posteriors

Maximum likelihood estimators are common in classical statistics. For instance, arithmetic mean, OLS estimates of regression coefficients – all are in fact estimators that maximize the likelihood of data given particular values of parameters, i.e. $max(P(\mathbf{y}|par))$. In general situations as considered here this likelihood is proportional to the product of probability densities of the data given particular values of parameters:

$L(par|\mathbf{y}) \sim \Pi_i P(\mathbf{y}|par_i)$

In this part we'll play with simple simulated Gaussian data and see how simple maximum likelihood estimation works. We'll learn how to use optimizing functions of R (which some of you may find useful in other applications) and how to produce multivariate graphs of likelihood surfaces.

First – we'll simulate simple normal data (10 observations) and see how do they look like in terms of the distributions they were taken from (after Hadfield 2010b). Likelihoods may be tricky and as you'll see – the likelihood of our data may be higher for different (!) parameters than those we'll use to simulate them. Note that since our data are sampled randomly you may obtain entirely different results that those presented below.

```
> ###code block 1

> dataG <- data.frame(y = rnorm(10,mean=0,sd=sqrt(1)))
> dataG$y
 [1] -0.2079101 -1.1445615 -0.0656215 -0.6294617  0.5422668
 [6]  0.7025364  0.7627269  0.1905778  1.6687900  2.0852642

> yscale <- seq(-3,3,0.1) #possible values of y for the plot
> Prob<-dnorm(yscale,mean=0,sd=sqrt(1)) #pdf
> plot(Prob~yscale,type="l")
> Prob.y <- dnorm(dataG$y, mean=0, sd=sqrt(1))
> points(Prob.y~dataG$y)

> L <- prod(Prob.y) #likelihood
> L
```

```
[1] 5.94403e-07
>
> Lalt <- prod(dnorm(dataG$y,mean=0,sd=sqrt(0.5)))
> Lalt
[1] 1.107162e-07


> plot(dnorm(yscale,0,sqrt(0.5))~yscale,type="l")
> lines(Prob~yscale,col="red")
> points(Prob.y~dataG$y,col="red")
> points(dnorm(dataG$y,0,sqrt(0.5))~dataG$y)
```

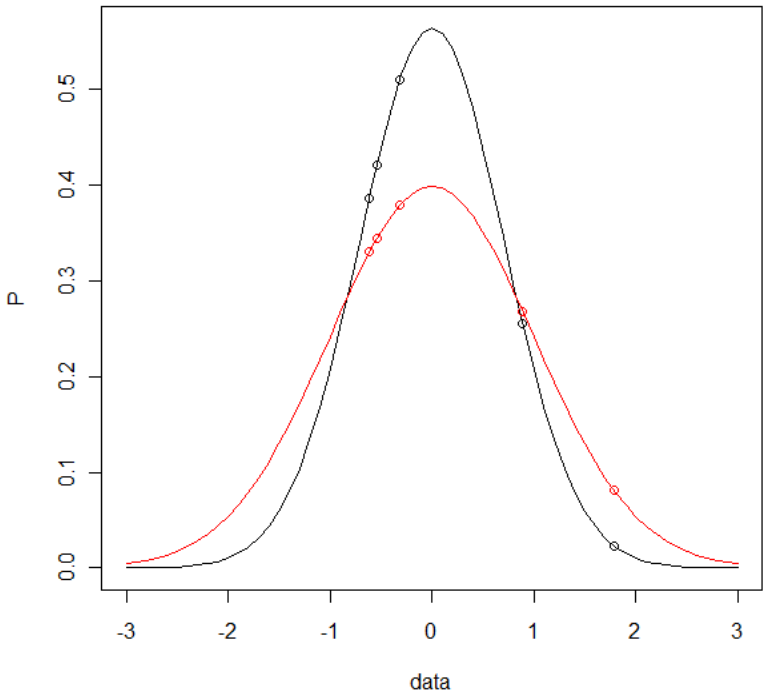


Fig 38: Sampled data superimposed on two normal distributions;
the red one is the true one

As you can see – the likelihood of our data is higher under different set of parameters and it's apparent from the plot (3 points of the black curve, generated using parameters different than those used for generating data, lie over the red, theoretically proper, curve). In order to fully understand what's happening we should evaluate the likelihood on the grid of possible parameters. Here we'll use simple loop

to iterate through the space of our parameters (mean and variance) to calculate possible values of *L* and then we'll plot them as a flattened perspective plot (contours). Be aware that each of you has slightly different values in you simulated data (in **rnorm** 'r' means random!) and you'll probably have to rescale your plots so that they could contain whole likelihood surface (to do this just play with the first two parameters in two commands in the frame below):

```
> ###code block 2
>
> x=seq(-1,1,0.05)
> y=seq(0,2,0.05)
> z=matrix(numeric(length(x)*length(y)),c(length(x),length(y)))
>
> for (i in 1:length(x)) {
+
+ for (j in 1:length(y)){
+
+ z[i,j]=prod(dnorm(dataG$y,mean=x[i],sd=sqrt(y[j])))
+ }
+ }
>
> z<-z/max(z)
> contour(x,y,z,nlevels=10,xlab="mean",ylab="variance")
```

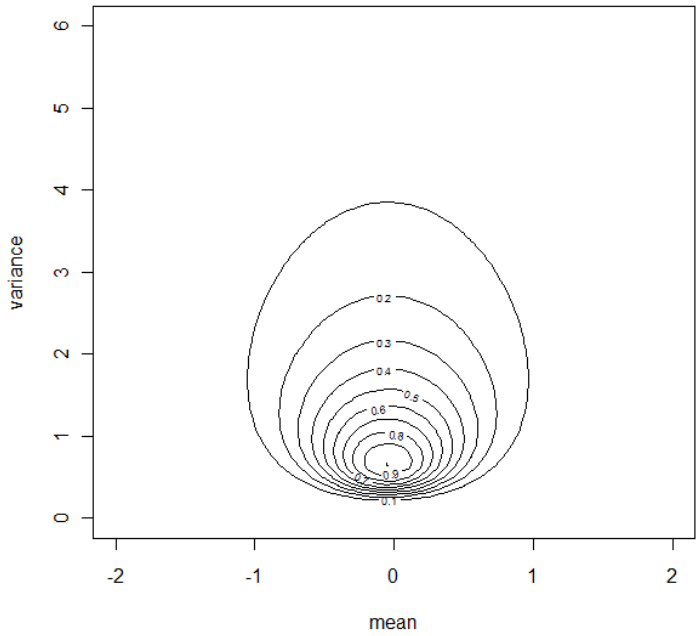


Fig 39: Contour representation of the likelihhod surface

Importantly we don't have to rely on visual inspection looking for ML estimator. We can use R built-in features designed for searching for functions maxima and minima. In such case you should define your maximized/minimized function (in our case (**`loglik`**) it will be the likelihood which is the product – or, on the log scale, the sum – of (logged) probability densities). In the optimizing routine you have to specify starting parameters (which may be our assumed parameters of the distribution; these will be coordinates of the space in which optimization will be done – you can locate them as the first argument in **`optim`**) – they have to be of the same number as parameters in the optimized function; you also have to provide all variables that are in the optimized function (here it's only **`y`**). Other arguments are: **`fn`** (defines the optimized function) and **`control`** (provides control parameters, such as **`fnscale`** (it multiplies the minimized function by -1, effectively maximizing it) or **`reltol`** (it sets the threshold for stopping the optimizing routine and deciding that the actual optimum has been reached). We'll compare our optimum with the estimates of a linear model (which uses REML instead of ML).

```
> ###code block 3
```

```
> loglik <- function(pars,y) {
+ sum(dnorm(y,pars[1],sqrt(pars[2]),log=TRUE))
+ }
>
> ML <- optim(c(mean=0,v=1),fn=loglik,y=dataG$y,
+ control=list(fnscale=-1,reltol=1e-16))
> ML$par
     mean         v
0.3904607 0.8768050

> REML <- glm(y~1,data=dataG)
> summary(REML)

Call:
glm(formula = y ~ 1, data = dataG)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.53502  -0.56280  -0.02404   0.35722   1.69480

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.3905     0.3121   1.251    0.242

(Dispersion parameter for gaussian family taken to be 0.9742278)

    Null deviance: 8.768  on 9  degrees of freedom
Residual deviance: 8.768  on 9  degrees of freedom
AIC: 31.064

Number of Fisher Scoring iterations: 2




> #REML estimator is better (the bias is smaller by
> #factor of n/n-1)
> ML$par["v"]*(10/9)
       v
0.9742278
```

Combining likelihood and prior knowledge

What's unique for Bayesian analysis is that we consider parameters as random rather than fixed and we use some knowledge about these parameters to estimate their values. In other words, the posterior probability of observing parameters of a given value depends both on the likelihood of the data of given these parameters and our prior knowledge about them:

$$P(par|\mathbf{y}) \sim L(par|\mathbf{y})P(par) \sim P(\mathbf{y}|par)P(par)$$

Diverse distributions could be used in the Bayesian framework to define priors but in our analyses we'll use two of them. Priors for fixed effects are defined using normal distribution with mean zero and very large (>1e+06) variance, making such prior essentially flat and uninformative. For (co)variances we use inverse

Wishart distribution (IW) which is slightly problematic for multivariate (co)variance structures (and we'll come back to them later). For simple variances IW is defined by two parameters: variance – **V** and belief parameter – **nu**. When belief goes to infinity, the distribution tends to a mode equal to **V**. In general the mode of the distribution is **(V*nu)/nu+2**. In R we can model IW using inverse gamma distribution (e.g. function **dinvgamma**) with parameters: **shape=nu/2** and **scale=nu*V/2**). Care is needed to ensure that the prior is proper (integrates to one as an ordinary distribution) and this condition holds for single variance components when **V>0** and **nu>0**. When **nu≤0** we get improper prior which – although difficult – may be useful (as we'll see later; Hadfield, 2010b).

Here we'll combine our likelihood function with prior densities to see how such estimates work compared to ML. First we'll define function for calculating prior probability for given values of parameters, then we'll combine these with likelihood and use to estimate values of the parameters. Since we're working on the log scale, it's summing and not multiplying that we'll employ.

```
> ###code block 4
>
> library(MCMCpack)
Loading required package: MASS
##
## Markov Chain Monte Carlo Package (MCMCpack)
##
## Support provided by the U.S. National Science Foundation
## (Grants SES-0350646 and SES-0350613)
##

> logprior <- function(pars,priorR,priorB) {
+ dnorm(pars[1],mean=priorB$mu,sd=sqrt(priorB$V),log=T)+
+ log(dinvgamma(pars[2],shape=priorR$nu/2,
+ scale=(priorR$nu*priorR$V)/2))
+ }

> prior <- list(R=list(V=1,nu=0.002),B=list(mu=0,V=1e+08))

> loglikprior <- function(pars,y,priorR,priorB) {
+ loglik(pars,y)+logprior(pars,priorR,priorB)
+ }

> Bayes <- optim(c(mean=0,v=1),fn=loglikprior,y=dataG$y,
+ priorR=prior$R,priorB=prior$B,
+ control=list(fnscale=-1,reltol=1e-16))

> x=seq(-1,1,0.05)
> y=seq(0,2,0.05)
> z1=matrix(numeric(length(x)*length(y)),c(length(x),length(y)))
```

```
> for (i in 1:length(x)) {
+
+ for (j in 1:length(y)){
+
+ z1[i,j]=exp(loglikprior(c(x[i],y[j]),
+ dataG$y,prior$R,prior$B))
+ }
+ }

> #z2<-z1/max(z1) sometimes does not work as NaNs are produced
> contour(x,y,z,nlevels=10,xlab="mean",ylab="variance")
> contour(x,y,z1,nlevels=10,xlab="mean",ylab="variance",
+ add=T,col="red")
```

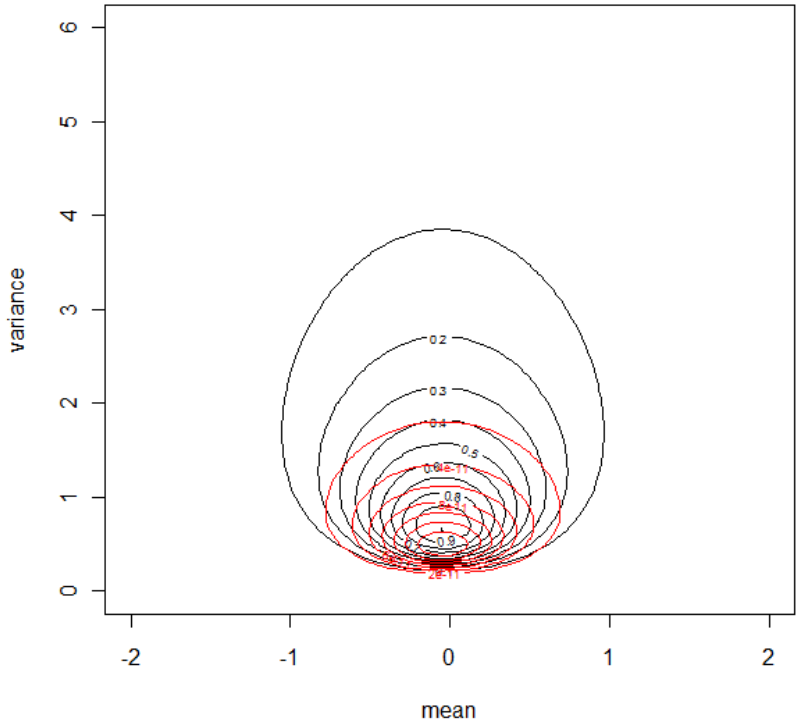


Fig 40: Likelihood surface (larger in black) and the same combined with the prior (smaller in red)

As you can see – variance estimates using prior are even more downwardly biased – which is caused by the fact that simple optimization of the **`L*prior`** ignores uncertainty of the mean estimate (Hadfield, 2010b). We can however integrate our

bivariate distribution along the mean scale to get the posterior for variance, which would take uncertainty in mean into account:

$$P(\sigma^2|\mathbf{y}) \sim \int P(\sigma^2, \mu|\mathbf{y}) d\mu$$

Important advantage of MCMC-based methods is that analytically it's most often impossible to get the posterior marginal distribution of a parameter.

```
> ###code block 5

> contour(x,y,z1,nlevels=10,xlab="mean",ylab="variance",
+ col="red")
> library(MCMCglmm)
> m1 <- MCMCglmm(y~1,data=dataG,prior=prior,thin=1,nitt=30000,
+ verbose=F)
> points(cbind(m1$Sol,m1$VCV),pch=".")
```

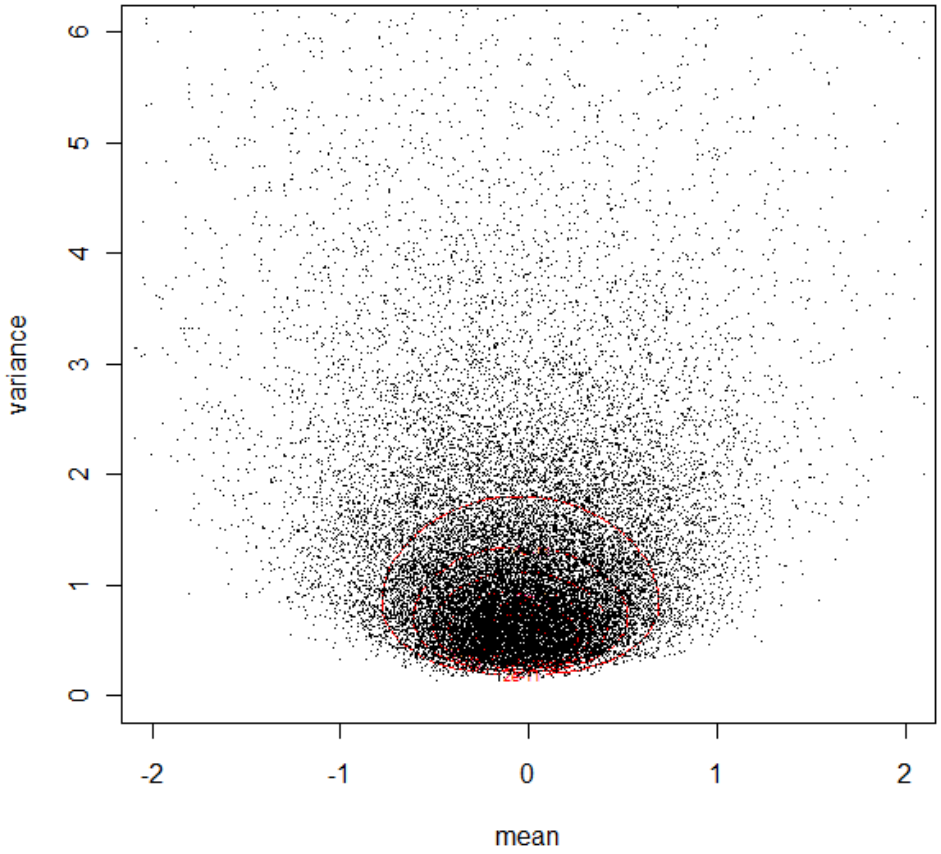


Fig 41: ML combined with the prior (lines) with the MCMC sample superimposed (points)

Of course, one would ask how sure we can be that our sample space (visualized above) is appropriate and guaranties we're integrating true distribution (i.e. integrating to one using the boundaries of our space)? If we look at the proportion of samples from the posterior contained within considered sample space you'll see it's almost 1. Thus, we can construct the posterior distribution safely over this range (try using whole sample from the posterior – such histogram would be impossible to interpret).

```
> ###code block 6
```

```
> prop.table(table(m1$Sol > -1 & m1$Sol<1 & m1$VCV<2))

    FALSE      TRUE
0.1444815 0.8555185
> hist(m1$VCV[which(m1$VCV<2)],breaks=30)
> abline(v=Bayes$par["v"],col="red") #estimates from optimising procedure
```

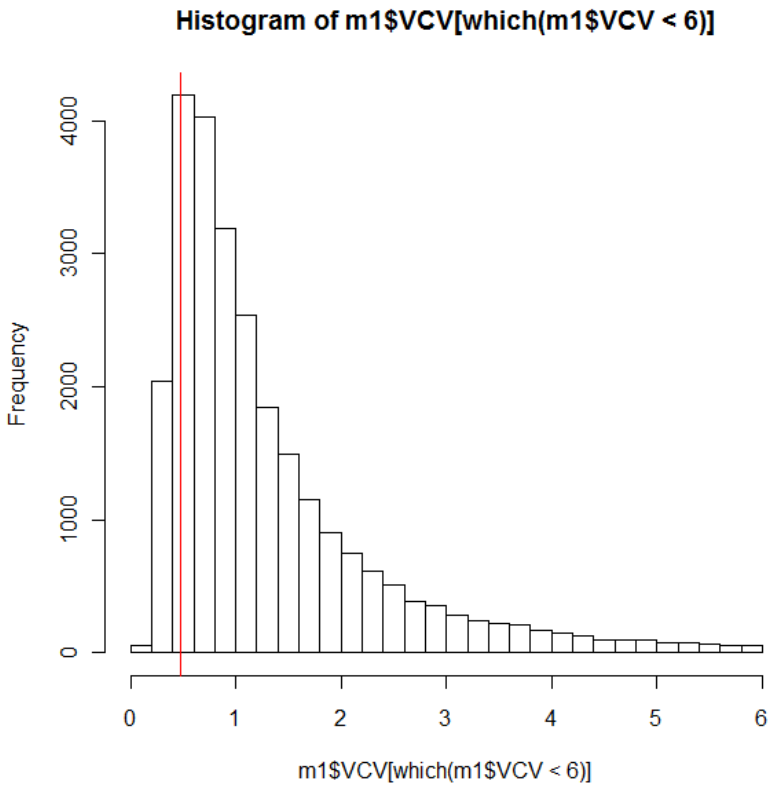


Fig 42: Posterior marginal distribution of the variance; line denotes REML estimate of the variance

Let's go nasty – improper priors

As I mentioned, sometimes priors are not proper, i.e. they don't integrate to unity. The simplest example is a uniform prior defined over **R**. It's not proper since it integrates to $+\infty$. Uniform prior would be proper only when defined over the range $A=[a,b]\subset\mathbf{R}$ so that $P(x\in A)(b-a)=1$.

In case of IW-distributed priors for single variance components they're improper when **nu≤0**. For **nu=0** we get flat prior for variance. This reduces well

known Bayesian equation to simple ML estimator: $P(par|\mathbf{y}) \sim P(\mathbf{y}|par)$. In other words – the joint posterior distribution will be equal to ML estimator but remember – modes you're getting analyzing problems are from marginal distributions, not from the joint one.

We may also define prior that will be non-informative for the variance and this could be achieved by setting `V=0` and `nu=-2`. Such prior makes joint posterior to deviate from ML estimates but marginal estimates of variance are in agreement with REML.

In general – priors in MCMCglmm lead often to confusion. Several conventions exist for defining them. E.g. improper priors can be useful in a way that they allow for reducing our problem to simple ML estimator or REML estimator (for marginal distributions of parameters). However, improper priors must be used with caution – improper prior distribution may lead to improper posterior distribution, which would be meaningless from a statistical point of view. The question is – which strategy to adopt in defining priors? First of all – use weak priors unless you want to impose some (strong) constraints on the variance. In general – having good data, with appropriate levels of replication, and sampling populations of random effects accordingly should make priors less influential – in other words, when the data contain enough information to estimate the parameters, priors should not influence these estimates. In case of less informative data you might consider using improper priors, but be extremely cautious. From my point of view two approaches are recommended: use either priors with `V=1` and `nu=0.002` or calculate the variance from your data and use it (partitioned accordingly with respect to random effects) as values for `V`. You'll see these approaches in further parts of this workshop.

Part B ~ More on Markov Chain methods

MCMCglmm and lmer – which to choose?

This is simple: if you have good, well replicated Gaussian data, with lots of information on large numbers of random effects' levels – use **lmer**. It performs well, but remember that significance tests may be a little cumbersome. However, if you want to fit categorical random interactions – avoid using **lmer**. It doesn't allow for residual variances to differ between levels of fixed effect and thus any differences here could possibly be confounded with the differences in variances associated with a particular random effect. For categorical interactions use **MCMCglmm**.

In case of non-Gaussian data use **MCMCglmm** – REML methods are not able to analytically derive likelihood in such data and work on approximations. If such approximations are then used in likelihood-ratio tests – results may not be reliable.

Finally – remember that Poisson and binomial data are almost always overdispersed. **lmer** has this famous "**quasi**" prefix for such distributions that should deal with it. However, it doesn't. **MCMCglmm** fits overdispersion by default – so it's much better choice. A good alternative is ASReml, which is faster than **MCMCglmm** – but it's not free which for many people is limiting. And it also works on REML estimates which may be problematic in case of "weird" distributions.

MCMC diagnostics

MCMCglmm works using randomization so utmost care should be taken to ensure that this random sampling actually samples joint posterior distribution of parameters. Specifically, you have to check if consecutive samples from the posterior are independent from each other. At the beginning they may not be independent as the walk through the posterior starts from some values, but then the chain should converge and samples should be independent.

At first, let's generate some "artificial" problems by shortening the MCMC chain in one of the previous models (on blue tits). We achieve this by setting the number of iterations to some low value (**nitt=2000**). Default **burnin=3000**, so we should lower this value below **2000**. We'll sample every second iteration (**thin=2**).

```
> ###code block 12

> library(MCMCglmm); data(BTdata)
> prior <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1,nu=0.002),
```

```
+ G2=list(V=1,nu=0.002)))
> m3.bad <- MCMCglmm(tarsus~sex, random=~fosternest+dam,
+ prior=prior,verbose=F, data=BTdata, nitt=2000,
+ burnin=500, thin=2)
> plot(m3.bad$VCV)
> autocorr(m3.bad$VCV)
, , fosternest

         fosternest         dam       units
Lag 0    1.00000000 -0.24455001 -0.10430602
Lag 2    0.82815231 -0.23892563 -0.10144780
Lag 10   0.47711854 -0.09162047 -0.09937022
Lag 20   0.23792851 -0.08204370 -0.09793121
Lag 100 -0.02210706  0.04458598 -0.04048683

, , dam


           fosternest          dam       units
Lag 0   -0.2445500147  1.000000000 -0.10084968
Lag 2   -0.2245963939  0.359331075 -0.01494919
Lag 10  -0.1340634723 -0.019236027  0.01143639
Lag 20  -0.1250328753 -0.009754384  0.02827148
Lag 100  0.0009055187 -0.073661071  0.01977242

, , units

          fosternest         dam       units
Lag 0   -0.104306016 -0.10084968  1.00000000
Lag 2   -0.094951195 -0.09790104  0.07213249
Lag 10  -0.036437767 -0.04785507 -0.01063517
Lag 20  -0.031778578  0.01160051 -0.03949663
Lag 100 -0.009335712  0.03043995 -0.03858646


> m3.good <- MCMCglmm(tarsus~sex, random=~fosternest+dam,
+ prior=prior,verbose=F, data=BTdata,
+ nitt=50000, burnin=3000, thin=50)
> plot(m3.good$VCV)
> autocorr(m3.good$VCV)
, , fosternest

            fosternest          dam       units
Lag 0     1.0000000000 -0.225723765 -0.18379490
Lag 50    0.0007770752 -0.027699347  0.02414860
Lag 250   0.0223765238  0.019155326 -0.01651623
Lag 500  -0.0197449053  0.002413883 -0.02576015
Lag 2500  0.0043333577  0.019968891  0.04245713

, , dam

          fosternest         dam        units
Lag 0    -0.22572376  1.00000000 -0.015647709
Lag 50   -0.06615831  0.01547784  0.029970618
Lag 250  -0.06405542  0.03430238 -0.001347705
Lag 500   0.03822892 -0.02795335 -0.015139277
Lag 2500 -0.02030468  0.07640567  0.014159344
```

```
, , units

           fosternest          dam       units
Lag 0    -0.183794898 -0.015647709  1.00000000
Lag 50   -0.011400580  0.002101844 -0.02310039
Lag 250  -0.005789198 -0.060701762  0.04574427
Lag 500  -0.026688031 -0.023276145  0.02380457
Lag 2500 -0.009362384  0.012835732  0.02213512

> #try below if you don't want to have huge
> #complex matrix outputs
> diag(autocorr(m3.good$VCV)[2,,])
   fosternest           dam         units
 0.0007770752  0.0154778375 -0.0231003891
```

The first model mixes poorly, and clear trends in time series suggest non-independence of samples drawn from posterior. Additionally, **autocorr** indicates substantial autocorrelation in random effects of **dam** and **fosternest** (in **units** it's smaller). After extending the chain problems disappear. Chains are in the form of flat time series, and autocorrelations are well below 0.05.

Finally, there's one more aspect of MCMC diagnostics: we should not only ensure independence of consecutive samples but also make sure that all effects are sampled good enough, i.e. samples we based our estimation on are large enough.

```
> ###code block 12a

> effectiveSize(m3.good$VCV)
fosternest        dam      units
  940.0000   816.6073   940.0000
```

More on overdispersion – Poisson data

We have already seen how non-gaussian data can mess up with our conclusions if handled incorrectly. Also, we have seen substantial influence of additional variability in our data that is not accounted for in the analyses. We have modelled this overdispersion of our data removing one of predictor variables, *a priori* known to influence the response. Omitting one variable not only extremely biases estimates but also changes deviance to *df* ratio. In general, if the model fitted is correct, the asymptotic distribution of deviance should be proportional do a Chi-squared variable with *n-p df* (roughly speaking number of data minus number of predictors): $D \sim \chi^2(df=n-p)$. If $D>n-p>E[\chi^2(df=n-p)]$ it may indicate overdispersion. To be more practical, in the presence of overdispersion the ratio of residual deviance to residual *df* will be greater than 1. It's value approximately tells us about the strength of overdispersion. Even using quasipoisson distribution does not change anything –

estimates look the same. However, when fitting the same model in MCMCglmm – estimates are much better. They're still biased but much closer to their true values. It is because MCMCglmm uses additive model of overdispersion. What does it mean?

In its usual form linear model I defined like this: $\mathbf{y}=\mathbf{X}\boldsymbol{\beta}+\mathbf{e}$ where $\mathbf{e}$ is residual (unexplained variance in the response). Taking expectations gives: $E[\mathbf{y}]=\exp(\mathbf{X}\boldsymbol{\beta})$. Exponent indicates, that it's a Poisson process for which log is the link function. We may present this on the scale of the latent variable: $\mathbf{l}=\boldsymbol{\eta}$ which is equivalent to $\log(E[\mathbf{y}])=\mathbf{X}\boldsymbol{\beta}$. However, in the presence of overdispersion, there's additional variation on top of the predicted value and it gives: $E[\mathbf{y}]=\exp(\mathbf{X}\boldsymbol{\beta}+\mathbf{e}^*)$ or $\mathbf{l}=\boldsymbol{\eta}+\mathbf{e}^*$. Now it is not entirely true that $\mathbf{y}\sim Pois(\exp(\mathbf{l}))$ because there is this additional variation over the variability of Poisson process (Hadfield, 2010b). We can actually see these additional "residuals" (quotation marks indicate that this residual shows deviation from the variance expected by the Poisson process for a given mean). We'll analyse data on traffic accidents in Sweden. Analysis was performed to see if speed limit has some effect on the number of accidents, and if there are any year-by-year and day-by-day trends.

```
> ###code block 14

> library(MASS)
> data(Traffic)
> Traffic$year<-as.factor(Traffic$year)
>
> m4.bad <- glm(y~limit+year+day,family="poisson",data=Traffic)
> summary(m4.bad)

Call:
glm(formula = y ~ limit + year + day, family = "poisson", data = Traffic)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-4.1774  -1.4067  -0.4040   0.9725   4.9920

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.0467406  0.0372985  81.685  < 2e-16 ***
limityes    -0.1749337  0.0355784  -4.917 8.79e-07 ***
year1962    -0.0605503  0.0334364  -1.811   0.0702 .
day          0.0024164  0.0005964   4.052 5.09e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 625.25  on 183  degrees of freedom
Residual deviance: 569.25  on 180  degrees of freedom
AIC: 1467.2

Number of Fisher Scoring iterations: 4
```

```
> m4.bad$deviance/m4.bad$df.residual
[1] 3.162493
> # >3 times greater variation than expected
```

We can extract information on these residuals – this would require recording the behaviour of the latent variable (logged expectation of the response in this case). We'll show how much this additional variation changes Poisson process.

```
> ###code block 15

> prior <- list(R=list(V=1,nu=0.002))
> m4.good <- MCMCglmm(y~limit+year+day,family="poisson",data=Traffic,
+ prior=prior,verbose=F,pl=T)#pl saves the latent variables
> lat92 <- m4.good$Liab[,92]#predicted by Poisson process
> eta92 <- m4.good$Sol[,"(Intercept)"]+m4.good$Sol[,"day"]*92
> #particular realisation of Poisson process on day 92 in 1961
> resid92 <- lat92-eta92
> mean(resid92)
[1] -0.1240384
> #this realisation is lower than expected from Poisson process
```

The figure below actually shows all 92 realisations of this estimated Poisson process in 1961 without the speed limit. As you can see although the Poisson process is one (straight line in the middle, linearized as we use predictions on the link-function scale, i.e. latent variables), each day randomly deviates from this prediction (thin dotted lines scattered around the thick line). One particular realisation (92nd day) is depicted using dashed line – it's the same observation as the one for which we have calculated the residual above (Hadfield, 2010b).

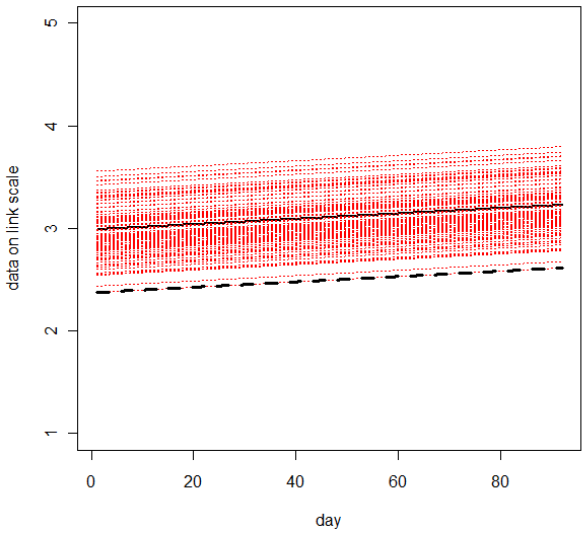


Fig 43: Graphical illustration of overdispersion (see text)

Overdispersion and random effects

The distinction between fixed and random effects is sometimes difficult and controversial (see the discussion about year effect in countless ecological papers). However, this controversy in Bayesian analysis largely vanishes since ALL effects are basically random, they just differ in the way we define their variances. For fixed effects variances are set as very large, yielding flat priors, whereas for variance components we shrink this variance to allow it's direct estimation (for in random effects it's variance we're interested in). Let's see how we can see this equivalence. First we'll fit simple fixed-effect model to our traffic data and obtain predictions for both years (Hadfield, 2010b).

```
> ###code block 16

> X <- model.matrix(y~limit+year+day,data=Traffic)
> X[c(1,2,3,91,92,183,184),]
    (Intercept) limityes year1962 day
1             1        0        0   1
2             1        0        0   2
3             1        0        0   3
91            1        0        0  91
92            1        0        0  92
183           1        1        1  91
184           1        1        1  92
> m5.fix <- MCMCglmm(y~limit+year+day,data=Traffic,
+ verbose=F,family="poisson")
```

```
> # using default prior
> #prediction for 1961 and 1962 with no speed limit
> y61.m5.fix <- m5.fix$Sol[,"(Intercept)"]
> y62.m5.fix <- m5.fix$Sol[,"(Intercept)"]+m5.fix$Sol[,"year1962"]
> posterior.mode(y61.m5.fix)
    var1
2.974852
> posterior.mode(y62.m5.fix)
    var1
2.915276
```

Now we redefine model so that year is treated as random effect BUT is associated with large variance, so basically it's the same as fixed effect. Note different method for obtaining predictions as in random effects intercept is suppressed by default.

```
> ###code block 17

> Z <- model.matrix(~year-1,data=Traffic)
> Z[c(1,2,3,91,92,183,184),]
    year1961 year1962
1          1        0
2          1        0
3          1        0
91         1        0
92         1        0
183        0        1
184        0        1
> X2 <- model.matrix(y~limit+day,data=Traffic)
> X2[c(1,2,3,91,92,183,184),]
    (Intercept) limityes day
1             1        0   1
2             1        0   2
3             1        0   3
91            1        0  91
92            1        0  92
183           1        1  91
184           1        1  92



> W<-cbind(X2,Z)#in bayesian statistics we use combined Z nad X matrices
> W[c(1,2,3,91,92,183,184),]
    (Intercept) limityes day year1961 year1962
1             1        0   1        1        0
2             1        0   2        1        0
3             1        0   3        1        0
91            1        0  91        1        0
92            1        0  92        1        0
183           1        1  91        0        1
184           1        1  92        0        1

> prior <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1e+08,fix=1)))
> m5.ran<-MCMCglmm(y~limit+day,random=~year,family="poisson",
```

```
+ data=Traffic,verbose=F,prior=prior,pr=T)
> #pr save the posterior of random effects
> y61.m5.ran <- m5.ran$Sol[,"(Intercept)"]+
+ m5.ran$Sol[,"year.1961"]
> y62.m5.ran <- m5.ran$Sol[,"(Intercept)"]+
+ m5.ran$Sol[,"year.1962"]

> #comparing posteriors for year effects from fixed and random effects

> y.fix <- mcmc(cbind(y1961=y61.m5.fix,y1962=y62.m5.fix))
> y.ran <- mcmc(cbind(y1961=y61.m5.ran,y1962=y62.m5.ran))
> plot(mcmc.list(y.fix,y.ran)) #virtually the same!
> #black trace is for year as fixed effects
> #red trace for year as random effect
> #unfortunately as we have just two levels of year
> #treating this as random confounds year effects with intercept
```

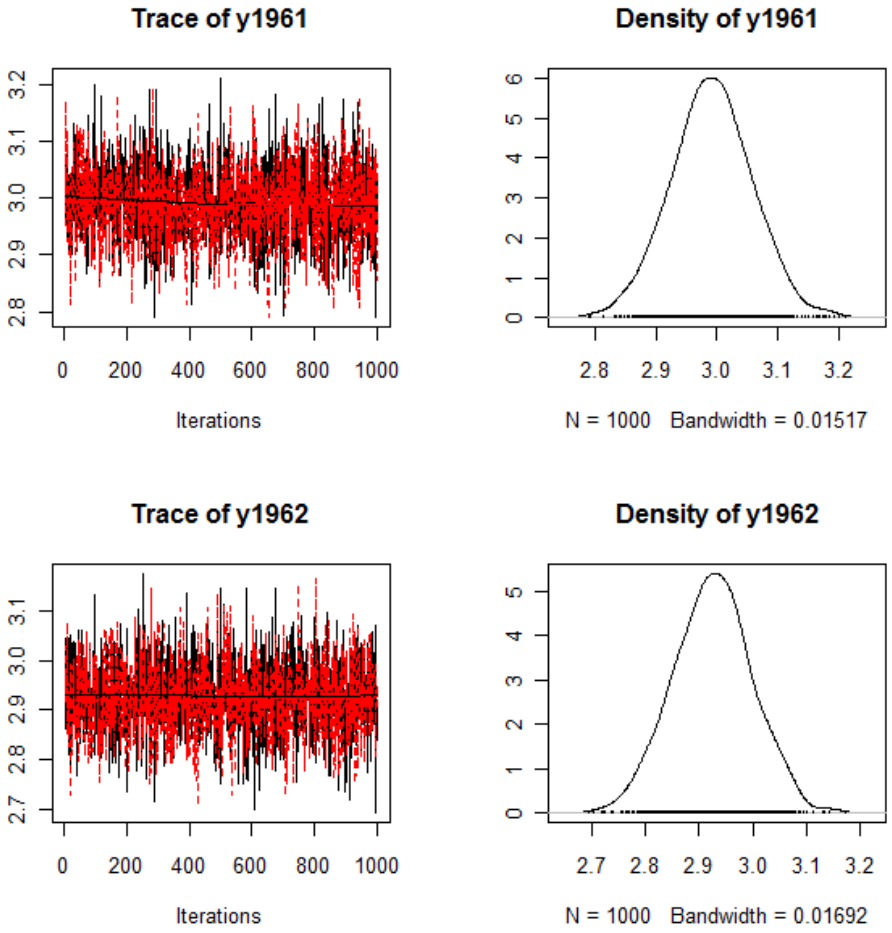


Fig 44: Treating year as random and fixed effects is the same here!

```
> plot(c(m5.ran$Sol[,"year.1962"]+
+ m5.ran$Sol[,"year.1961"])/2,m5.ran$Sol[,"(Intercept)"])
```

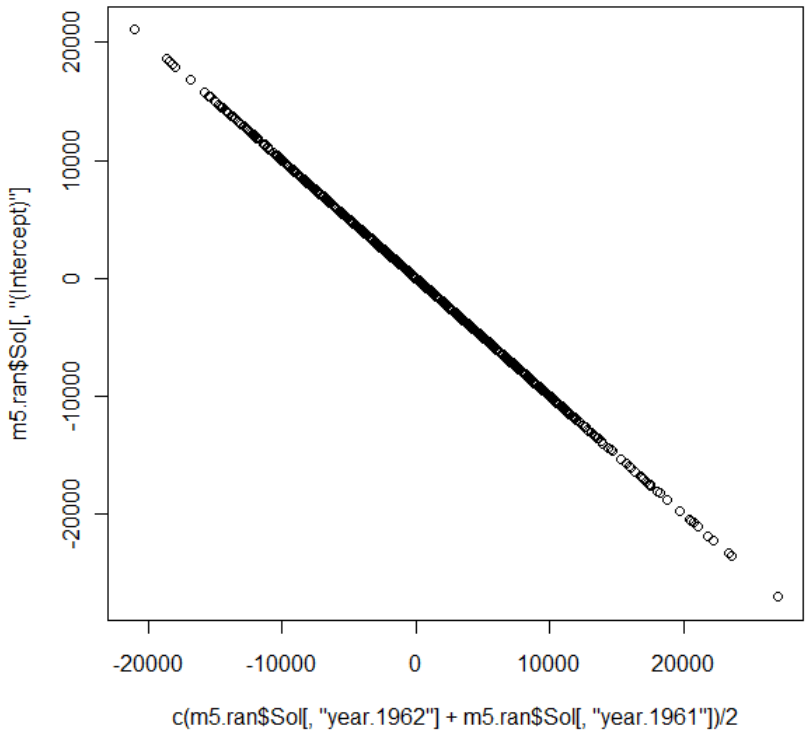


Fig 45: Intercept and year effects are virtually the same due to confounding of these two factors

And what if we made a more sensible decision and treated day as random effect? We'll leave day as continuous predictor to see any trends associated with day, but also we'll put categorical variable day as random effect, to account for between day variability. Recall that we've earlier observed this variability as overdispersed residuals in the Poisson process. Accounting for variability in days almost entirely removes overdispersion and shrinks residual variance to close to zero.

```
> ###code block 18

> Traffic$day<-as.factor(Traffic$day)
> prior <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1,nu=0.002)))
> m6 <- MCMCglmm(y~limit+year+as.numeric(day),random=~day,
+ family="poisson", data=Traffic, prior=prior, verbose=F)
> summary(m6)
```

```
 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 1166.191

 G-structure:  ~day

    post.mean l-95% CI u-95% CI eff.samp
day   0.09221  0.06065   0.1296    170.4

 R-structure:  ~units

      post.mean  l-95% CI u-95% CI eff.samp
units  0.006757 0.0002729  0.01838    49.26

 Location effects: y ~ limit + year + as.numeric(day)

                 post.mean   l-95% CI   u-95% CI eff.samp
(Intercept)      3.0116915  2.8451222  3.1551507    342.8
limityes        -0.2495462 -0.3345628 -0.1533956    145.4
year1962        -0.0377975 -0.1201447  0.0389475    193.7
as.numeric(day)  0.0024443 -0.0002819  0.0050961    268.4
                 pMCMC
(Intercept)     <0.001 ***
limityes        <0.001 ***
year1962         0.330
as.numeric(day)  0.096 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> plot(m6$VCV)
> autocorr(m6$VCV)
, , day

               day       units
Lag 0    1.00000000 -0.23114258
Lag 10   0.30761083 -0.20437420
Lag 50   0.12384555 -0.12673639
Lag 100  0.06815593 -0.11596727
Lag 500 -0.02644770  0.04207223

, , units

               day       units
Lag 0   -0.23114258  1.00000000
Lag 10  -0.23203677  0.84447766
Lag 50  -0.17455563  0.53681546
Lag 100 -0.13089769  0.36073419
Lag 500  0.02705169 -0.09263029

> #we'll run the model for longer to treat
> #autocorrelation in residuals

> m6 <- MCMCglmm(y~limit+year+as.numeric(day),random=~day,
+ family="poisson",data=Traffic,prior=prior,verbose=F,
+ nitt=100000,burnin=20000,thin=50)
```

```
> plot(m6$VCV)
> #traces look better butperhaps improper
> #or expanded priors would be better
```

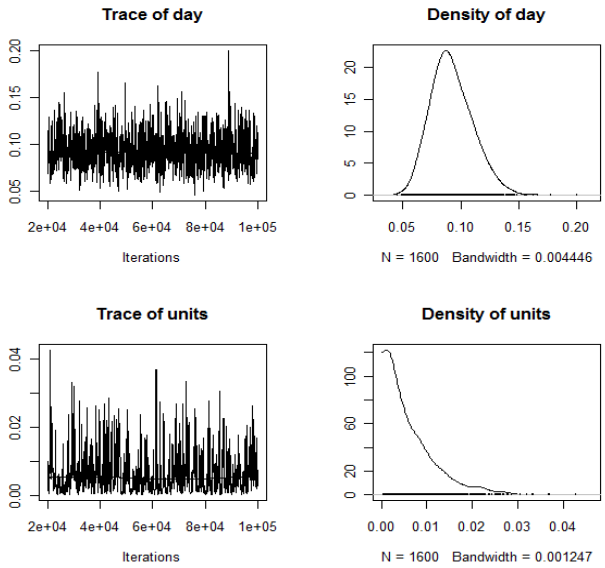


Fig 46: Traces for model with day as random effect

Binary/categorical data

Often in biology our data can be expressed as categories, ordered or without any numerical value (e.g. colours, sexes, success/failure data). In such cases we should use **`categorical`** family (or **`ordinal`** if our categories are ordered in any way), associated with link-functions logit or probit, respectively. Such data can be troubling and difficult to analyse.

We're in the best positions if we have binomial data, i.e. we have some units and within every unit we count some successes and some failures. Having such data makes possible to see if there's any heterogeneity in those units with respect to underlying probabilities associated with the binomial process. Here we'll generate simple binomial data which show such heterogeneity. Note that if in such data only intercept is fitted, it indicates heterogeneity as this intercept would be different than probabilities in every unit.

```
> ###code block 19

> ones <- rbinom(20, size=5, prob=c(0.2,0.8))
> zeros <- 5-ones
> bdata <- rbind(ones,zeros)
```

```
> bdata<-rbind(bdata,unit=letters[1:20])
> bdata<-as.data.frame(t(bdata))

> prior <- list(R=list(V=1,nu=0.002))

> m7.bin <- MCMCglmm(cbind(ones,zeros)~1,
+ data=bdata,family="multinomial2",
+ prior=prior,verbose=F,nitt=100000,
+ burnin=20000,thin=50)
> summary(m7.bin)

 Iterations = 99951
 Thinning interval  = 20001
 Sample size  = 1600

 DIC: 193.2368

 R-structure:  ~units

      post.mean  l-95% CI u-95% CI eff.samp
units    0.4696 0.0003961    1.537    705.2

 Location effects: cbind(ones, zeros) ~ 1

            post.mean l-95% CI u-95% CI eff.samp pMCMC
(Intercept)   -0.1216  -0.6218   0.3347      983 0.596
> install.packages("boot");library(boot)
> inv.logit(summary(m7.bin)$solutions[1]) #intercept is 0.5
[1] 0.4696424

> plot(m7.bin$VCV)

> data(PlodiaR)#and there is substantial variation above that from
> #binomial process
```

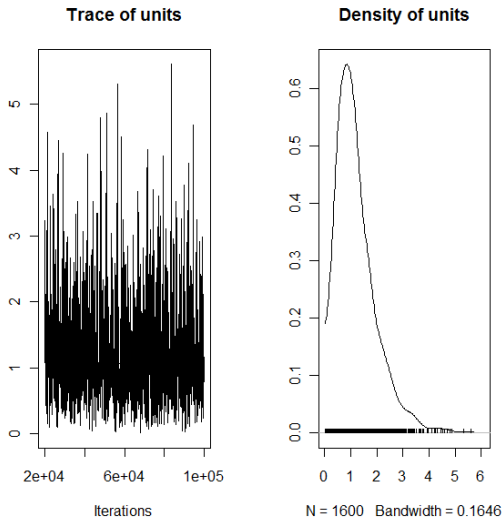
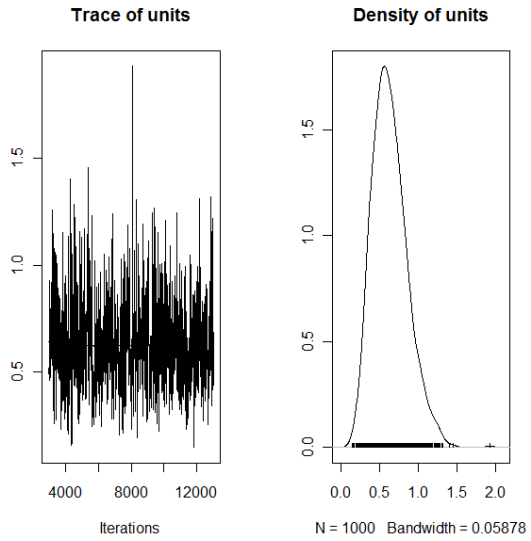


Fig 47: Residuals show substantial overdispersion

```
> m8.bin <- MCMCglmm(cbind(Pupated,Infected)~1,
+ family="multinomial2",
+ data=PlodiaR, verbose=F)
> plot(m8.bin$VCV)#again there is additional residual variation
> #that variation may be attributed to family effects
```



```
> #are Family effects really so variable?
> mode.mu <- posterior.mode(m8.bin$Sol)
> mode.V <- posterior.mode(m8.bin$VCV)
> ondatascale <- inv.logit(rnorm(10000, mean=mode.mu,
+ sd=sqrt(mode.V)))
> hist(ondatascale) #yes, they are!
```

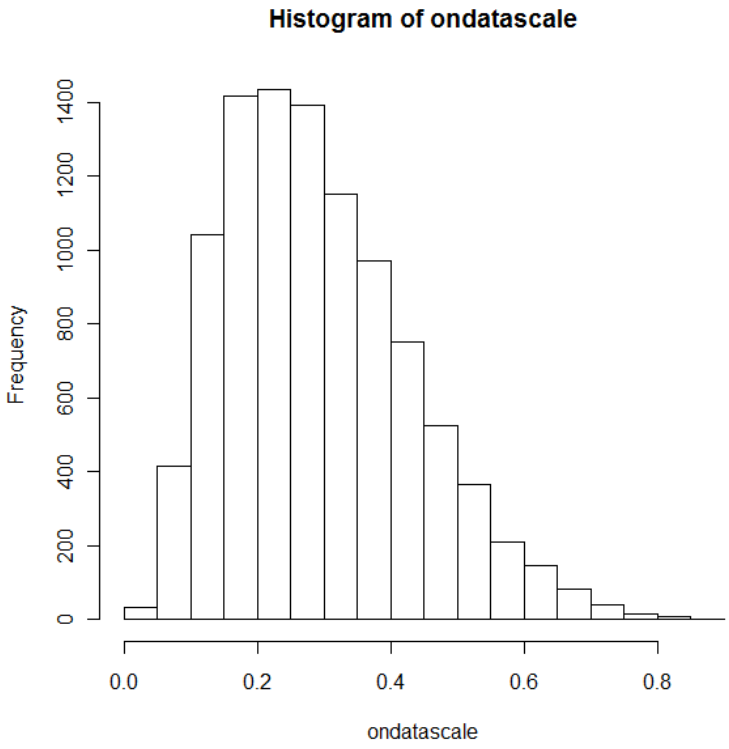


Fig 48: Histogram of the family effects based on estimates from the model

Things become more complicated if we don't have such unit-grouping and every binary observation is repeated only once. Then we are not able to distinguish between equal probabilities in every unit or extreme asymmetry in some groups compared to others. Such scenarios would be indistinguishable and importantly every numerical inference would be biased by the choice of underlying residual (units) variance as it would be meaningless. We'll reanalyse Plodia data, but rewritten in the form of binary variables. As in such process residual variance cannot be estimated we'll fix it at some value and see what happens for different fixing values (Hadfield, 2010b).

```
> ###code block 20

> data(PlodiaRB)
> prior1 <- list(R=list(V=1,fix=1),G=list(G1=list(V=1,nu=0.002)))
> prior2 <- list(R=list(V=2,fix=1),G=list(G1=list(V=1,nu=0.002)))
>
> m9.bin1 <- MCMCglmm(Pupated~1,random=~FSfamily,
+ family="categorical",
+ data=PlodiaRB,prior=prior1,verbose=F)
> m9.bin2 <- MCMCglmm(Pupated~1,random=~FSfamily,
+ family="categorical",
+ data=PlodiaRB,prior=prior2,verbose=F)
>
> plot(mcmc.list(m9.bin1$Sol,m9.bin2$Sol))
> plot(mcmc.list(m9.bin1$VCV,m9.bin2$VCV))#both posteriors differ!
> #red trace is for V=2
```

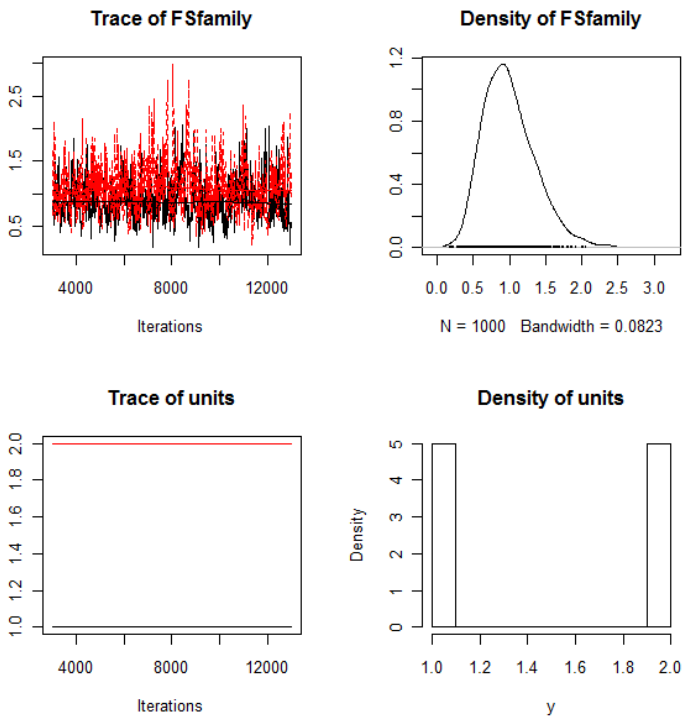


Fig 49: Models with residual variance fixed at two values

Both intercept and family variance posteriors differ with regard to the residual variance we've chosen. However it should not worry us. First of all – what matter the most here is not the absolute variation among families, but the degree to which two states (Pupated/Infected) are correlated within the same family. This information is contained in the coefficient of intraclass correlation, calculated like this: **IC** = *Var*(**FSfamily**)/(*Var*(**FSfamily**)+*Var*(**units**)+**c**), where the constant **c=pi^2/3** for logit link, and **c=1** for probit link. You can check that both IC's have the same posterior distribution:

```
> ###code block 21

> IC1 <- m9.bin1$VCV[,1]/(rowSums(m9.bin1$VCV)+pi^2/3)
> IC2 <- m9.bin2$VCV[,1]/(rowSums(m9.bin2$VCV)+pi^2/3)
> plot(mcmc.list(IC1,IC2))
```

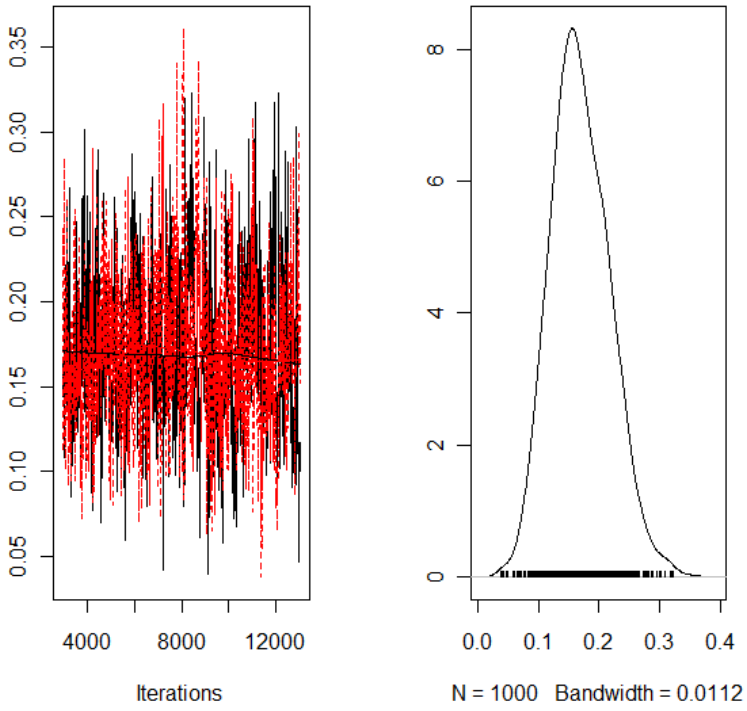
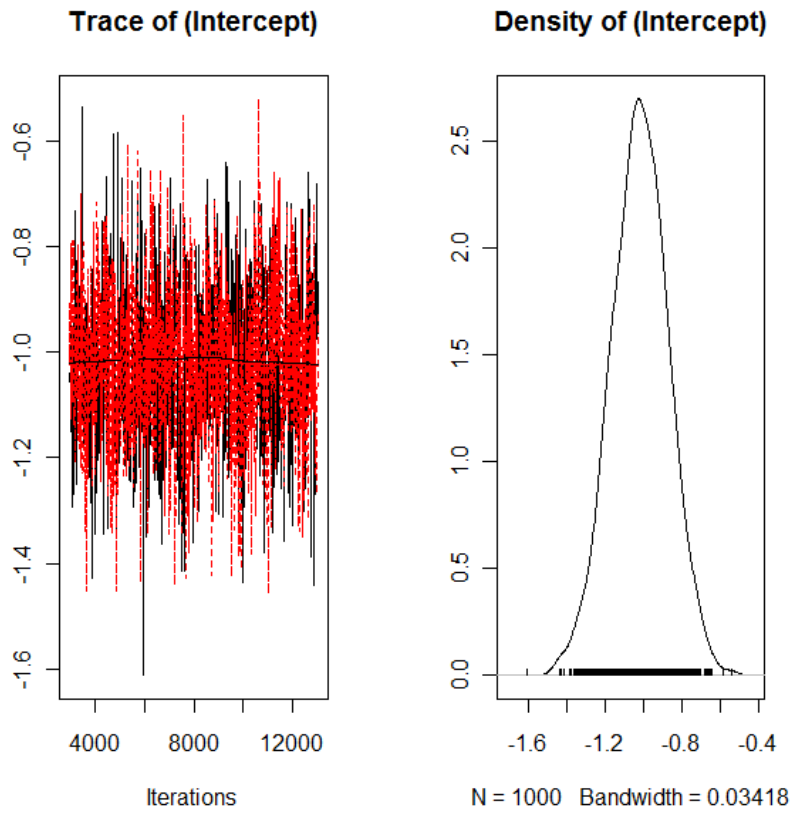


Fig 50: Posterior distributions of intraclass correlation coeffcients

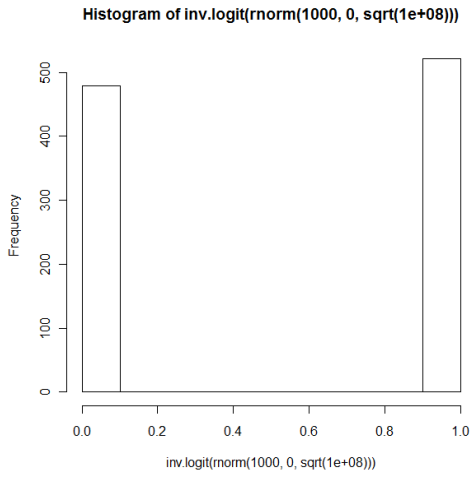
As for intercept, we can use Hadfield's results (2010b), due to Diggle et al. (2004), and rescale estimates so that they assumed some particular value of residual variance (*Var*(**units**)=**v**). Location effects (intercept, regression coefficients) can be rescaled by factor **sqrt((1+c^2*v)/(1+c^2****Var***(units)))** and variance estimates may be rescaled by factor **(1+c^2*v)/(1+c^2****Var***(units)).** The constant is **1** for probit and **16*sqrt(3)/15*pi** for logit. Let's try this for assumed residual variance of zero (**v=0**). Posteriors of Intercept are the same, up to Monte Carlo error.

```
> ###code block 22

> c <- 16*sqrt(3)/(15*pi)
> Int1 <- m9.bin1$Sol/sqrt(1+c^2*m9.bin1$VCV[,2])
> Int2 <- m9.bin2$Sol/sqrt(1+c^2*m9.bin2$VCV[,2])
> plot(mcmc.list(Int1,Int2)) #the same
```



Importantly, binary data can cause problems when there's large (near complete) separation, i.e. when most successes happened in one unit and most failures in other. This is because although on the link (logit) scale prior for the mean is flat (large variance), it's not flat at all on the data scale and has two distinct modes:

```
> ###code block 23

> hist(inv.logit(rnorm(1000,0,sqrt(1e+08))))
> #alternatively
> #hist(plogis(rnorm(1000,0,sqrt(1e+08))))
```



This inconsistency between the link and data scales isn't that important if one analyses well structured data, e.g. where representation of "ones" and "zeros" is roughly equal in experimental units. However, if the effect on one factor is so strong that particular levels of this factor have only ones or zeros – problems may appear. This type of data is called to have large separation. Let's simulate toy data with such huge separation (output of a theoretical experiment with control (1) and treatment (2) where treatment gets most/all zeros from the response) and see how we can analyse them using usual **`glm()`** and **`MCMCglmm()`**. Obviously, in such a case the effect of the treatment should be highly significant. It's apparent, that only after changing the prior (and removing intercept) we can get some sensible results (Hadfield, 2010b). The correction we use is setting the prior for the mean to the value of $1+(\pi^2/3)$. The improvement in mixing and traces' shape is apparent and exact binomial test confirms that results of our model are sensible:

```
> ###code bloc 24

> exper <- gl(2,25)
> y <- rbinom(50,1,c(0.5, 0.001)[exper])
> bdata2 <- data.frame(exp=exper,y=y)
> table(bdata2)
   y
exp  0  1
  1 14 11
  2 25  0
>
> m10.glm <- glm(y~exp,data=bdata2,family="binomial")
> summary(m10.glm)#no significant effect!

Call:
glm(formula = y ~ exp, family = "binomial", data = bdata2)

Deviance Residuals:
       Min          1Q      Median          3Q         Max
-1.077e+00  -1.077e+00  -7.976e-05  -7.976e-05   1.281e+00

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -0.2412     0.4029  -0.599    0.549
exp2         -19.3249  2150.8026  -0.009    0.993

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 52.691  on 49  degrees of freedom
Residual deviance: 34.296  on 48  degrees of freedom
AIC: 38.296

Number of Fisher Scoring iterations: 18


> prior.def<-list(R=list(V=1,fix=1))
> m10.mc <- MCMCglmm(y~exp,data=bdata2,family="categorical",
+ prior=prior.def,verbose=F)#significant result but...
> summary(m10.mc)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 36.75788

 R-structure:  ~units

      post.mean l-95% CI u-95% CI eff.samp
units         1        1        1        0

 Location effects: y ~ exp

            post.mean l-95% CI u-95% CI eff.samp  pMCMC
(Intercept)   -0.2432  -1.1484   0.7405  321.737  0.632
exp2         -10.9369 -17.2524  -2.4858    6.142 <0.001 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> plot(m10.mc$Sol)#...extreme autocorrelation observed
```

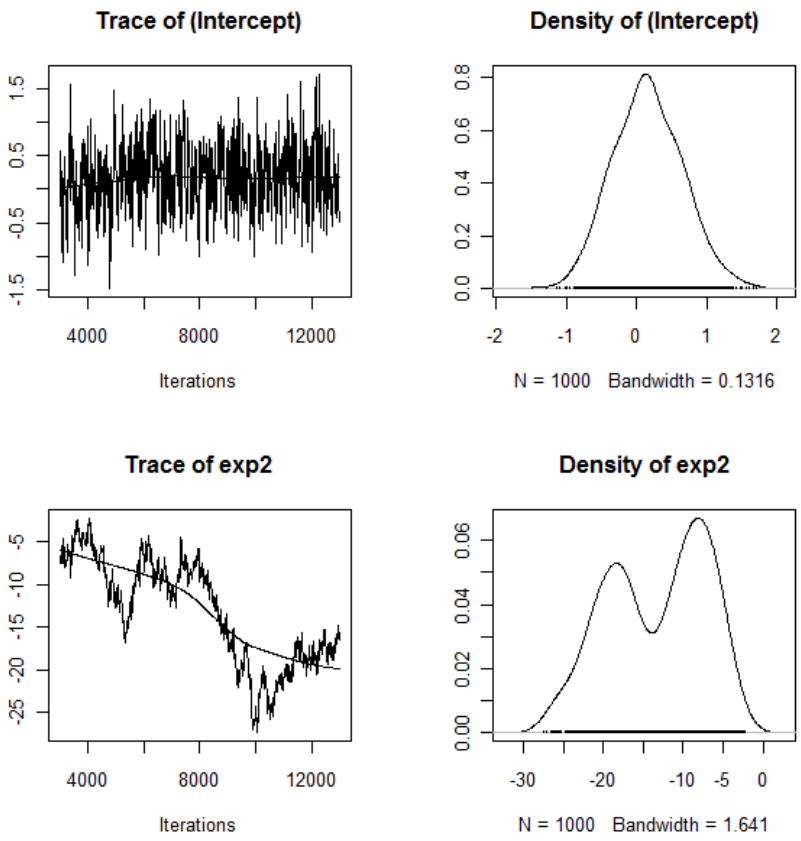


Fig 51: Bad priors cause extreme autocorrelation

```
> prior.better <- list(R=list(V=1,fix=1),
+ B=list(mu=c(0,0),V=diag(2)*(1+pi^2/3)))
> m10.mc2 <- MCMCglmm(y~exp,data=bdata2,family="categorical",
+ prior=prior.better, verbose=F)
> plot(m10.mc2$Sol)
> #looks much better but still may need running for longer
```

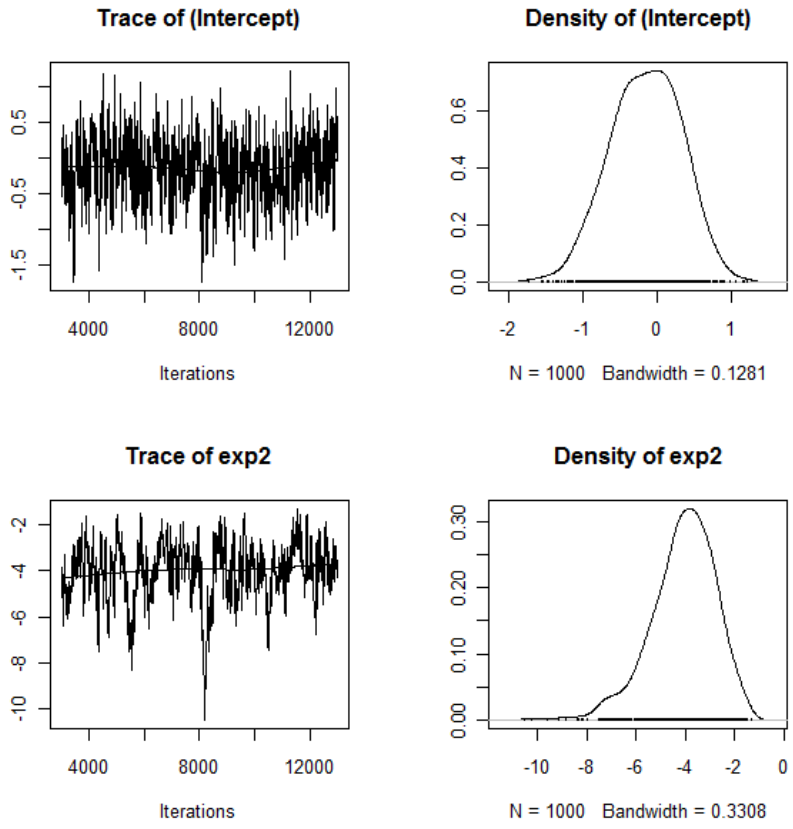


Fig 52: Employing good priors removes most problems

```
> #checking if the results conform to simpler test - exact binomial
> m10.test <- binom.test(table(bdata2)[2,2],25)
> m10.test

        Exact binomial test

data:  table(bdata2)[2, 2] and 25
number of successes = 0, number of trials = 25, p-value
= 5.96e-08
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.0000000 0.1371852
sample estimates:
probability of success
                     0


> predict(m10.mc2,interval="confidence")[26,]
        fit         lwr         upr
0.045980115 0.003148058 0.116423314
```

```
Warning message:
In predict.MCMCglmm(m10.mc2, interval = "confidence") :
  predict.MCMCglmm is still developmental - be careful
```

Closer look at categorical random interaction

We've heard something on random interactions in **lmer**. Here we'll extend this concept in **MCMCglmm** as it gives much greater control on (co)variance structures.

We could repeat our analysis when looking at the interaction between sex and dam in our system (BTdata). Previously we used two variance functions, allowing or not for non-zero covariances. Now we'll repeat this analyses to have a closer look at these results.

```
> ###code block 25

> #if you haven't - do the following
> library(MCMCglmm); data(BTdata)

> prior.a <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1,nu=0.002),
+ G2=list(V=diag(3),nu=0.002)))
> m11.bta <- MCMCglmm(tarsus~sex, random=~fosternest+idh(sex):dam,
+ prior=prior.a,verbose=F, data=BTdata)
> #using idh structure which sets covariances to zero
> plot(m11.bta$VCV)
> #UNK has low dam variance which may be problematic
```

We can see the actual matrix of correlations in the dam effects and its representation in the $\mathbf{R}^3$ space.

```
> ###code block 26

> Vdam.a <- diag(colMeans(m11.bta$VCV)[2:4])
> colnames(Vdam.a) <- colnames(m11.bta$VCV)[2:4]
> Vdam.a
       Fem.dam  Male.dam    UNK.dam
[1,] 0.1765957 0.0000000 0.00000000
[2,] 0.0000000 0.1715039 0.00000000
[3,] 0.0000000 0.0000000 0.05000367
>
> plotsubspace(Vdam.a,axes.lab=T)
Loading required package: rgl
> #elipsoid depicting this covariance structure
```

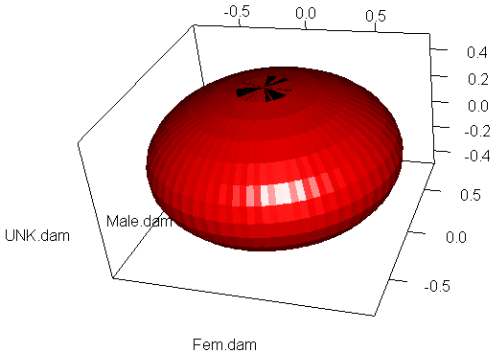


Fig 53: Elipsoid depicting covariance structure with correlations fixed at zero

The same can be done with an alternative definition of covariance structure using **`us()`**. We've used this function already but now we'll specify better prior. In general, priors for complex (co)variance structures depend on the particular structure (see next section).

```
> ###code block 27
>
> prior.b <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1,nu=0.002),
+ G2=list(V=diag(3)*0.02,nu=4)))
> m11.btb <- MCMCglmm(tarsus~sex, random=~fosternest+us(sex):dam,
+ prior=prior.b,
+ verbose=F, data=BTdata)
Warning message:
In MCMCglmm(tarsus ~ sex, random = ~fosternest + us(sex):dam, prior = prior.b,  :
  some combinations in us(sex):dam do not exist and 75 missing records have been
generated
> plot(m11.btb$VCV)

> Vdam.b <- matrix(colMeans(m11.btb$VCV)[2:10],3,3)
> colnames(Vdam.b) <- colnames(m11.btb$VCV)[2:4]
> Vdam.b
     Fem:Fem.dam Male:Fem.dam UNK:Fem.dam
[1,]   0.2319357    0.1994277   0.2237089
[2,]   0.1994277    0.2117362   0.2120803
[3,]   0.2237089    0.2120803   0.2889578
>
> plotsubspace(Vdam.b,axes.lab=T)
> #elipsoid depicting this covariance structure
```

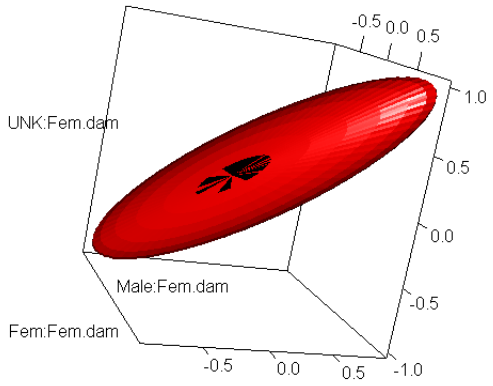


Fig 54: Elipsoid depicting covariance structure with correlations estimated

```
> plot(posterior.cor(m11.btb$VCV[,2:10])[,c(3,4,8)])
> #all r roughly equal to 1
```

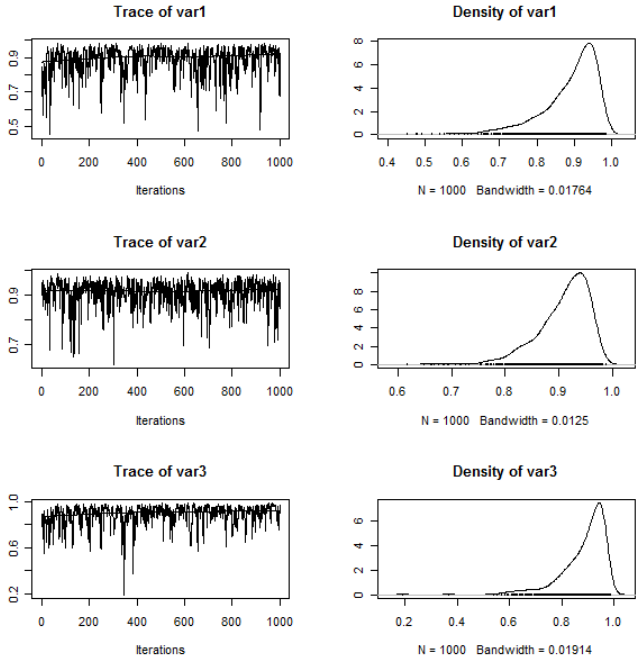


Fig 55: Posterior distributions of correlations

```
> "simpler model";m2.bt1$DIC
[1] "simpler model"
[1] 1992.66
> "us() variance structure";m11.btb$DIC
[1] "us() variance structure"
[1] 1997.765
> "idh() variance structure";m11.bta$DIC
[1] "idh() variance structure"
[1] 2037.151
```

As you can see both correlations are strong (almost 1) and variances are equal. Model with zero covariances is the worst, based on DIC values. Remaining two are similar but simpler one (equal variances and unity correlations) is better. In general, be careful when comparing models with different prior structures (as it was done here). DIC differences smaller than 2 should be treated with caution in such cases.

Priors for complex covariance structures

Complex variance structures have to take into account possible dependence of variances (which arises in case of non-zero covariances) (Hadfield, 2010b). For **idh()** variance structures it's simple: each variance in the structure is distributed independently, so new prior (**nu_** and **V_**, notation adopted from Hadfield (2010)) relates to a single-variance prior (**nu** and **V**) like this:

$\sigma_i^2 \sim IW($**nu_=nu**, **V_=V[1,1]**$)$

Hence prior specification in the example: **V=diag(3), nu=0.002**.

For **us()** structures it's more complicated:

$\sigma_i^2 \sim IW($**nu_=nu-dim(V)+1, V_=V[1,1]*nu/nu_**$)$

Consequently, we used **V=diag(3)*0.02 and nu=4**. We did use **nu=4** instead of usual **nu=4.002** and lower variance value to make this prior proper but also uninformative for correlation. We could alternatively use an improper prior, by setting **V=diag(dim(V))*0** and **nu=dim(V)-3**, but remember dangers of using improper priors.

Using inverse gamma distribution, with **shape=nu/2** and **scale=(nu*V)/2** we can actually visualize this prior for one of its elements:

```
> ###code block 28

> nu.star <- prior.b$G$G2$nu - dim(prior.b$G$G2$V)[1]*1
```

```
> V.star <- prior.b$G$G2$V[1,1]*(prior.b$G$G2$nu/nu.star)
> xv <- seq(1e-16,1,length=100)
> library(MCMCpack)
> dv<-dinvgamma(xv,shape=nu.star/2,scale=(nu.star*V.star)/2)
> detach(package:MCMCpack)
> plot(dv~xv,type="l")
```

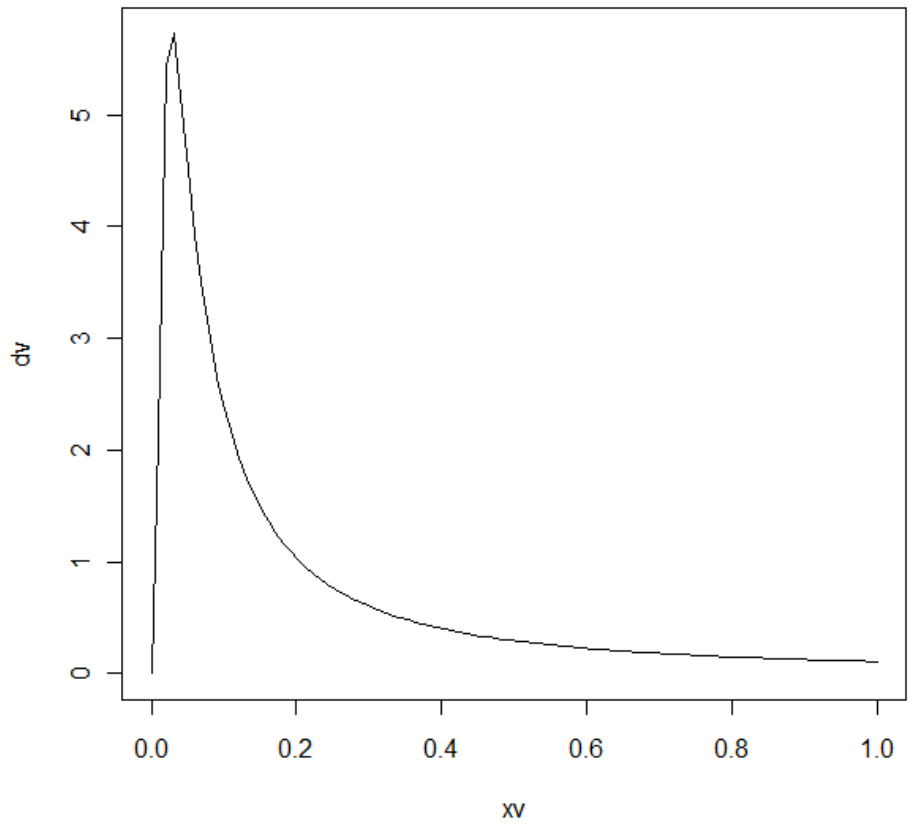


Fig 56: Inverse Wishart distribution; see text for details

Part C ~ Extending linear mixed models

Brief introduction to phylogenies in R

Phylogenies in some way are similar to pedigrees. They also represent relationships, however not between individuals, but between species or higher taxa. We may use them in **MCMCglmm** in exactly the same way as we did with pedigrees and thus build phylogenetic comparative models, accounting for variability that might have arisen from evolutionary history rather than genuine ecological/individual-based processes. First we'll learn how to build and handle phylogenies in R. We'll use the package **ape** and as its output objects can be directly handled by **MCMCglmm**.

We'll work with the mammals species phylogeny based on mammals super-tree and provided in Adams (2007).

```
> ###code block B6

> mammals <- read.nexus("mammals.nex")#read a tree in Nexus format
> mammals

Phylogenetic tree with 40 tips and 35 internal nodes.

Tip labels:
        Rattus_rattus, Sigmodon_hispidus, Peromyscus_eremicus, Peromyscus_maniculatus,
Neotoma_cinerea, Microtus_pennsylvanicus, ...

Rooted; includes branch lengths.
> summary(mammals)

Phylogenetic tree: mammals

  Number of tips: 40
  Number of nodes: 35
  Branch lengths:
    mean: 19.89730
    variance: 641.1923
    distribution summary:
   Min. 1st Qu.  Median 3rd Qu.    Max.
   0.10    3.10    9.10   26.52   94.50
  No root edge.
  First ten tip labels: Rattus_rattus
                        Sigmodon_hispidus
                        Peromyscus_eremicus
                        Peromyscus_maniculatus
                        Neotoma_cinerea
                        Microtus_pennsylvanicus
                        Microtus_montebelli
```

```
                      Chaetodipus_penicillatus
                      Dipodomys_ordii
                      Dipodomys_compactus
  No node labels.
> mammals.plot<-plot(mammals,font=1,cex=0.75)
> nodelabels()
```

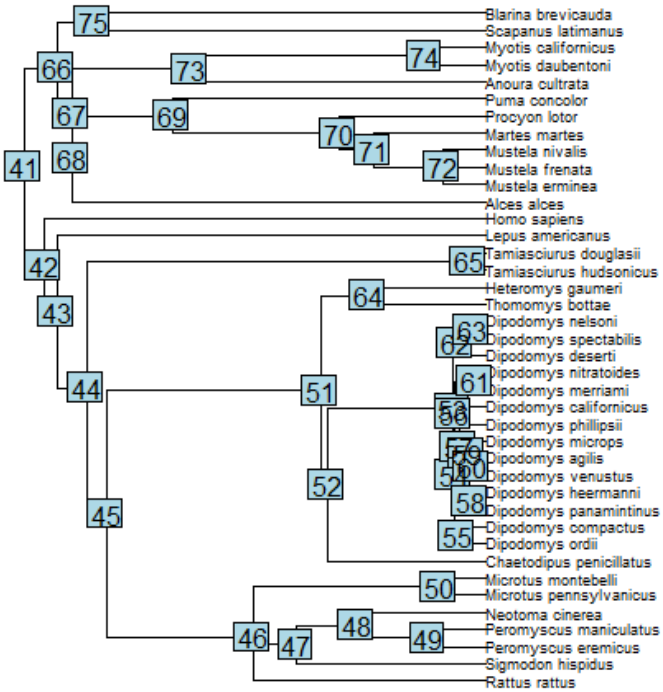


Fig 57: Mammals phylogeny

```
> #if you want you can write the tree in newick or nexus format
> write.nexus(mammals,file="mammals.nex")
> write.tree(mammals,file="mammals.nck")
```

If you don't have the tree and just have information to build one (e.g. DNA sequences) you can do this in **ape**. You can choose among different methods of clustering and different models of evolution.

```
> ###code block B7

> data(woodmouse)
> woodmouse
15 DNA sequences in binary format stored in a matrix.
```

```
All sequences of same length: 965

Labels: No305 No304 No306 No0906S No0908S No0909S ...

Base composition:
    a     c     g     t
0.307 0.261 0.126 0.306
> base.freq(woodmouse)
        a         c         g         t
0.3065414 0.2613083 0.1260264 0.3061239

> write.dna(woodmouse,"woodmouse.fas",format="fasta")#saving DNA data
> rodents <- read.dna("woodmouse.fas",format="fasta")#and reading it

> rodents[1,] #first sequence
1 DNA sequences in binary format stored in a matrix.

All sequences of same length: 965

Labels: No305

Base composition:
    a     c     g     t
0.304 0.262 0.129 0.306
> #here we can represent DNA sequence as text and paste single bases
> #together using no character as separator (collapse argument)
> paste(as.character(rodents[1,1:50]),collapse="")
 [1] "nttcgaaaaacacacccactactaaaanttatcagtcactccttcatcga"

> #calculate phylogeny based on these sequences
> dist.dna(rodents[1:5,])
             No305       No304       No306     No0906S
No304   0.015975800
No306   0.013815969 0.004210551
No0906S 0.019213434 0.013802125 0.009514854
No0908S 0.017059224 0.011665428 0.007391898 0.012726856
> rodents.dist<-dist.dna(rodents)
> as.matrix(dist.dna(rodents[1:5,])) # looks much better
            No305       No304       No306     No0906S     No0908S
No305   0.00000000 0.015975800 0.013815969 0.019213434 0.017059224
No304   0.01597580 0.000000000 0.004210551 0.013802125 0.011665428
No306   0.01381597 0.004210551 0.000000000 0.009514854 0.007391898
No0906S 0.01921343 0.013802125 0.009514854 0.000000000 0.012726856
No0908S 0.01705922 0.011665428 0.007391898 0.012726856 0.000000000

> #build a tree using UPGMA
> cluster<-hclust(rodents.dist)
> rodents.upgma<-as.phylo(cluster)
> plot(rodents.upgma,cex=0.75,font=1,no.margin=T)
```

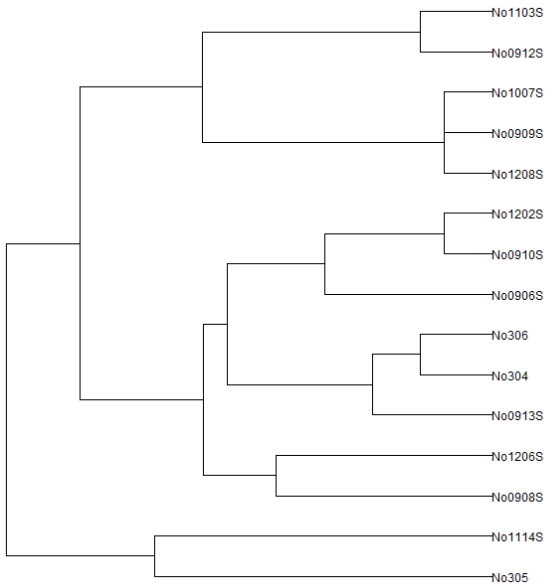


Fig 58: UPGMA tree

```
> #we can use neighbour joining instead
> cluster.nj<-nj(rodents.dist)
> rodents.nj<-as.phylo(cluster.nj)
> plot(rodents.nj,cex=0.75,font=1,no.margin=T)
```

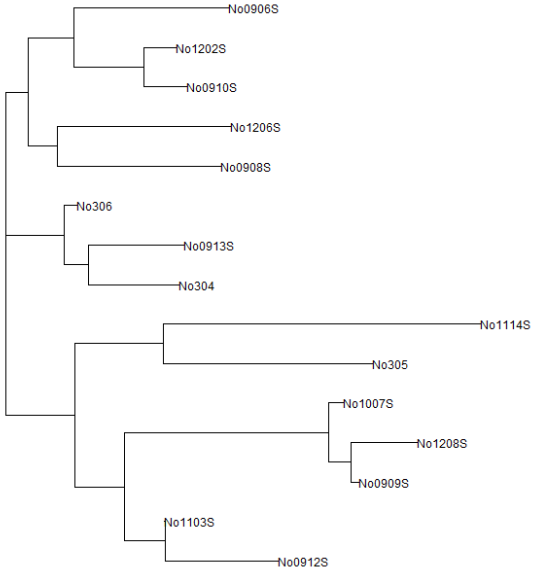


Fig 59: NJ tree

The package ape provides also improved Gascuel's NJ method **bionj()** and several evolution models for calculating distances between sequences (e.g. **model="JC69"**; other models are **"K81"**, **"F84"**, **"GG85"** – see **?dist.dna()** for more details).

Having several trees it's good to be able to compare them.

```
> ###codeblock B8

> #compare trees
> all.equal(rodents.nj,rodents.bionj)
[1] FALSE
> #and ignoring branch lengths - i.e. comparring only topologies
> all.equal(rodents.nj,rodents.bionj,use.edge.length=F)
[1] TRUE
> #topologies are the same

> #having a lot of trees you can calculate distances between them
> #here we use rtree() as in distribution functions to generate random
> #trees
> dist.topo(rtree(30),rtree(30))
[1] 54
```

However, real comparison of trees employs testing, either using bootstrapping or likelihood methods. Here we'll bootstrap our NJ tree. From those who are interested – see package **phangorn** which offers much more advanced functions for bootstrapping and ML-ing trees.

```
> ###code block B9

> #inside the bootstrapping function we must define the
> #tree-building function
> rodents.boot <- boot.phylo(rodents.nj,rodents,
+ function(x){nj(dist.dna(x))},
+ B=200,block=1)
> rodents.boot/2
 [1] 100.0  22.0  53.5  51.5  56.0  42.0  67.5
 [8] 65.5  87.5  90.0  87.0  99.5  59.0
> plot(rodents.nj)
> nodelabels(rodents.boot/2)
```

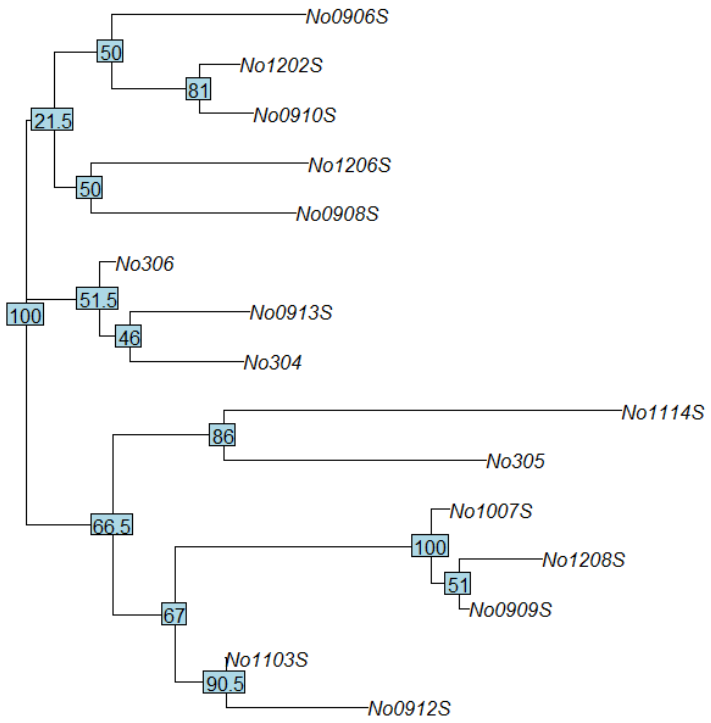


Fig 60: Mammalian tree with bootstrapping results

What are phylogenies for? If one has the data on some measurements done to several taxonomic units and a tree relating these taxonomic units one to another, it is possible to extract from these data phylogenetically independent information, in the form of phylogenetically independent contrasts (PICs). For more details see the documentation of the `pic()` function. However, currently PICs are not so widely used, mainly because mixed models allow for direct incorporation of phylogenetic information, similarly as it is done for genealogical information in animal model. How is it done?

Comparative analysis – simple simulated case

In the context of linear modelling phylogenies are used to remove any phylogenetic dependencies from our data. Most modern software packages allow for direct incorporation of phylogenetic information. Here we'll use simple simulated data on some hypothetical trait (called *y*). Evolution of this trait will be simulated across the phylogeny of bird families, supplied together with the **`ape`** package. First we'll load required packages and data.

```
> ### code block B13

> library(MCMCglmm); data(bird.families)
> bf.sim <- rTraitCont(bird.families,
+ sigma=runif(Nedge(bird.families),0.1,0.7))
> #see ?rTraitCont for more details
> bf.sim <- data.frame(y=bf.sim,animal=names(bf.sim))
> bf.sim[1:20,]
                        y         animal
Struthionidae  -0.60573476  Struthionidae
Rheidae        -2.68284592        Rheidae
Casuariidae     2.46897329    Casuariidae
Apterygidae     1.09753708    Apterygidae
Tinamidae       0.24638486      Tinamidae
Cracidae       -0.67455764       Cracidae
Megapodiidae    1.74500202   Megapodiidae
Phasianidae     0.61040649    Phasianidae
Numididae       1.65685699      Numididae
Odontophoridae  0.77349921 Odontophoridae
Anhimidae      -2.55968724      Anhimidae
Anseranatidae   1.37302713  Anseranatidae
Dendrocygnidae  0.05206279 Dendrocygnidae
Anatidae       -2.81114176       Anatidae
Turnicidae      2.64692783     Turnicidae
Indicatoridae  -0.16953127  Indicatoridae
Picidae        -0.91794680        Picidae
Megalaimidae    0.82388019   Megalaimidae
Lybiidae        1.38448623       Lybiidae
Ramphastidae    0.19388505   Ramphastidae

> #we'll add some residuals on top
> err <- rnorm(137,sd=sqrt(3)) #there are 137 points in the data
> bf.sim[,1]<-bf.sim[,1]+err

> #and replication
> bf.sim2 <- as.data.frame(bf.sim[sample(1:137,50),])
> err2 <- rnorm(50,runif(20,1,2),sqrt(3))
> bf.sim2[,1]<-bf.sim2[,1]+err2

> bf.sim3 <- as.data.frame(bf.sim[sample(1:137,50,replace=T),])
> err3 <- rnorm(50,runif(20,1.5,2.5),sqrt(3))
> bf.sim3[,1]<-bf.sim3[,1]+err3

> #and combine the three
> bf.sim<-rbind(bf.sim,bf.sim2,bf.sim3)
> summary(bf.sim)
       y                       animal
 Min.   :-6.6159   Casuariidae    :  4
 1st Qu.:-1.9046   Laridae        :  4
 Median : 0.2256   Pedionomidae   :  4
 Mean   : 0.2603   Acanthisittidae:  3
 3rd Qu.: 2.1467   Climacteridae  :  3
 Max.   : 9.9202   Cracidae       :  3
                   (Other)        :216
```

Analysing these data with phylogeny is as simple as fitting an animal model.

```
> prior.b52 <- list(R=list(V=1,nu=0.002),G=list(G1=list(V=1,nu=0.002)))
> m.b53 <- MCMCglmm(y~1,random=~animal,pedigree=bird.families,
+ data=bf.sim,verbose=F,prior=prior.b52)
Warning message:
In MCMCglmm(y ~ 1, random = ~animal, pedigree = bird.families, data = bf.sim,  :
  some combinations in animal do not exist and 134 missing records have been generated
> plot(m.b53$VCV) #phylogenetic signal is significant
```

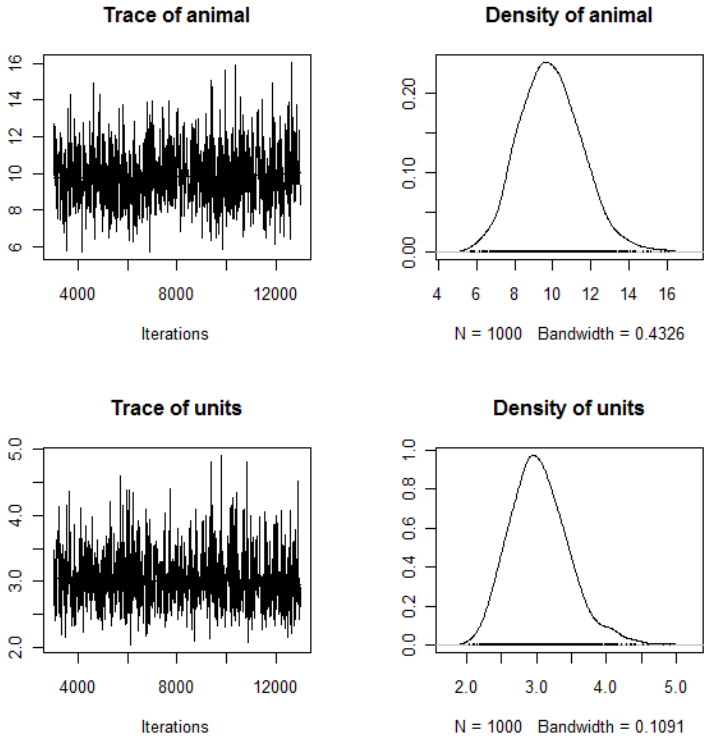


Fig 61: Comparative model; see large animal variance

```
> posterior.mode(m.b53$Sol)
(Intercept)
   1.040746
> m.b53a <- MCMCglmm(y~1,data=bf.sim,verbose=F)
> posterior.mode(m.b53a$Sol)
(Intercept)
  0.6232146

> #as in ordinary animal model we can estimate so
> #called phylogenetic heritability
> #the proportion of total variance explained by
> #phylogenetic effects of shared ancestry
```

```
> hp2 <- m.b53$VCV[,"animal"]/rowSums(m.b53$VCV)
> posterior.mode(hp2)
     var1
0.7875332
> HPDinterval(hp2)
        lower     upper
var1 0.680153 0.8452737
attr(,"Probability")
[1] 0.95
```

As you can see, not accounting for phylogeny may yield false picture of the reality underlying measured traits. We could expand this model and analyse more than one trait and see if they evolve in a correlated fashion across the phylogeny – which would be equivalent of calculating ordinary genetic correlation in an animal model framework.

Meta-analysis

Now we'll extend what we've learned so far and fit meta-analysis. Meta-analytical approach became very popular recently as it allows for answering very general questions. In its essence meta-analysis is very simple – instead of analysing raw data we take already calculated trends/statistics and look at their variability. In general, meta-analysis asks if predicted values of statistics holds after accounting for many studies, or if predicted relationship exists at the level of many studies. In such a case you assume that any error (residual variation) in our data is due to error in estimating statistics. In other words we can insert this error as some *a priori* known "residuals". Note, that sometimes meta-analyst is able to get accurate "raw" data from publications. In this case we use ordinary GLMM with response and estimated residual variance (one of the best examples is Cornwallis et al. (2010)).

Here we'll use example from Adams (2007). He examined if there are any body size clines in mammals, i.e. if mammals are larger in larger latitudes, where the climate is cooler. He gathered data on different mammal taxa from many papers, and for each paper he calculated effect size as the correlation between mammal body size and latitude. Following his paper and general strategy of meta-analysis we'll estimate measurement error (sampling variance of the statistic) based on the number of geographic locations from which data were available in each study. At first we'll try simple meta-analysis, ignoring any phylogenetic dependence of examined taxa.

```
> ### code block C1

> #if not loaded already:
> library(MCMCglmm)
> clines <- read.csv("mamm_clines.csv",head=T)
> clines<-clines[,-4]
```

```
> #effect size
> clines$FisherZ<-0.5*log((1+clines$corr)/(1-clines$corr))
> clines$mev<-1/(clines$N-3)#measurement error as variance

> prior.c1 <- list(R=list(V=1,nu=0.002))

> m.c11 <- MCMCglmm(FisherZ~1, verbose=F, prior=prior.c1,
+ data=clines)
> summary(m.c11)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 96.98406

 R-structure:  ~units

      post.mean l-95% CI u-95% CI eff.samp
units    0.6406   0.3784   0.9262     1000

 Location effects: FisherZ ~ 1

            post.mean l-95% CI u-95% CI eff.samp pMCMC
(Intercept)   0.29774  0.05018  0.52676     1000 0.012 *
---

Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> #we add sampling error of statistics mev
> m.c12 <- MCMCglmm(FisherZ~1, verbose=F, prior=prior.c1,
+ data=clines, mev=clines$mev)
> #note the mev argument typical for meta-analysis

Loading required package: polynom
> summary(m.c12)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000

 DIC: 88.50683

 R-structure:  ~units

                                          post.mean l-95% CI u-95% CI
leg(mev, -1, FALSE):leg(mev, -1, FALSE).meta         1        1        1
                                          eff.samp
leg(mev, -1, FALSE):leg(mev, -1, FALSE).meta        0

 Location effects: FisherZ ~ 1

            post.mean l-95% CI u-95% CI eff.samp pMCMC
(Intercept)   0.21998  0.01472  0.45575    961.3 0.044 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
> plot(m.c12) #check for autocorrelation problems
```

```
> #funnel plot below confirms overall positive effect
> plot(corr~N,data=clines,type="p",pch=20,ylab="Correlation",
+ xlab="N locations")
> abline(h=mean(clines$corr),lwd=1,lty=3)
```

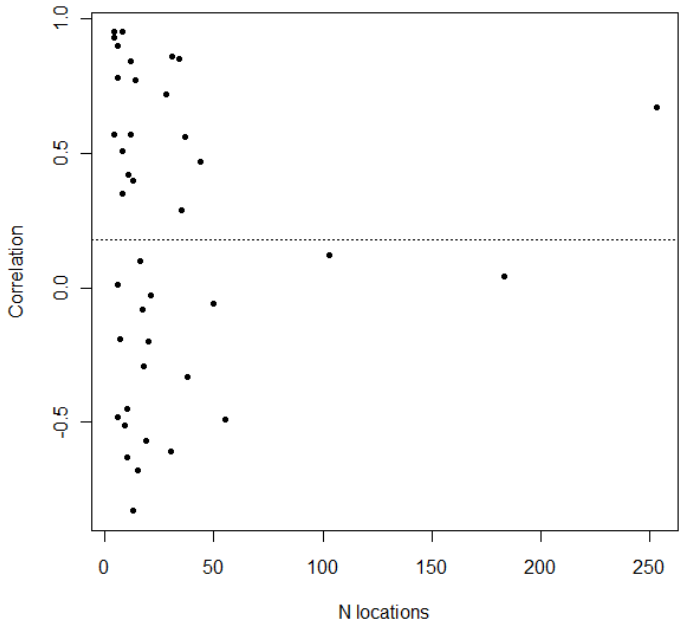


Fig 62: Funel-plot for Adams (2007)

It seems that there's an overall tendency in mammals to be bigger as they live further from the equator. If this phenomenon was due to ecological processes it might indicate that, as endotherms, mammals tend to be larger in cooler climate to conserve heat. Such pattern would thus indicate that during evolution mammals evolved this mechanism of saving body heat. However, such correlation of body size and latitudinal distribution could also arise simply during evolutionary history as a result of non-random migration patterns etc. If so we would expect that closely related species would show similar relationship of body size vs. latitude; in other words, in such a scenario phylogenetic variation would explain large proportion of variance in our effect size measures. To test this we perform comparative meta-analysis, taking into account phylogeny of mammals. As it turns out – the overall effect disappears clearly showing that any observed relationships are only due to shared evolutionary history.

```
> library(ape)
> mammals <- read.nexus("mammals.nex") #relevant phylogenetic tree
> plot(mammals,cex=0.75)
```

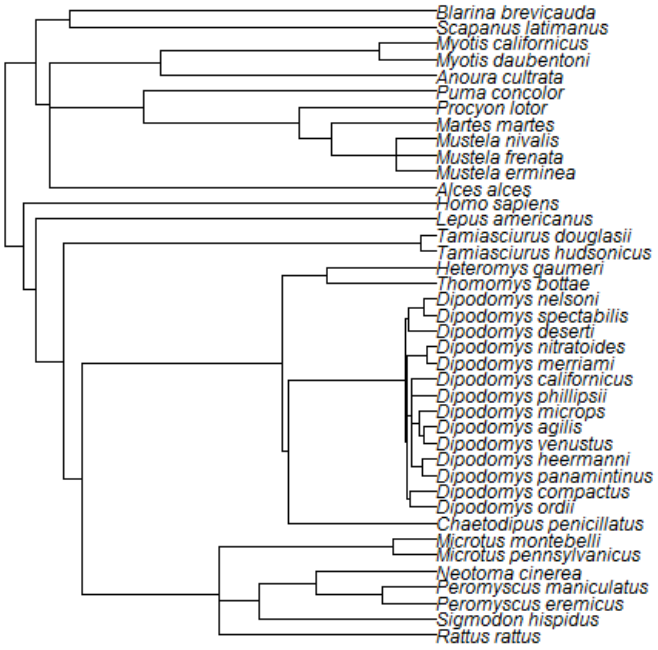


Fig 63: Phylogenetic tree form Adams (2007)

```
> names(clines)[1]<-"animal" #names of the taxa to the animal variable
> prior.c2 <- list(R=list(V=1,nu=0.002),
+ G=list(G1=list(V=1,nu=0.002)))
> m.c13 <- MCMCglmm(FisherZ~1, verbose=F, prior=prior.c2,
+ data=clines,
+ mev=clines$mev,random=~animal,
+ pedigree=mammals,
+ nitt=150000,burnin=30000,thin=150)
Warning message:
In MCMCglmm(FisherZ ~ 1, verbose = F, prior = prior.c2, data = clines,  :
```

```
  some combinations in animal do not exist and 34 missing records have been generated
```

```
> plot(m.c13$VCV) #phylogenetic signal is significant although small
```

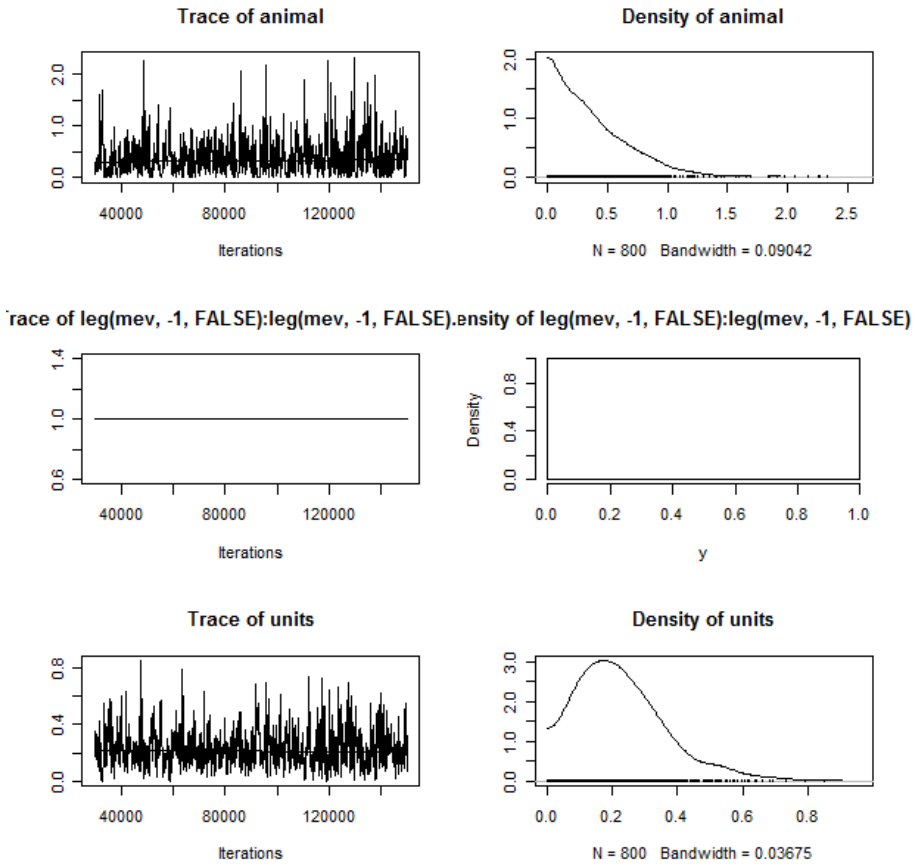


Fig 64: Random effects in comparative meta-analysis from Adams (2007)

```
> summary(m.c13)

 Iterations = 149851
 Thinning interval  = 30001
 Sample size  = 800

 DIC: 57.50496

 G-structure:  ~animal

       post.mean  l-95% CI u-95% CI eff.samp
```

```
animal    0.3906 0.0003283     1.05    546.9

 R-structure:  ~units

      post.mean  l-95% CI u-95% CI eff.samp
units    0.2281 0.0009862   0.5038    521.4

 Location effects: FisherZ ~ 1

            post.mean l-95% CI u-95% CI eff.samp pMCMC
(Intercept)    0.3414  -0.1611   0.9201      800 0.143



> diag(autocorr(m.c13$VCV)[2,,])
                                       animal
                                    0.1270430
leg(mev, -1, FALSE):leg(mev, -1, FALSE).meta
                                          NaN
                                        units
                                    0.1609003
> #should be ran for longer - >300000
> #BUT effect we looked for disappeared...
```

Random regression

In the basic part chapter we learned how to fit categorical random interactions. It's equal to allowing for differences in the intercept across the levels of random term. However, sometimes it is sensible to add also differences in slopes among individuals/units. Such models are called random regression models. We will use strategy from Hadfield (2010) for the longitudinal data on chicken growth. First, let's analyse it with a simple model. Fitting random effect of the chick id means that we want to have separate intercepts for each chick. As the data are not linear we will stick to some polynomial approximations of curvilinearity.

```
> ###code block B14

> #if not loaded:
> library(MCMCglmm)
> data(ChickWeight)
> xyplot(weight~Time|Chick,data=ChickWeight)
> #it visualises the effect of time on growth for all chicks
```

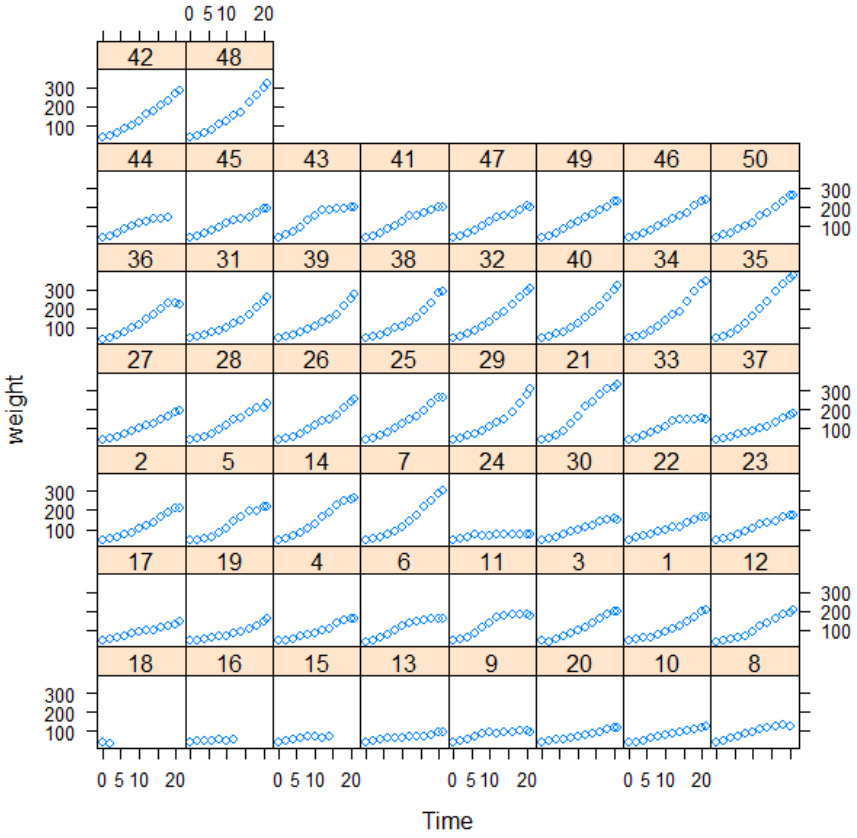


Fig 65: Growth curves for 50 chickens

```
> prior.b61 <- list(R=list(V=1e-16,nu=-2),G=list(G1=list(V=1,nu=1)))
> m.b61 <- MCMCglmm(weight~Diet+poly(Time,2,raw=T),random=~Chick,
+ data=ChickWeight, verbose=F, pr=T, prior=prior.b61,
+ saveX=T, saveZ=T)

> #simple random effect model with curvilinear pattern
> #we also save X and Z - fixed and random effects design matrices
> #all random effects (i.e. posteriors for BLUPs) using pr=T

> pop.int <- posterior.mode(m.b61$Sol[,1]) #overall intercept
> pop.slope <- posterior.mode(m.b61$Sol[,5]) #overall linear slope
> pop.quad <- posterior.mode(m.b61$Sol[,6]) #overall quadratic slope
> chick.int <- posterior.mode(m.b61$Sol[,c(7:56)]) #chicks' intercepts

> time <- ChickWeight$Time[1:12]

> plot(pop.int+pop.slope*I(time^1)+pop.quad*I(time^2)~time,
```

```
+ type="l",lwd=2,ylim=c(-50,400)) #plots overall population curve
> for(i in 1:50) {
+ lines(pop.int+chick.int[i]+pop.slope*I(time^1)+
+ pop.quad*I(time^2)~time,lty=3,col="red")
+ } #plots curves for each chocks with their specific intercepts
```

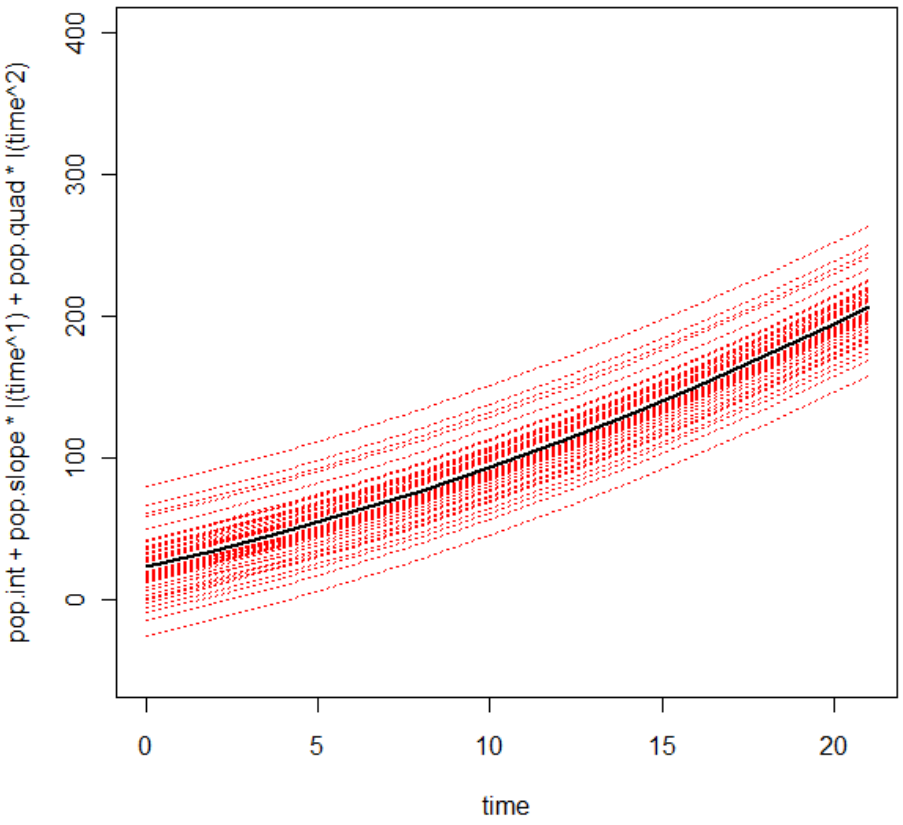


Fig 66: Population curve (thick black) and chick-specific curves (dotted red)

```
> #we can print predictions from our model for each chick by
> #multiplying design matrix W=[X,Z] for effects by
> #parameter vector theta=[beta,u]
> W1 <- cBind(m.b61$X,m.b61$Z)
> theta <- posterior.mode(m.b61$Sol)
> prediction1 <- W1 %*% theta #%*% means matrix product
> xyplot(weight+prediction1[,1]~Time|Chick,data=ChickWeight)
```

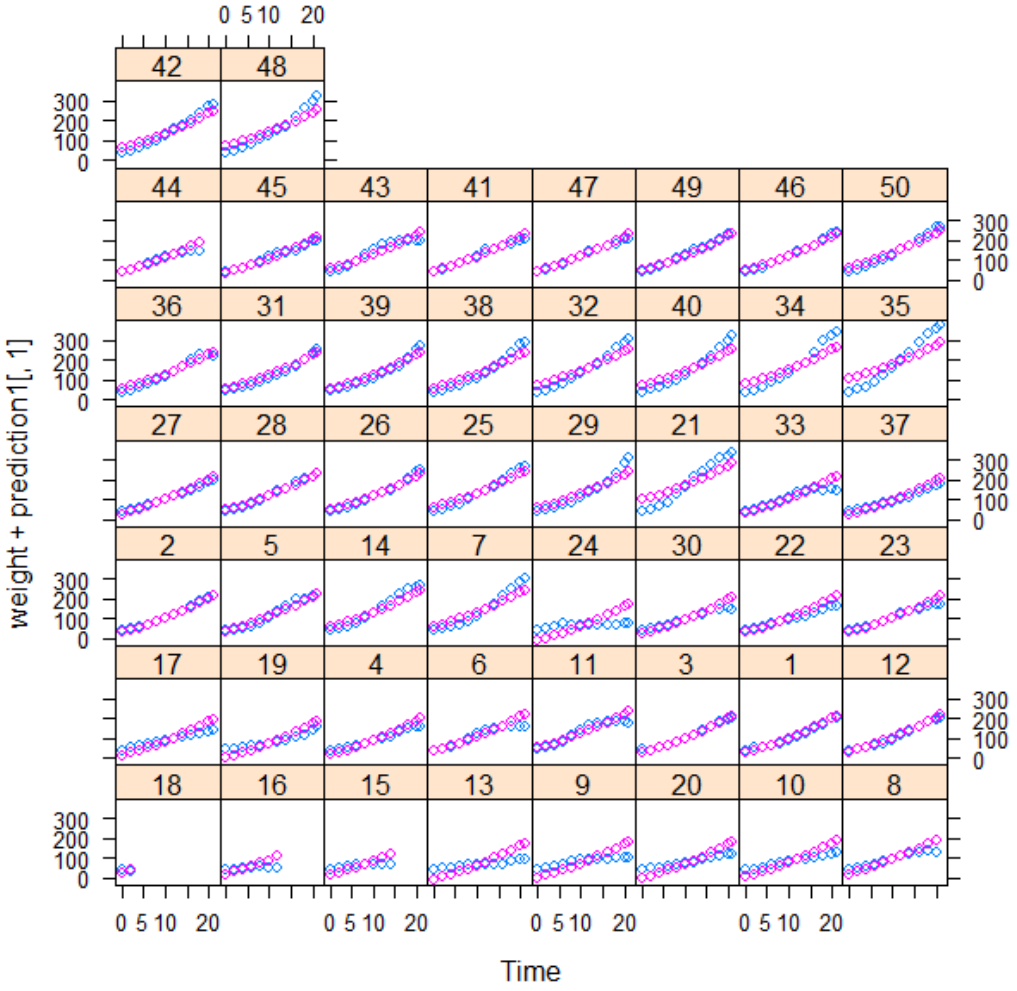


Fig 67: Chickens' growth curves with predictions added (pink, see PDF version)

As expected, model fits well, predictions look reasonable. However, slight differences are visible between predicted and real curves for some chicks. Thus, we might as well allow for differences in slopes between chicks. In the simpler model in random effects we fitted just single variance, i.e. σ^2 **`(Intercept).`** Interacting random term with both intercept and slope yields 2x2 covariance structure:

$$\mathbf{V_{Chick}} = \begin{bmatrix} \sigma^2_{\mathbf{(Intercept)}} & \sigma_{\mathbf{(Intercept),Time}} \\ \sigma_{\mathbf{(Intercept),Time}} & \sigma^2_{\mathbf{Time}} \end{bmatrix}$$

```
> prior.b62 <- list(R=list(V=1e-16,nu=-2),
+ G=list(G1=list(V=diag(2),nu=2)))
> #use matrix (hence diag()) since random effects have complex structure
> m.b62 <- MCMCglmm(weight~Diet+poly(Time,2,raw=T),
+ random=~us(1+Time):Chick,
+ data=ChickWeight, verbose=F,
+ pr=T,prior=prior.b62,saveX=T,saveZ=T)
> #use us() to allow for covariance between intercept and slope


> diag(autocorr(m.b62$VCV)[2,,]) #diagnostics
(Intercept):(Intercept).Chick        Time:(Intercept).Chick
                   0.10739398                    0.03065336
       (Intercept):Time.Chick               Time:Time.Chick
                   0.03065336                   -0.01052061
                        units
                  -0.01388139
> r.int.slope <- m.b62$VCV[,2]/sqrt(m.b62$VCV[,1]*m.b62$VCV[,4])
> posterior.mode(r.int.slope)
      var1
-0.9701405
> #correlation close to space boundary - should be run for longer

> #could do predictions by hand or like here by using predict()
> xyplot(weight+predict(m.b62,marginal=NULL)~Time|Chick,data=ChickWeight)
Warning message:
In predict.MCMCglmm(m.b62, marginal = NULL) :
  predict.MCMCglmm is still developmental - be careful
```

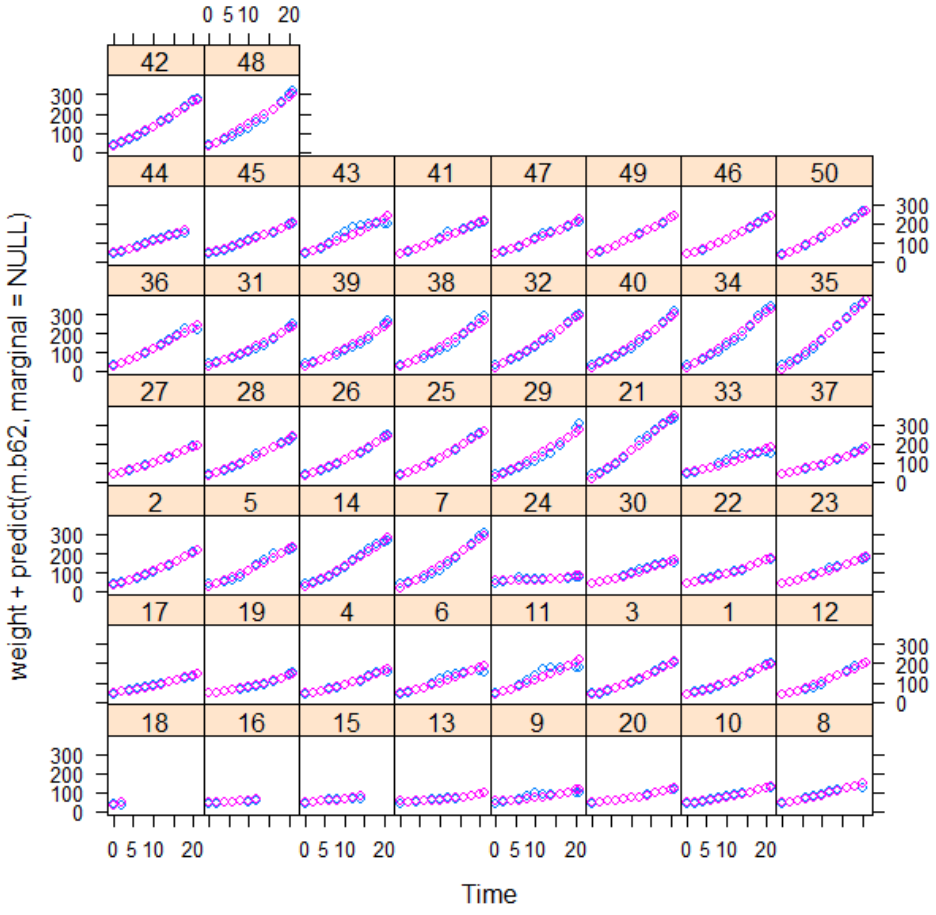


Fig 68: Chickens' growth curves with predictions from first random regression

```
> #looks MUCH better - may could be better
> #adding second random slope for quadratic term?
> prior.b63 <- list(R=list(V=1e-16,nu=-2),
+ G=list(G1=list(V=diag(3),nu=3)))
> m.b63 <- MCMCglmm(weight~Diet+poly(Time,2,raw=T),
+ random=~us(1+poly(Time,2,raw=T)):Chick,
+ data=ChickWeight, verbose=F, pr=T,
+ prior=prior.b63,saveX=T,saveZ=T)

> #DICs confirm it's the best model - hence chicks differ both in
> #intercepts and (quadratic)slopes
> m.b61$DIC;m.b62$DIC;m.b63$DIC
[1] 5525.139
[1] 4543.945
[1] 3933.421
```

```
> #to confirm we could see if REML estimators corroborate
> #these conclusions
> library(lme4)
> m.b61reml <- lmer(weight~Diet+poly(Time,2,raw=T)+(1|Chick),
+ data=ChickWeight)
> summary(m.b61reml)@AICtab[1]
      AIC
 5578.963

> m.b62reml <- lmer(weight~Diet+poly(Time,2,raw=T)+(1+Time|Chick),
+ data=ChickWeight)
> summary(m.b62reml)@AICtab[1]
      AIC
 4732.387

> m.b63reml <- lmer(weight~Diet+poly(Time,2,raw=T)+
+ (1+poly(Time,2,raw=T)|Chick),
+ data=ChickWeight)
> summary(m.b63reml)@AICtab[1]
      AIC
 4267.013
> detach(package:lme4)
```

Unfortunately, in pursue for the best model we forgot about one thing. In case of random slope models we should check not only if model is the best-fitting one, but also how well it's variance structure describes variance in the real data. Particularly, having intercept + *n* slopes fitted as random we expect that variance should change as the function of *n*-th degree with the continuous predictor (see Hadfield, 2010b for more detailed description of this example). We'll see how it works for toy data and then inspect our models. In general, from linear modelling theory, variance in the response should follow something like this: *Var*[*y*]=diag(**ZVZ'**) where **Z** is the design matrix for random effects and **V** is estimated covariance matrix. We can calculate this directly, having saved design matrices in our models (**`saveZ=T`**). However, here we'll create our own **Z** to avoid problems caused by duplication of records (we had several Diets and several Time points for every Chicken). We create hypothetical design matrix as if there was one chicken measured over 100 time points.

```
> toyslope <- rnorm(30)#30 random slopes ~N(0,1)
> #prepare space for the plots
> plot(0,type="n",xlim=c(-1,1),ylim=c(-3,3),ylab="y",xlab="time")
> for (i in 1:30) { #for each of 30 slopes
+ abline(a=0,b=toyslope[i]) #print its line
+ }
```

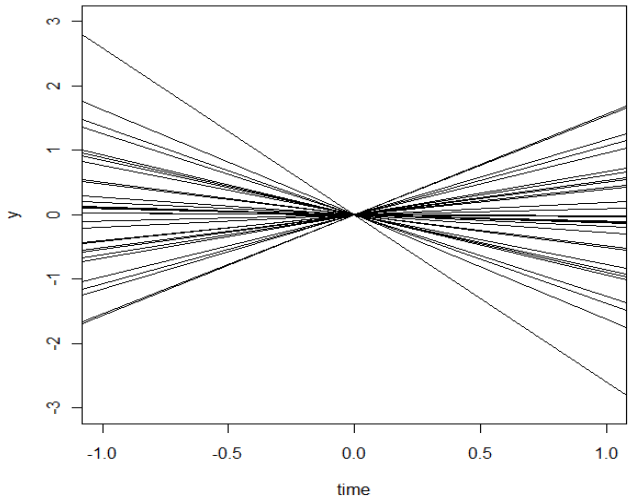


Fig 69: Simulated data with increasing variance

```
> time<-seq(0,21,length=100)
> polynomial<-leg(time,2,normalized=F)
> #better than poly because generates first column of ones giving
> #appropriate design matrix for fixed and random slope effects

> #coeficients for fixed effects from above 3 models
> beta1 <- c(posterior.mode(m.b61$Sol[,1]),posterior.mode(m.b61$Sol[,5]),
+ posterior.mode(m.b61$Sol[,6]))
> beta2 <- c(posterior.mode(m.b62$Sol[,1]),posterior.mode(m.b62$Sol[,5]),
+ posterior.mode(m.b62$Sol[,6]))
> beta3 <- c(posterior.mode(m.b63$Sol[,1]),posterior.mode(m.b63$Sol[,5]),
+ posterior.mode(m.b63$Sol[,6]))

> #covariance matrices and residuals from above 3 models
> VCV1 <- matrix(posterior.mode(m.b61$VCV)[1],1,1)#single variance
> VCV2 <- matrix(posterior.mode(m.b62$VCV)[1:(2^2)],2,2)#4 parameters
> VCV3 <- matrix(posterior.mode(m.b63$VCV)[1:(3^2)],3,3)
> #9 parameters
> units1 <- posterior.mode(m.b61$VCV)[2]
> units2 <- posterior.mode(m.b62$VCV)[5]
> #5th parameter cause 4 for (co)variances
> units3 <- posterior.mode(m.b63$VCV)[10]
> #10th cause 9 pars for (co)variances

> plot(weight~Time,data=ChickWeight,cex.lab=1.5) #point data

> mu1 <- polynomial %*% beta1 #population line across time
```

```
> sd1 <- sqrt(units1+diag(polynomial[,1,drop=F]%*%
+ VCV1%*%t(polynomial[,1,drop=F])))
> #%*% multiplies matrices; drop lets matrix be a matrix
> #after extracting one dimension, otherwise it would be a vector
> #and would cause problems when
> #trying to multiply to get ZVZ'; by using first column of
> #polynomial we create 'new' Z matrix appropriate for
> #the time sequence we have, of length
> #100 rather than 12

> lines(mu1~time,lwd=2) #adds population line
> lines(I(mu1+1.96*sd1)~time,lty=2,lwd=1,col="red")
> lines(I(mu1-1.96*sd1)~time,lty=2,lwd=1,col="red")
> #adds the error associated with population line
```

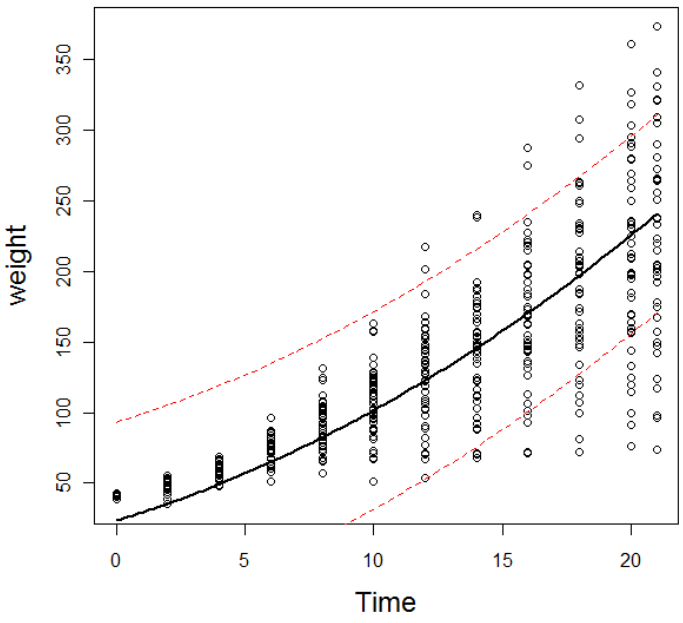


Fig 70: Estimated variance structure from simple mixed-effect model

```
> #very poor fit since SE is constant and points' scatter increases

> plot(weight~Time,data=ChickWeight,cex.lab=1.5)
> mu2 <- polynomial %*% beta2 #population line across time
> sd2 <- sqrt(units2+diag(polynomial[,1:2,drop=F]%*%VCV2%*%
+ t(polynomial[,1:2,drop=F])))
> lines(mu2~time,lwd=2)
```

```
> lines(I(mu2+1.96*sd2)~time,lty=2,lwd=1,col="red")
> lines(I(mu2-1.96*sd2)~time,lty=2,lwd=1,col="red")
> #very good fit of variance change to data
```

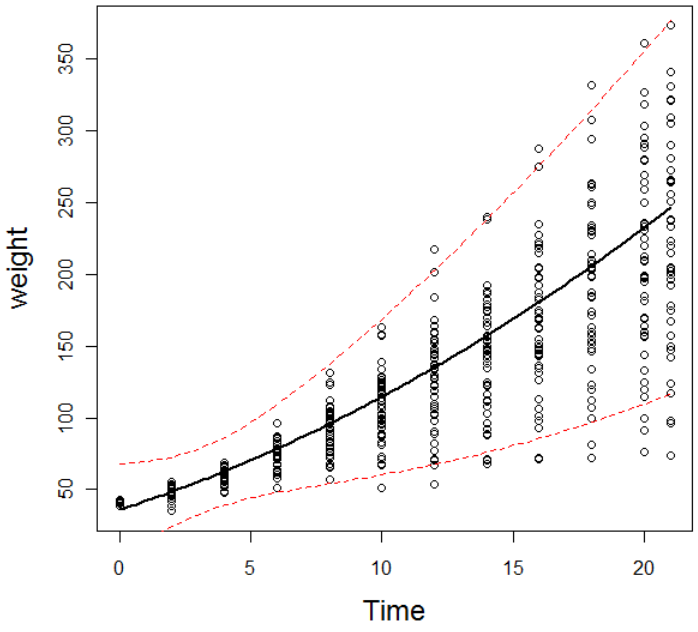


Fig 71: Estimated variance structure for linear random slope model

```
> #good fit of SE structure to points' variance
> plot(weight~Time,data=ChickWeight,cex.lab=1.5,ylim=c(-150,600))
> mu3 <- polynomial %*% beta3 #population line across time
> sd3 <- sqrt(units2+diag(polynomial[,1:3,drop=F]%*%VCV3%*%
+ t(polynomial[,1:3,drop=F])))
> lines(mu3~time,lwd=2)
> lines(I(mu3+1.96*sd3)~time,lty=2,lwd=1,col="red")
> lines(I(mu3-1.96*sd3)~time,lty=2,lwd=1,col="red")
```

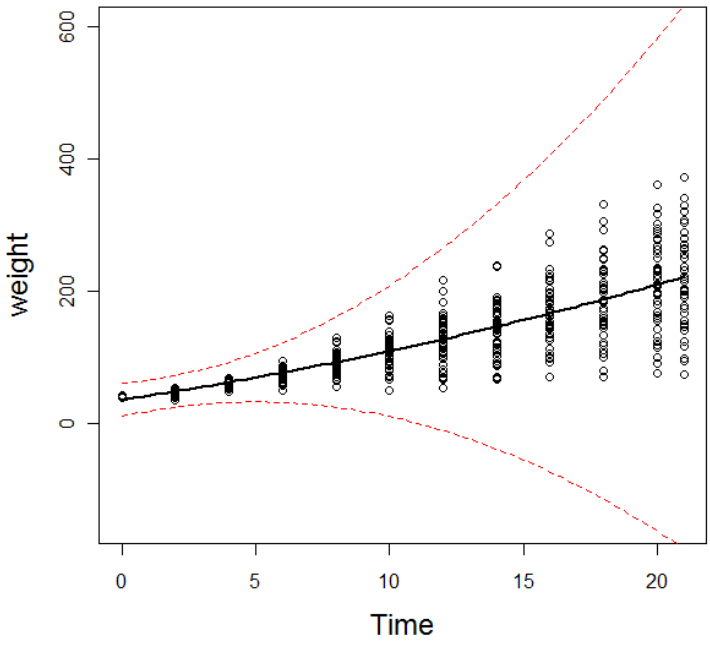


Fig 72: Estimated variance structure from quadratic random slope model

```
> #very poor  fit of variance change to data, 2nd model seems the best!
```

As you can see – the goodness of fit is not the only thing one should look for in continuous random effects – other factors such as variance homogeneity should also be considered.

Part D ~ Advanced applications of MCMCglmm

Parameter expanded priors

One drawback of using MCMC is it's randomness and sensitivity to the *i*-1th values of the chain. In practise it means that if in our model some variance components yield low values, close to zero, the chain may be trapped at some low value close to zero causing mixing-problems and in general poor convergence. It may also happen when some parameters, such as correlations, are close their space boundaries (-1 and 1). Such problems arise especially when residual variance *per se* cannot be estimated, as it is in binomial or Poisson models.

We can try alleviate these problems by using stronger priors – or improper priors. However, there's a much better solution called parameter expansion. Assume we have the design matrix $\mathbf{W}$ of the form $[\mathbf{X}\ \mathbf{Z_1}\ \mathbf{Z_2}\ \ldots\ \mathbf{Z_k}]$. We can rescale this matrix (and thus – whole MC-sampled parameter space) by some parameters $\boldsymbol{\alpha} = [1, \alpha_1, \alpha_2, \ldots \alpha_k]$. This would yield $\mathbf{W}_{\boldsymbol{\alpha}} = [\mathbf{X}\ \mathbf{Z_1}\ \alpha_1\ \mathbf{Z_2}\ \alpha_2\ \ldots\ \mathbf{Z_k}\ \alpha_k]$. With these alphas we would actually sample new location effects that could be rescaled to original values: $\boldsymbol{\theta} = (\mathbf{I}_{\beta} \oplus^{k}_{i=1} \mathbf{I}_{u\{i\}} \cdot \alpha_i)\boldsymbol{\theta}_{\alpha}$. Likewise, rescaling could also be applied to (co)variance matrices: $\mathbf{V} = Diag(\boldsymbol{\alpha}_V)\ \mathbf{V}_{\alpha}\ Diag(\boldsymbol{\alpha}_V)'$ (Hadfield, 2010b)

Here, we'll analyse data on sex-ration in blue tits (you already know this dataset) using both parameter-expanded and standard priors. We'll compare mixing properties of these runs. The expanded prior is the half-Cauchy distribution with the scale of 1000.

```
> ###code block C3

> #if not loaded:
> library(MCMCglmm)
> data(BTdata)

> #we'll remove unkown sex
> BTdata$sex[which(BTdata$sex=="UNK")]<-NA
> BTdata$sex<-gdata::drop.levels(BTdata$sex)
> #we remove UNK level from the variable

> prior.c31 <-list(R=list(V=1,fix=1),
+ G=list(G1=list(V=1,nu=0.002,
```

```
+ alpha.mu=0,alpha.V=1000))) #parameter-expanded prior
> prior.c32 <- list(R=list(V=1,fix=1),
+ G=list(G1=list(V=1,nu=0.002)))

> m.c4a <- MCMCglmm(sex~1,random=~dam,data=BTdata,
+ family="categorical",prior=prior.c31,verbose=F,
+ nitt=25000,burnin=5000,thin=25)

> m.c4b <- MCMCglmm(sex~1,random=~dam,data=BTdata,
+ family="categorical",prior=prior.c32,verbose=F,
+ nitt=25000,burnin=5000,thin=25)

> plot(mcmc.list(m.c4a$VCV[,"dam"],m.c4b$VCV[,"dam"]),
+ col=c("red","green"))
> effectiveSize(m.c4a$VCV[,"dam"])
    var1
354.4503
> effectiveSize(m.c4b$VCV[,"dam"])
    var1
163.9470
```

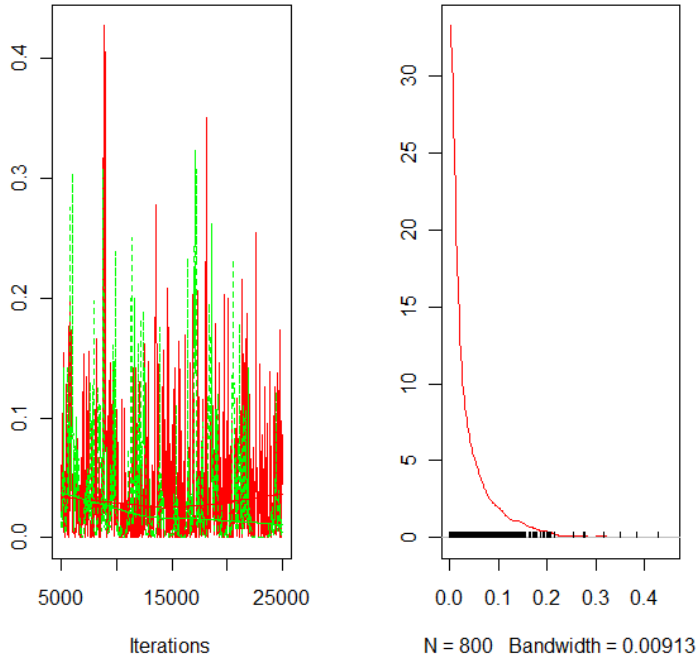


Fig 73: Traces from Wishart (green) and parameter expanded (red) priors (see PDF version)

Clearly, mixing of the non-expanded model is poorer (see the lower sample size). Also, the green trace on the above graph (see the electronic version) shows clear downward trend. The red trace (reflecting parameter expansion) is fairly constant, with much larger sample size associated. For more detailed formulation of parameter expansion and excellent discussion on the subject see Gellman (2006).

Zero-inflated models (ZIP) and zero-altered (ZAP) models

In biology often we end up with data where our treatments had no effect on the subject. It's especially apparent for count data, generated by Poisson processes, and binary data (binomial process). In such data, zeros occur often – and sometimes too often. In MCMCglmm there's one special class of distributions – zero-inflated distributions – to deal with zero-inflation. Models built using ZIP in fact analyse two separate traits (bivariate models). E.g. in zero-inflated Poisson (ZIP), first variable models probability from a Poisson process, and second models probability (binomial) that zero comes from a zero-inflated process (yes or no). We have to account for this structure of effects in our (co)variance structure, remembering that covariance between these two processes cannot be estimated as they never occur together in one data point (hence **`rcov=~idh(trait):units`**). To illustrate we will fit a ZIP model to data on PhD. Students' publishing rates, related to different features of their supervisors. Note two things: first – priors take into account bivariate nature of the model (the **`diag(2)`** function); secondly – in model we use **`at.level(trait,1)`** to indicate that we want to see the effect of all factors on the first level of the variable **`trait`** (i.e. only for the real Poisson process and not for the zero-inflated process).

```
> ### code block C6

> #if not loaded
> library(MCMCglmm)
> install.packages("pscl"); library(pscl)
> data(bioChemists)
> head(bioChemists) #see ?bioChemists for the description of the data
  art   fem     mar kid5  phd ment
1   0   Men Married    0 2.52    7
2   0 Women  Single    0 2.05    6
3   0 Women  Single    0 3.75    6
4   0   Men Married    1 1.18    3
5   0 Women  Single    0 3.75   26
6   0 Women Married    2 3.59    2

> #it seems there are lots of zeros in art (number of papers/year)
> sum(bioChemists$art==0)/length(bioChemists$art)#more than 30% are zeros
[1] 0.3005464
> #end we'd expect only 18% under Poisson process
> ppois(0,mean(bioChemists$art))
[1] 0.1839859
```

```
> prior.c71 <- list(R=list(V=diag(2),nu=0.002,fix=2))

> m.c71 <- MCMCglmm(art~trait-1+at.level(trait,1):fem+
+ at.level(trait,1):mar+
+ at.level(trait,1):kid5+at.level(trait,1):phd+at.level(trait,1):ment,
+ rcov=~idh(trait):units,data=bioChemists,prior=prior.c71,
+ family="zipoisson",verbose=F)
Warning message:
In MCMCglmm(art ~ trait - 1 + at.level(trait, 1):fem + at.level(trait,  :
  some fixed effects are not estimable and have been removed. Use singular.ok=TRUE to
sample these effects, but use an informative prior!

> plot(m.c71$Sol[,1:4]) #note poor mixing for zero-inflated process (2nd)
```

As you can see – fitting ZIP model is simple. However, even when we think we need ZIP, it may be not really necessary – as seen here, based on naïve quantiles or post-fitting check based on predicted values.

Alternative for ZIP models can be found and it's called Hurdle models. They're very similar to ZIP models in that they also model two variables. However, the first one models the probability from zero-truncated Poisson distribution (Poisson process without zeros; in ZIP it was just Poisson process distribution) and the second one models binary process (yes or not) that the response is zero (in ZIP that was probability that zero comes from zero-inflation). We will not cover Hurdle models here but they should be considered as a reasonable alternative for ZIP models. See Hadfield (2010b) for more details.

Finally – there are situation when we might want to model both zero-inflation and zero-deflation (less zeros than expected from Poisson process). These so called ZAP models are very useful and provide additional dimension in analyses. Similarly to ZIP models they are bivariate models. To fit a ZAP model one have to remember about two things: the formula for fixed effects should be expressed as a simultaneous interaction of the **`trait`** term and other terms; residuals should be in the form of **`trait:units`** interaction, ensuring equal amounts of overdispersion in both modelled traits. Below we fit simple ZAP model to the PhD data.

```
> ### code block C10

> m.c8 <- MCMCglmm(art~trait*(fem+mar+kid5+phd+ment),
+ rcov=~trait:units, data=bioChemists,
+ family="zapoisson", verbose=F)
> summary(m.c8)

 Iterations = 12991
 Thinning interval  = 3001
 Sample size  = 1000
```

```
 DIC: 3039.902

 R-structure:  ~trait:units

           post.mean l-95% CI u-95% CI eff.samp
trait:units    0.3761   0.2574   0.5019    41.08

 Location effects: art ~ trait * (fem + mar + kid5 + phd + ment)

                     post.mean  l-95% CI  u-95% CI eff.samp  pMCMC
(Intercept)           0.329840 -0.020848  0.635959    199.9  0.068 .
traitza_art          -0.530032 -1.060065  0.034144    119.8  0.074 .
femWomen             -0.201981 -0.372851 -0.043369    402.8  0.016 *
marMarried            0.094844 -0.100929  0.261683    365.9  0.310
kid5                 -0.140413 -0.268911 -0.010079    311.2  0.024 *
phd                   0.013362 -0.074771  0.090847    311.3  0.746
ment                  0.019686  0.012197  0.026658    444.5 <0.001 ***
traitza_art:femWomen  0.023808 -0.249587  0.288446    219.8  0.840
traitza_art:marMarried  0.145405 -0.135105  0.466840    233.6  0.324
traitza_art:kid5     -0.062199 -0.268586  0.154656    234.3  0.552
traitza_art:phd       0.007936 -0.137197  0.141371    132.5  0.904
traitza_art:ment      0.029625  0.012160  0.049018    105.7  0.002 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

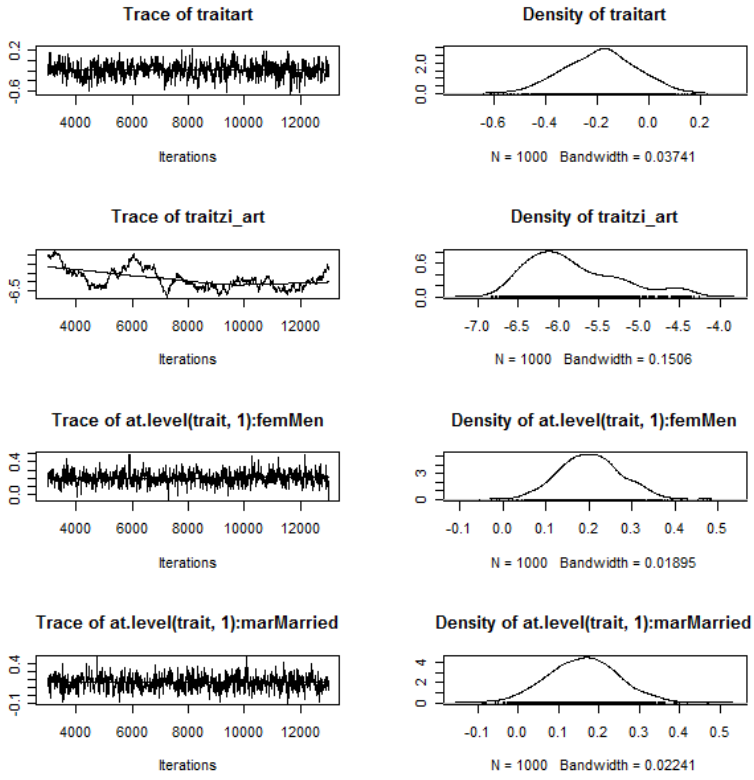


Fig 74: First page of fixed-effects posteriors from a ZIP model; note poor mixing of the ZI process (traitzi_art)

Model yields two sets of parameters: original coefficients for Poisson process and second set of coefficients for zero-altering process (if they're zero – there's no zero-altering; if they're negative – we detect zero-inflation, when they're positive – there's zero-deflation). Interpretation is simple – the more papers our mentor produces, the greater zero-deflation.

When thinking about ZIP/ZAP models it is important to avoid unnecessary complication. Although the data may look zero-inflated – simple check with observed and predicted by Poisson distribution numbers of zeros may not be enough (Hadfield, 2010b). In the above example the proportion of zeros is roughly twice as large as predicted by simple Poisson process. However we can use simple non-zero-inflated model to obtain predictions about the number of zeros – and then compare them to the observed number of zeros in the response variable. First – we fit simple non-ZIP model to the data. We save the design matrix X to be able to extract predictions:

```
> ### code block C6 continued

> prior.c72<-list(R=list(V=1,nu=0.002))
> m.c72 <- MCMCglmm(art~fem+mar+kid5+phd+ment,data=bioChemists,
+ prior=prior.c72,family="poisson",verbose=F,saveX=T)
```

After that we use solutions of fixed effects and the design matrix to obtain point predictions and then using these predictions we generate 1000 Poisson-distributed samples. Distribution of numbers of zeros in these samples clearly shows, that our data (represented by thick line) doesn't deviate from simple Poisson distribution (Hadfield, 2010b)

```
> ob.zer <- sum(bioChemists$art==0) #observed number of zeros
> nr.zer <- 1:1000 #place for bootstrapped samples
> for(i in 1:1000) {
+ pred1 <- rnorm(915,(m.c72$X%*%m.c72$Sol[i,])@x,sqrt(m.c72$VCV[i]))
+ nr.zer[i]<-sum(rpois(915,exp(pred1))==0)
+ }
> hist(nr.zer,breaks=20)
> abline(v=ob.zer,lwd=2)
```

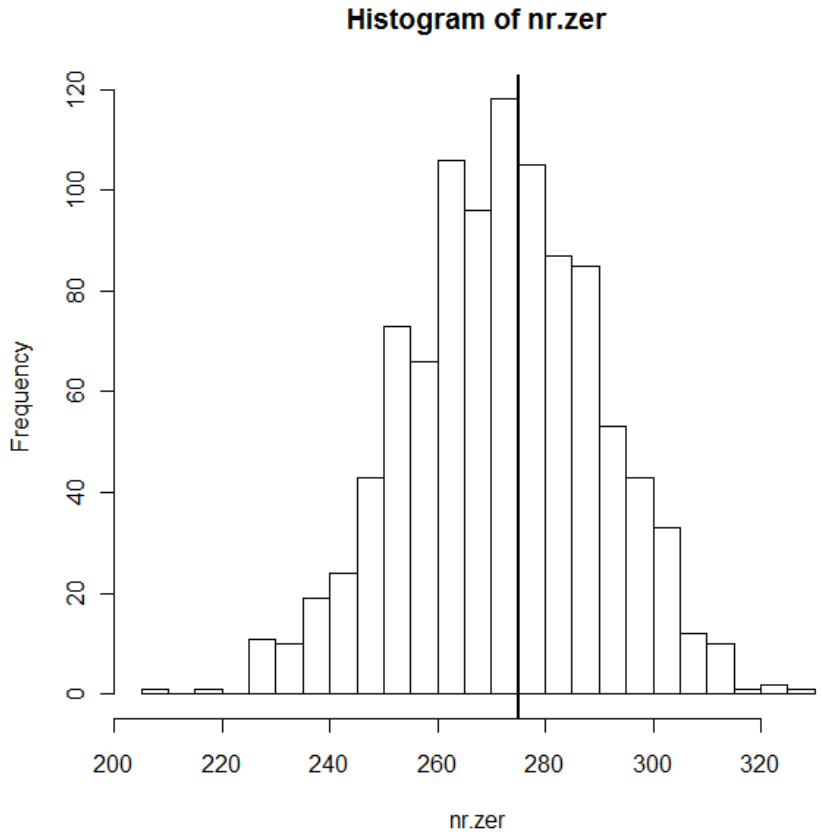


Fig 75: Histogram of samples from distribution of predicted numbers of zeros; black line represents observed number of zeros

Short Guide to the Most Essential R Functions

In the table below: LOGICAL – logical test, such as is.na(data) or data==1; ACTION – an executable expression, such as data<-3 or lm(y~x) or 2+4; BODY – set of expressions; path – access path to a file; [requires NAME] – installation of the NAME package is reuired; FORMULA – formula object; MODEL – model object; NAME – any custom NAME. Anu numbers indexing lists of commands in the first column are only for reference and should not be used with commands provided.

`!x, x|y, x&y, xor(x,y)`	NOT x, x OR y, x AND y, logical exclusive OR on x, y
`#`	Comment line – not executed
`+, -, *, /, %%, %/%, %*%, ^`	add, subtract, multiply, divide, modulo, integer division, matrix product, power
`==, >, <, >=, <=, !=`	Equal, smaller than, larger than, smaller or equal, larger or equal, not equal
`A -> B`	Assignemnt – B gets the value of A
`abs(NAME)`	Absilute value
`cor(NAME1,NAME2)`	Correlation of elemenets of two objects
`cov(NAME1,NAME2)`	Covariance of elements of two objects
`exp(NAME)`	Exponent (e^{NAME})
`Inf, NA, NaN`	Infinity, missing value, not-a-number variable
`install.packages("NAME")`	Install a package "NAME"
`is.na(NAME)`	Logical test if NAME is a missing value

`library(NAME)`	Load a library NAME
`list.files()`	List all files in the current working directory
`log(NAME)`	Logarithm of NAME
`ls()`	Display all object in the workspace
`mean(NAME)`	Mean of elements of name
`median(NAME)`	Median of elements of name
`prod(NAME)`	Product of elements of NAME
`quantile(NAME)`	Quantiles (median, minimum, maximum, 25% and 75% quantile)
`round(x, digits=n)`	Round x to n digits
`save(file="NAME")`	Save workspace to file
`savehistory(file="NAME")`	Save history of commands to file
`sd(NAME)`	Standard deviation of elements of NAME
`search()`	Display the namespace and all loaded packages and attached objects
`setwd(path)`	Set working directory to path
`sqrt(NAME)`	Square-root of NAME
`sum(NAME)`	Sum of elements of NAME
`T or TRUE, F or FALSE`	Logical variable – true or false
`var(NAME)`	Variance of elements of NAME
`as.vector(X), as.list(X), as.matrix(X), as.data.frame(X), as.array(X), as.numeric(X), as.character(X), as.logical(X) as.factor(X)`	Treat X as the type specified without changing its type
`c(a,b,c,d,...)`	Concatenate obejcts to a vector
`class(), attributes()`	Check class and attributes of an object
`cumprod(VECTOR)`	Cumulatiove product of elements of VECTOR

`cumsum(VECTOR)`	Cumulative sum of elements of VECTOR
`det(MATRIX)`	Determinant of MATRIX
`dim(ARRAY)`	Returns lengths of dimensions of ARRAY (may also be matrix and data-frame)
`eigen(MATRIX)`	Eigenvalue of MATRIX
`fix(NAME)`	Opens window for manual edition of the table NAME
`is.vector(), is.list, etc.`	Logical test if object is of type specified
`length(VECTOR)`	Number of elements in a VECTOR
`max(NAME)`	Maximum value of NAME
`min(NAME)`	Minimum value in NAME
`names(NAME)`	Names of the elements of the vector or variables of the data-frame – you can assign new values
`order(VECTOR)`	Returns permutation of elements that – when applied as and index – sorts elements of VECTOR ascending
`paste(VECTOR, sep=".")`	Paste elements of VECTOR as a text string using sep as separators (may also be "")
`range(VECTOR)`	The range of values
`rank(VECTOR)`	Ranks of values
`rev()`	Reverses a function, eg. Rev(sort(x)) sorts x descending
`rownames(NAME), colnames(NAME)`	Returns names of columnsand rows of thematrix or data-frame; may also be used for assigning names
`sort(VECTOR)`	Sorts elements ascending
`summary(NAME)`	Generic function, returns type-specific summary
`t(MATRIX)`	Transpose a matrix
`which(VECTOR, LOGICAL)`	Indexes of elements satysying the condition LOGICAL
`$ e.g. data$name`	Accesses the variable using its name (in data-frames)

`[] e.g. data[2,3]`	Accesses column, row or element; in >2D objects dimensions are specified in the order: rows, columns, …; omitting one dimension but retaining commas means that we want the whole dimension extracted
`apply(matrix, 1 or 2, FUNCTION)`	Applies FUNCTION to rows (1) or columns (2) of matrix
`1. attach(NAME), detach(NAME)` `2. detach(package:NAME)`	1. Attaches or detaches an object 2. Detaches package NAME
`boxcox(NAME)`	[requires MASS] Box-Cox transformation of the data
`cbind(x,y)`	Column-wise bind of two objects (numbers of rowns must be the same)
`na.omit(NAME)`	Returns object with NAs removed; in data-frame whole rows in at least one NA are removed
`rbind(x,y)`	Row-wise bind two objects; numbers of coulmns are the same
`read.csv(file=path)`	Read CSV (comma-separated) file
`read.delim2()`	Read file with commans as decimal separators; arguments as in read.table()
`read.table(path, header=T, sep="\t", skip=N)`	Read file in path, header=T sets the first line as names of variables, sep sets the character separating columns, skip skips N first columns
`subset(NAME, LOGICAL)`	Extract from data-frame NAME cases satisfying LOGICAL condition, eg. subset(data, sex=="M")
`table(group1, group2)`	Create contingency table counting cases in grouping variables (one or two)
`tapply(data, group, FUNCTION)`	Apply function to data group-wise
`with(NAME, procedures)`	Alternative for attach; procedures use data from NAME without the need of specifying variable names by $
`write.table(data, file=path, sep="\t")`	Save data to disc using filename path and sep as column separator

`break`	Break lood and go outside to the next operation
`F <- function(ARGUMENTS) {BODY}`	Define function F, taking several ARGUMENTS (names, comma separated), executing some expressions using these arguments in BODY
`for (i in X) {ACTIONS}` `for (i in X) ACTION`	Loop – iterate through elements of X (may be vector or range), for each execute ACTIONS or single ACTION

`1. if (LOGICAL) {ACTIONS}`
`2. if (LOGICAL) {ACTIONS}`
`else {ACTIONS}`
`3. ifelse (LOGICAL,`
`ACTIONS1, ACTIONS2)`
	1. Execute ACTIONS if LOGICAL is TRUE
	2. See above, if FALSE execute else
	3. Execute ACTIONS1 if LOGICAL is TRUE, execute ACTIONS2 otherwise

`next`	Stop iteration and go to the next one (does not break the entire loop)
`repeat {ACTION if (LOGICAL) break}`	Execute ACTION as long as LOGICAL remains false
`while (LOGICAL) {ACTIONS}`	Execute ACTIONS as long as LOGICAL remains TRUE
`rep(A, length.out=B, times=C, each=D)`	Repeat A C times, or as many times as necessary to fill length.out; if each defined – each element of A (if it's a vector) will be repeated D times; e.g. rep(c(1,2),times=2,each=4) yields 1111222211112222
`rnorm(N, mean, sd), pnorm(X, mean, sd), qnorm(P, mean, sd), dnorm(X, mean, sd)`	Use normal distribution with parameters mean and sd to: generate N random samples (r); get probability x<=X (p); get quantile X for P(x<=X) (q); get the density function for X (d); see help for more arguments, e.g. log=T yields log transformed values

OTHER DISTRIBUTIONS [add r, q, p or d; first argument may be P, X or N]: t(., df), f(., df1, df2), binom(., size, probab), pois(., lambda), gamma(., shape, scale), chisq(., df), nbinom(., size, probab, mu), lnorm(., meanlog, sdlog), hyper(., m, n, k), geom(., probab), multinom(., size, prob), logis(., location, scale), exp(., rate), cauchy(., location, scale), unif(., a, b)	*t* distribution, *F*, binomial, Poisson, gamma, Chi-squared, negative binomial, lognormal, hypergeometric, geometric, multinomial, logistic, exponential, Cauchy, uniform. See respective help files for more details and arguments.
rTraitCont(tree, model, sigma, alpha)	Simulate evolution along the tree phylogeny, using selected evolution model, sigma as standard deviation for random process at each branching and alpha as slelective force acting along the tree
rtree()	Generate random tree; see help for more details
sample(A, B, replace=T or F)	Choose random sample of size B from vector A, if replace TRUE each element will may be sampled more tha once; executing with replace=F and B>length(A) yields error
1. seq(A, B, by=C) 2. seq(A, B, length.out=C)	1. Generate numbers between A and B with increment of by 2. Generate sequence between A and B of the final length of length.out If A>B the sequence is generated in descending order
unique(A)	Extract all unique values from A
binom.test(n_succ, n_trials, P)	Binomial test for population with P successes

`chisq.test(x,y) or chisq.test(A)`	Accepts two vectors or a matrix (contingency table)
`cor.test(x,y,method)`	Correlation test; available methods: spearman, kendall, pearson
`fischer.test()`	Exact Fisher test, takes two vectors or one matrix
`kruskal.test()`	Kruskal-Wallis test; takes one list with groups as subvectors, two vectors – one with data nad one with group ids or formula object
`ks.test()`	Takes two vectors with data or one vector and the name of distribution to test (e.g. ks.test(x,pnorm)
`prop.test()`	Propotion test
`qqnorm(), qqline()`	Give quantile-quantile plot testing for normality and adds a line to it
`shapiro.test()`	Shapiro-Wilk test for normality, takes one vector of data
`t.test(A,B,var.equal=T or F, paired=T or F)`	t-test, takes two vectors of formula object
`TukeyHSD()`	Tukey Honest Significant Difference; takes anova or lm model object
`var.test()`	Takes two vectors and compares variances using F-test
`wilcox.test(A,B,paired=T or F)`	Wilcoxon signed-rank test – takes two vectors
`power.t.test(delta=A, sd=B, power=C, n=D, sig.level=E, alternative=F)`	Power calculation. Specify all parameters but one and it will be estimated based on the remaning ones. See help for detailed description of arguments.

`a <- numeric(N)`
 	
`for (i in 1:N) {`
`a[i] <- STATISTIC using sample(data,replace=T) }`
 	
`hist(a)`
 	
`quantile(a, c(0.025, 0.975))`
	Sample bootstrapping with N randomizations using sample function; STATISTIC is the expression calculating the value of test statistic; hist generates histogram of bootstrapped samples; quantile allow for hypothesis testing

FUNCTION <- function(A,i) STAT(A[i]) BOOT <- boot(data, FUNCTION, N)	[requires boot] First, the FUNCTION is defined – it calculates the test statistic. Then it is bootstrapped.
boot.ci(BOOT)	Confidence intervals from bootstrapping.
FORMULA 1. y ~ x 2. x + y 3. x:y 4. x*y 5. x – y 6. x/y 7. 1 8. (x + y + z)^2 9. poly(x, 2, raw=T) or x+ I(x^2) 10. s(x) 11. lo(x)	1. Simple formula, with independent (x) and dependent (y) variable 2. + defines additional variables 3. colon forms interaction 4. * fits interaction and all main effects 5. – removes a term 6. Slash defines nesting, from higher to lower level 7. One represents intercept 8. Fits all two factor interactions of x, y, z and main effects 9. Fits quadratic term of x 10. Uses smoother to fit x (in GAM) 11. Uses LOESS (local regression) to fit x (in GAM)
lm(FORMULA, data=NAME, weights=A)	Linear model for data, weights optional
predict(MODEL, newdata)	Prediction from model; if newdata specified (as additional data-frame) prdictions for new values are made
resid(MODEL)	Residuals from model
update(MODEL, ~. –A)	Update model's formula
summary(MODEL)	Summary of model
plot(MODEL)	Diagnostic plots
anova(MODEL)	ANOVA table for model (if supported)
anova(MODEL1, MODEL2)	Compare two models using ANOVA
gam(FORMULA, data)	[requires mgcv] Additive linear models
tree(FORMULA, data)	[requires tree] Tree regression models
plot(TREEMODEL), text(TREEMODEL)(	Plots tree regression and adds text labels

`step(MODEL)`	Stepwise simplification of MODEL based on AIC
`contrasts(DATA$FACTOR)`	Displays contrasts for factor variable
`contrasts(DATA$FACTOR) <- metrix of contrasts`	Sets contrasts for factor variable
`summary.lm(MODEL)`	Regression-like summary of a model
`summary.aov(MODEL)`	ANOVA-like summary of a model
`glm(FORMULA, data=NAME, family=distribution name)`	Generalized linear model with distribution defined by family; possible values: gaussian, poisson, binomial, exponential, gamma, quasibinomial, quasipoisson.

`MCMCglmm(`
 	
`y ~ fixed effects OR cbind(y, z) ~ trait + fixed effects,`
 	
`random=~a + b OR`
 	
`random=~idh(fixed):a + us(fixed):b OR`
 	
`random=~idh(trait):a + us(trait):b,`
 	
`rcov=~idh(fixed):units,`
 	
`data=NAME,`
 	
`pedigree=NAME,`
 	
`mev=NAME,`
 	
`family=NAMES OR family=c(NAME,NAME),`
 	
`prior=NAME,`
 	
`saveX=T or F, saveZ=T or F,`
 	
`pr =T or F, pl=T or F)`
	Fits generalized linear mixed models using Markov Chain Monte Carlo method
	Fixed effects formula; cbind() used if more than two response variables; trait is a restricted name indexing response variables in multivariate models
	Random effects formula; idh used for covariance structures with covariances set to zero; us used for (co)variance structures with covariances not fixed; in random effects – animal used for additive genetic/phylogenetic effect in animal models; be sure to create proper structure in multivariate models (hence the 'trait' effect)
	Optional, defines residual (co)variance structure
	Name of the data object
	Optional, name of the pedigree datafile/phylogenetic tree from ape()
	Optional, in meta-analysis defines vector of measurements error
	Defines the type of distribution; c() used when more than one response; not necessary if gaussian
	Defines the name of the prior
	Saves (if T) design matrices for fixed and random effects
	Saves (if T) random effects (BLUPs) and

	latent variables (fitted values on link scale)

`my_prior <-`
`list(R=list(V=1,nu=0.002),`
`B=list(mu=0, V=1e+06),`
`G=list(G1=list(V=1,nu=0.002),`
`G2=list(V=1,fix1),`
`G3=list(V=1,nu=0.002,alpha.mu=0,`
`alpha.V=1000)))`
	Prior for MCMCglmm; R – priors for residual variance; G – priors for random effects (as many as there are random terms); B – priors for fixed effects (if more than one: mu=c(0,0,0), V=diag(3)*1e+06); B is optional and required only in difficult models (such as binary data with large separation; see relevant chapters)

`fitted(MODEL)`	Returns values fitted by model (equal to predict() with no newdata argument)

`lmer(`
`y ~ x + y + (1|a) + (fixed|b),`
`family=distribution name,`
`data=NAME)`
	Fits (generalized) linear mixed models using REML; random effects formed by (X|...)

`mcmcsamp(MODEL from lmer)`	[requires arm] Uses lmer object to create MCMC samples for estimated parameters

`plot(x,y OR y~x OR object,`
 	
`main,`
`xlab,`
`ylab,`
`xlim,`
`ylim,`
`cex.axis,`
`cex.lab,`
`cex.main,`
`cex,`
`pch,`
`lty,`
`lwd)`
	Generic function for creating plots; takes two vectors (x and y variables), a formula object or a (model) object.
	Graph title
	x axis label
	y axis label
	Limits for x axis in the form of c(A,B)
	Limits for y axis
	Font size for axes' ticks in points
	Font size for axes' labels in points
	Font size for graph's label
	Size of graph's points
	Type of points (see points() function)
	Line type for line plots (see lines())
	Line width in pixels

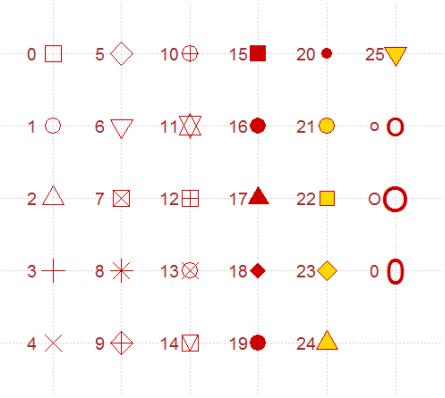
`abline(a=X,b=Y)` `abline(h=A)`	Adds line to a plot, by defining slope and intercept (a,b), horizontal line for Y=A,

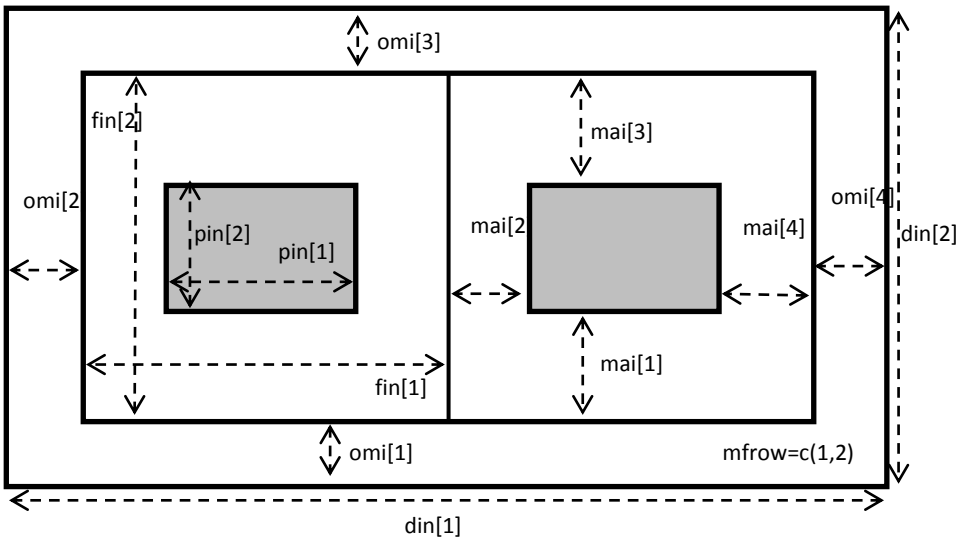
`abline(v=B)` `abline(lm model)`	vertical line for X=B or line from a lm object
`boxplot(Y~X)`	Creates boxplot fro data given group(X)-wise
`hist(X, freq=T or F, breaks=N)`	Histogram (with frequencies if freq=T), with custom number of bars (breaks)
`identify(x,y)`	Identifies points on the graph
`legend(x,y,legend)`	Adds a legend to the graph
`library(lattice) and` `library(gplot)`	Two libraries for high-level specialized graphs (see manuals and help files)
`lines(x,y,lty=N)`	Adds lines to a plot. Types of lines (lty): lty=1 solid line lty=2 dashed line lty=3 dotted line lty=4 dash-and-dot line lty=5 broken line lty=6 broken-and-dot line
`locator(x)`	Identifies points on the graph

`par(`
 	
`font,`
 	
`mar, mai,`
 	
`mfrow,`
 	
`oma, omi,`
 	
`din, fin, pin)`
 	
`mfg`
	Sets graphical parameters (see figure below)
	1-standard, 2-*italic*, 3-**bold**, 4-***bold italic***, 5-σψμβολ
	Width of margins in Inches or lines, as four-element vectors
	Sets number of columns and rows on the plot
	Widths of outer margins in Inches or lines, as four-element vectors
	Length and width of the image (in Inches or lines) as two-element vectors
	Position of active figure in device with multiple figures

`persp(x,y,z)`	3D plot, with x and y independent variables and one dependent variable z

`png(file=path) PLOTTING dev.off()`
`jpeg(file=path) PLOTTING`
`dev.off()`
`pdf(file=path) PLOTTING dev.off()`
	Using devices for saving graphs to graphic files; can also be done using Save As menu in the R Console (Windows/Mac OS)

`points(x, pch=N)`	Adds points to the graph. Type of pints (pch): 
`rainbow(N), heat.colors(N), terrain.colors(N), cm.colors(N)`	Generates color vectors of size = N



References

Adams D.C. 2007. Phylogenetic meta-analysis. Evolution, 62: 567-572.

Cornwallis et al. 2010. Promiscuity and the evolutionary transition to complex societies Nature, 466: 969-972.

Crawley, M. 2010. The R book. Wiley-Blackwell.

Davey, J. 2009. R course for the Ashworth Laboratories. Edinbourgh, UK.

Diggle, P. et al. 2004. Analysis of longitudinal data. Oxford University Press.

Gelman, A. 2006. Prior distributions for variance parameters in hierarchical models. Bayesian Analysis, 1: 515-533.

Gianola D. Sorensen D. 2004. Likelihood, Bayesian and MCMC methods in quantitative genetics. Springer.

Hadfield, J. 2010a. MCMC-based methods for multi-response generalized linear mixed models: the MCMCglmm R package. Journal of Statistical Software, 33: 1-22.

Hadfield, J. 2010b. Course Notes for the MCMCglmm package.

Michelson, A.A. 1880. Experimental determination of the velocity of light made at the U.S. Naval Academy, Annapolis. Astronomical Papers, 1: 109-145.

R Development Team. 2009. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Index

G

H

I

K

L

M

N

O