# A Little Book of R for Bioinformatics 2.0

Avirl Coghlan, with contributions by Nathan L. Brouwer

2021-07-27

# Contents

# Preface to version 2.0

Welcome to *A Little Book of R for Bioinformatics 2.0*!.

This book is based on the original *A Little Book of R for Bioinformatics* by Dr. Avril Coghlan (Hereafter "ALBRB 1.0"). Dr. Coghlan's book was one of the first and most thorough introductions to using R for bioinformatics, and was generously published under the Creative Commons 3.0 Attribution License (CC BY 3.0). In addition to describing how to do bioinformatics in R, Coghlan provided numerous functions to facilitate important tasks, practice questions, and references to further reading.

ALBRB 1.0 was extremely useful to me when I was learning bioinformatics and computational biology. In this version of the book, which I'll refer to as ALBRB 2.0, I have adapted Dr. Coghlan's original book to suit my own teaching needs.

Below I've outlined the general types of changes I've made to the original book. I have tried to link back to the original content that these updates are derived from and note how changes were made. Any errors or inconsistencies should be ascribed to me, not Dr. Coghlan. If you have any feedback, please email me at brouwern@gmail.com

Nathan Brouwer, June 2021

**Changes implemented in ALBRB 2.0 by Nathan Brouwer**

1. Converted the entire book to RMarkdown and published it via bookdown.
2. Added instructions for using RStudio and RStudio Cloud.
3. Updated instructions to reflect any changes in software, including changes to how the bioinformatics repository Bioconductor now works.
4. Split up chapters into smaller units.
5. Reorganized the order of some material.
6. Added links to the book I am developing, *Computational Biology for All*.
7. Moved most functions and datasets to my teaching package compbio4all.
8. Functions names changed from camelCase to underscore_format
9. Functions re-written so as not to use Bioconductor to reduce/eliminate dependency of compbio4all on Bioconductor
10. Changed some plotting to ggplot2 or ggpubr.
11. Added additional subheadings

12. Added vocab and function lists to the beginning of many chapters
13. At times replaced non-biological examples with biological ones.
14. Change from British to American English (Sorry!Couldn't help myself.)
15. Provided additional links to external resources.
16. Added use or `rentrez` for querying NCBI databases

# Chapter 1

# How to review this book

## 1.1   File format

These lessons are written in RStudio using RMarkdown. Each .RMd file is a mix of test, written in plain format, and **code chunks**, which look like this

Code chunks start with there apostrophes and {r}, like this: "'{r}. They end with three apostrophes"'. They will appear gray when opened up in RStudio but be white in the normal R code editor or other text editor.

## 1.2   What can you do?

- Read and fix typos : )
- Add comments within the file
- Email me general comments about the files (structure, topics, confusing parts etc)

## 1.3   Adding comments

You're welcome to add comments anywhere to the files, do the exercises and type up the key, propose you own exercises, etc.

### 1.3.1   HTML-tagged comments

The easiest way is to type a comment into the normal text part of the .Rmd file and then surround it with an html comment tag. A comment saying "A comment" will therefore look like this:

In RStudio you can type up a comment, highlight it then hit Shift+Control+C on a PC or Shift+Command+C on a mac.

### 1.3.2 Code chunk comments

Another way to add a comment is to make a RMarkdown code chunk thne type up your comments in it by commenting out each line, like the one below. Key to doing this is to put eval = F, echo = F in the braces after the r. eval = F, echo = F tells RStudio to leave that alone when the .Rmd file gets rendered into and web page or PDF.

In RStudio you can add code chunks with a shortcut key. On a Mac the shortcut is OPTION + COMMAND + I.

### 1.3.3 Keys to exercise

Some files will have exercises at the end. I don't always include the key, and you're welcome to try the exercise and type up a key (or fix any errors in mine). As for code chunk comments include eval = F, echo = F in the braces so the key won't appear when rendered. So a problem and its key would look something like this

### 1.3.4 Problem - fix this code

```
print(correct answer)
```

# Chapter 2

# Test audio link

Here is an example of an audio link embedded in a bookdown book.

```
knitr::include_url("https://www.genome.gov/sites/default/files/tg/en/narration/single_nucleotide_
                   height = "50px")
```

# Chapter 3

# Test shiny app

Here is an example of a Shiny app embedded in a bookdown book.

```
knitr::include_app(url = "https://wrightaprilm.shinyapps.io/treesiftr_app/")
```

# Chapter 4

# Test YouTube vidoe

Here is an example of a YouTube video embedded in a bookdown book.

via knitr chunk

```
knitr::include_url("https://www.youtube.com/watch?v=zNzZ1PfUDNk&feature=youtu.be")
```

# Chapter 5

# Test webpage insertion

```
knitr::include_url(url ="https://docs.google.com/spreadsheets/d/18dt7cYHAaszV5Iq8Y-7YOEFnv6hIXbD3
```

# Chapter 6

# Datacamp test

By default, `tutorial` will convert all R chunks.

```r
a <- 2
b <- 3

a + b
```

```
## [1] 5
```

# Chapter 7

# Downloading R

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 7.1  Preface

The following introduction to $R$ is based on the first part of "How to install $R$ and a Brief Introduction to R" by Avril Coghlan, which was released under the Creative Commons 3.0 Attribution License (CC BY 3.0). For additional information see the Appendices and "Getting $R$ onto your computer".

## 7.2  Introduction to R

$R$ (www.r-project.org) is a commonly used free statistics software. $R$ allows you to carry out statistical analyses in an interactive mode, as well as allowing programming.

## 7.3  Installing R

To use R, you first need to install the $R$ program on your computer.

### 7.3.1  Installing $R$ on a Windows PC

These instructions will focus on installing $R$ on a Windows PC. However, I will also briefly mention how to install $R$ on a Macintosh or Linux computer (see below).

27

**These steps have not been checked as of 8/13/2019 so there may be
small variations in what the prompts are. Installing R, however, is
basically that same as any other program. Clicking "Yes" etc on
everything should work.**

> **PROTIP:** Even if you have used $R$ before its good to regularly
> update it to avoid conflicts with recently produced software.

Minor updates of $R$ are made very regularly (approximately every 6 months),
as $R$ is actively being improved all the time. It is worthwhile installing new
versions of $R$ a couple times a year, to make sure that you have a recent version
of $R$ (to ensure compatibility with all the latest versions of the $R$ packages that
you have downloaded).

To install $R$ on your **Windows** computer, follow these steps:

1. Go to https://cran.r-project.org/
2. Under "Download and Install R", click on the "Windows" link.
3. Under "Subdirectories", click on the **"base"** link.
4. On the next page, you should see a link saying something like "Download
   $R$ 4.1.0 for Windows" (or $R$ X.X.X, where X.X.X gives the version of the
   program). Click on this link.
5. You may be asked if you want to save or run a file "R-x.x.x-win32.exe".
   Choose "Save" and save the file. Then double-click on the icon for the file
   to run it.
6. You will be asked what language to install it in.
7. The $R$ Setup Wizard will appear in a window. Click "Next" at the bottom
   of the $R$ Setup wizard window.
8. The next page says "Information" at the top. Click "Next" again.
9. The next page says "Select Destination Location" at the top. By default,
   it will suggest to install $R$ on the C drive in the "Program Files" directory
   on your computer.
10. Click "Next" at the bottom of the $R$ Setup wizard window.
11. The next page says "Select components" at the top. Click "Next" again.
12. The next page says "Startup options" at the top. Click "Next" again.
13. The next page says "Select start menu folder" at the top. Click "Next"
    again.
14. The next page says "Select additional tasks" at the top. Click "Next"
    again.
15. $R$ should now be installing. This will take about a minute. When $R$
    has finished, you will see "Completing the $R$ for Windows Setup Wizard"
    appear. Click "Finish".
16. To start R, you can do one of the following steps:
17. Check if there is an "R" icon on the desktop of the computer that you are
    using. If so, double-click on the "R" icon to start R. If you cannot find an
    "R" icon, try the next step instead.
18. Click on the "Start" button at the bottom left of your computer screen,
    and then choose "All programs", and start $R$ by selecting "R" (or $R$ X.X.X,

where X.X.X gives the version of R) from the menu of programs.

19. The *R* console (a rectangle) should pop up:

### 7.3.2 How to install *R* on non-Windows computers (eg. Macintosh or Linux computers)

**These steps have not been checked as of 8/13/2019 so there may be small variations in what the prompts are. Installing R, however, is basically that same as any other program. Clicking "Yes" etc on everything should work.**

The instructions above are for installing *R* on a Windows PC. If you want to install *R* on a computer that has a non-Windows operating system (for example, a Macintosh or computer running Linux, you should download the appropriate *R* installer for that operating system at https://cran.r-project.org/ and follow the *R* installation instructions for the appropriate operating system at https://cran.r-project.org/doc/FAQ/R-FAQ.html#How-can-R-be-installed_003f .

## 7.4 Starting *R*

To start R, Check if there is an *R* icon on the desktop of the computer that you are using. If so, double-click on the *R* icon to start *R*. If you cannot find an *R* icon, try the next step instead.

You can also start *R* from the Start menu in Windows. Click on the "Start" button at the bottom left of your computer screen, and then choose "All programs", and start *R* by selecting "R" (or *R* X.X.X, where X.X.X gives the version of R, e.g.. *R* 2.10.0) from the menu of programs.

Say "Hi" to *R* and take a quick look at how it looks. Now say "Goodbye", because we will never actually do any work in this version of *R*; instead, we'll use the **RStudio IDE (integrated development environment)**.

# Chapter 8

# Installing the RStudio IDE

**By:** Nathan Brouwer

The name "R" refers both to the programming language and the program that runs that language. When you download *it*R* there is also a basic **GUI** (graphical user interface) that you can access via the *R* icon.

Other GUIs are available, and the most popular currently is **RStudio.** RStudio a for-profit company that is a main driver of development of R. Much of what they produce has free basic versions or is entirely free. They produce software (RStudio), cloud-based applications (**RStudio Cloud**), and web server infrastructure for business applications of R.

A brief overview of installing RStudio can be found here "Getting RStudio on to your computer"

## 8.1   Getting to know RStudio

For a brief overview of RStudio see "Getting started with RStudio"

A good overview of what the different parts of RStudio can be seen in the image in this tweet: https://twitter.com/RLadiesNCL/status/1138812826917724160?s=20

## 8.2   RStudio versus RStudio Cloud

RStudio and RStudio cloud work almost identically, so anything you read about RStudio will apply to RStudio Cloud. RStudio is easy to download an use, but RStudio Cloud eliminates even the minor hiccups that occur. Free accounts with RStudio Cloud allow up to 15 hours per month, which is enough for you to get a taste for using R.

# Chapter 9

# Installing $R$ packages

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

$R$ is a programming language, and **packages** (aka **libraries**) are bundles of software built using $R$. Most sessions using $R$ involve using additional $R$ packages. This is especially true for bioinformatics and computational biology.

> **NOTE**: If you are working in an RStudio Cloud environment organized by someone else (e.g. a course instructor), they likely are taking care of many of the package management issues. The following information is still useful to be familiar with.

## 9.1 Downloading packages with the RStudio IDE

There is a point-and-click interface for installing $R$ packages in RStudio. There is a brief introduction to downloading packages on this site: http://web.cs.ucla.edu/~gulzar/rstudio/

I've summarized it here:

1. "Click on the"Packages" tab in the bottom-right section and then click on "Install". The following dialog box will appear.
2. In the "Install Packages" dialog, write the package name you want to install under the Packages field and then click install. This will install the package you searched for or give you a list of matching package based on your package text.

## 9.2  Downloading packages with the function `install.packages()`

The easiest way to install a package if you know its name is to use the *R* function `install.packages()`'. Note that it might be better to call this "download.packages" since after you install it, you also have to load it!

Frequently I will include `install.packages(...)` at the beginning of a chapter the first time we use a package to make sure the package is downloaded. Note, however, that if you already have downloaded the package, running `install.packages(...)` will download a new copy. While packages do get updated from time to time, but its best to re-run `install.packages(...)` only occassionaly.

We'll download a package used for plotting called `ggplot2`, which stands for "Grammar of Graphics." `ggplot2` was developed by Dr. Hadley Wickham, who is now the Chief Scientists for RStudio.

To download `ggplot2`, run the following command:

```r
install.packages("ggplot2") # note the " "
```

Often when you download a package you'll see a fair bit of angry-looking red text, and sometime other things will pop up. Usually there's nothing of interest here, but sometimes you need to read things carefully over it for hints about why something didn't work.

## 9.3  Using packages after they are downloaded

To actually make the functions in package accessible you need to use the `library()` command. Note that this is *not* in quotes.

```r
library(ggplot2) # note: NO " "
```

# Chapter 10

# Installing Bioconductor

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0), including details on install Bioconductor and common prompts and error messages that appear during installation.

## 10.1 Bioconductor

*R* **packages** (aka "libraries") can live in many places. Most are accessed via **CRAN**, the **Comprehensive R Archive Network**. The bioinformatics and computational biology community also has its own package hosting system called Bioconductor. *R* has played an important part in the development and application of bioinformatics techniques in the 21th century. Bioconductor 1.0 was released in 2002 with 15 packages. As of winter 2021, there are almost 2000 packages in the current release!

> **NOTE**: If you are working in an RStudio Cloud environment organized by someone else (eg a course instructor), they likely are taking care of most of package management issues, inlcuding setting up Bioconductor. The following information is still useful to be familiar with.

To interface with Bioconductor you need the BiocManager package. The Bioconductor people have put BiocManager on CRAN to allow you to set up interactions with Bioconductor. See the BiocManager documentation for more information (https://cran.r-project.org/web/packages/BiocManager/vignettes /BiocManager.html).

Note that if you have an old version of R you will need to update it to interact with Bioconductor.

## 10.2   Installing BiocManager

BiocManager can be installed using the `install.packages()` packages command.

```
install.packages("BiocManager") # Remember the "  "; don't worry about the red text
```

Once downloaded, BioManager needs to be explicitly loaded into your active R session using `library()`

```
library(BiocManager) # no quotes; again, ignore the red text
```

Individual Bioconductor packages can then be downloaded using the `install()` command. An essential packages is `Biostrings`. To do this ,

```
BiocManager::install("Biostrings")
```

## 10.3   The ins and outs of package installation

**IMPORANT** Bioconductor has many **dependencies** - other packages which is relies on. When you install Bioconductor packages you may need to update these packages. If something seems to not be working during this process, restart R and begin the Bioconductor installation process until things seem to work.

Below I discuss the series of prompts I had to deal with while re-installing Biostrings while editing this chapter.

### 10.3.1   Updating other packages when downloading a package

When I re-installed `Biostrings` while writing this I was given a HUGE blog of red test that contained this:

```
'getOption("repos")' replaces Bioconductor standard repositories, see '?repositories' :
details

replacement repositories:
    CRAN: https://cran.rstudio.com/

Bioconductor version 3.11 (BiocManager 1.30.16), R 4.0.5 (2021-03-31)
Old packages: 'ade4', 'ape', 'aster', 'bayestestR', 'bio3d', 'bitops', 'blogdown',
  'bookdown', 'brio', 'broom', 'broom.mixed', 'broomExtra', 'bslib', 'cachem', 'callr'
  'car', 'circlize', 'class', 'cli', 'cluster', 'colorspace', 'corrplot', 'cpp11', 'cu
  'devtools', 'DHARMa', 'doBy', 'dplyr', 'DT', 'e1071', 'ellipsis', 'emmeans', 'emojif
  'extRemes', 'fansi', 'flextable', 'forecast', 'formatR', 'gap', 'gargle', 'gert', 'G
  'ggfortify', 'ggplot2', 'ggsignif', 'ggVennDiagram', 'gh', 'glmmTMB', 'googledrive',
  'gtools', 'haven', 'highr', 'hms', 'htmlTable', 'httpuv', 'huxtable', 'jquerylib',
  'KernSmooth', 'knitr', 'later', 'lattice', 'lme4', 'magick', 'manipulateWidget', 'MA
```

```
'Matrix', 'matrixcalc', 'matrixStats', 'mgcv', 'mime', 'multcomp', 'mvtnorm', 'nnet',
'openssl', 'openxlsx', 'parameters', 'pBrackets', 'pdftools', 'phangorn', 'phytools',
'pillar', 'plotly', 'processx', 'proxy', 'qgam', 'quantreg', 'ragg', 'Rcpp',
'RcppArmadillo', 'remotes', 'rgl', 'rio', 'rJava', 'rlang', 'rmarkdown', 'robustbase',
'rsconnect', 'rversions', 'sandwich', 'sass', 'segmented', 'seqinr', 'seqmagick', 'servr',
'sf', 'shape', 'spatial', 'statmod', 'stringi', 'systemfonts', 'testthat', 'textshaping',
'tibble', 'tidyselect', 'tidytree', 'tinytex', 'tufte', 'UniprotR', 'units', 'vctrs',
'viridis', 'viridisLite', 'withr', 'xfun', 'zip'
```

Hidden at the bottom was a prompt: "Update all/some/none? [a/s/n]:"

Its a little vague, but what it wants me to do is type in a, s or n and press enter to tell it what to do. I almost always chose "a", though this may take a while to update everything.

### 10.3.2 Packages "from source"

You are likely to get lots of random-looking feedback from R when doing Bioconductor-related installations. Look carefully for any prompts as the very last line. While updating `Biostrings` I was told: "*There are binary versions available but the source versions are later:*" and given a table of packages. I was then asked "*Do you want to install from sources the packages which need compilation? (Yes/no/cancel)*"

I almost always chose "no".

### 10.3.3 More on angry red text

After the prompt about packages from source, R proceeded to download a lot of updates to packages, which took a few minutes. Lots of red text scrolled by, but this is normal.

**Console**    **Terminal** ✕    **R Markdown** ✕    **Jobs** ✕

~/google_backup_sync_nlb24/lbrb/ ⇗

```
trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/
Content type 'application/x-gzip' length 198800 bytes (194 KB
==================================================
downloaded 194 KB

trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/t
Content type 'application/x-gzip' length 220977 bytes (215 KB
==================================================
downloaded 215 KB

trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/t
Content type 'application/x-gzip' length 121221 bytes (118 KB
==================================================
downloaded 118 KB

trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/t
Content type 'application/x-gzip' length 269000 bytes (262 KB
==================================================
downloaded 262 KB

trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/U
Content type 'application/x-gzip' length 236776 bytes (231 KB
==================================================
downloaded 231 KB

trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/u
Content type 'application/x-gzip' length 1260870 bytes (1.2 M
===========================
```
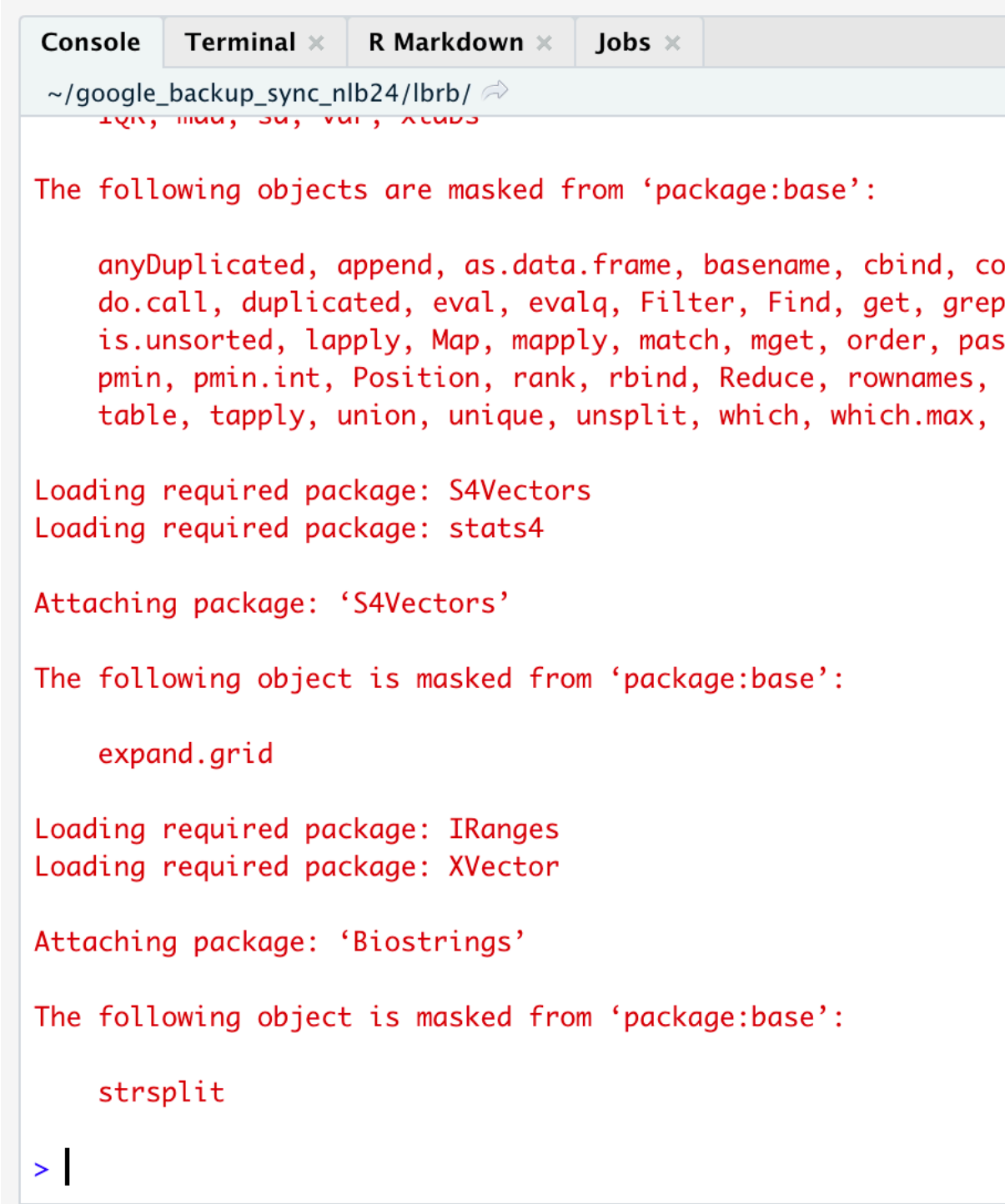
## 10.4 Actually loading a package

Again, to actually load the `Biostrings` package into your active R sessions requires the `libary()` command:

```
library(Biostrings)
```

As you might expect, there's more red text scrolling up my screen!

**Console**    Terminal ×    R Markdown ×    Jobs ×

~/google_backup_sync_nlb24/lbrb/ ⇗

```
IQR, mad, sd, var, xtabs

The following objects are masked from 'package:base':

    anyDuplicated, append, as.data.frame, basename, cbind, co
    do.call, duplicated, eval, evalq, Filter, Find, get, grep
    is.unsorted, lapply, Map, mapply, match, mget, order, pas
    pmin, pmin.int, Position, rank, rbind, Reduce, rownames,
    table, tapply, union, unique, unsplit, which, which.max,

Loading required package: S4Vectors
Loading required package: stats4

Attaching package: 'S4Vectors'

The following object is masked from 'package:base':

    expand.grid

Loading required package: IRanges
Loading required package: XVector

Attaching package: 'Biostrings'

The following object is masked from 'package:base':

    strsplit

>
```

I can tell that is actually worked because at the end of all the red stuff is the R prompt of ">" and my cursor.

# Chapter 11

# A Brief introduction to R

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

This chapter provides a brief introduction to R. At the end of are links to additional resources for getting started with R.

## 11.1 Vocabulary

- scalar
- vector
- list
- class
- numeric
- character
- assignment
- elements of an object
- indices
- attributes of an object
- argument of a function

## 11.2 R functions

- <-
- [ ]
- $
- table()
- function

- c()
- log10()
- help(), ?
- help.search()
- RSiteSearch()
- mean()
- return()
- q()

## 11.3   Interacting with R

You will type *R* commands into the RStudio **console** in order to carry out analyses in *R*. In the RStudio console you will see the R prompt starting with the symbol ">". ">" will always be there at the beginning of each new command - don't try to delete it! Moreover, you never need to type it.



We type the **commands** needed for a particular task after this prompt. The command is carried out by *R* after you hit the Return key.

Once you have started R, you can start typing commands into the RStudio console, and the results will be calculated immediately, for example:

```
2*3
```

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

Note that prior to the output of "6" it shows "[1]".

Now subtraction:

```
10-3
```

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

Again, prior to the output of "7" it shows "[1]".

*R* can act like a basic calculator that you type commands in to. You can also use it like a more advanced scientific calculator and create **variables** that store information. All variables created by R are called **objects**. In R, we assign values to variables using an arrow-looking function **<-** the **assignment operator**. For example, we can **assign** the value 2*3 to the variable x using the command:

```
x <- 2*3
```

To view the contents of any R object, just type its name, press enter, and the contents of that R object will be displayed:

```
x
```

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

## 11.4 Variables in R

There are several different types of objects in R with fancy math names, including **scalars**, **vectors**, **matrices** (singular: **matrix),** arrays**,** dataframes**,** tables**, and** lists**. The** scalar** variable x above is one example of an R object. While a scalar variable such as x has just one element, a **vector** consists of several elements. The elements in a vector are all of the same **type** (e.g.. numbers or alphabetic characters), while **lists** may include elements such as characters as well as numeric quantities. Vectors and dataframes are the most common variables you'll use. You'll also encounter matrices often, and lists are ubiquitous in R but beginning users often don't encounter them because they remain behind the scenes.

### 11.4.1 Vectors

To create a vector, we can use the `c()` (combine) function. For example, to create a vector called `myvector` that has elements with values 8, 6, 9, 10, and 5, we type:

```
myvector <- c(8, 6, 9, 10, 5) # note: commas between each number!
```

To see the contents of the variable `myvector`, we can just type its name and press enter:

```
myvector
```

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

### 11.4.2 Vector indexing

The `[1]` is the **index** of the first **element** in the vector. We can **extract** any element of the vector by typing the vector name with the index of that element given in **square brackets [...]**.

For example, to get the value of the 4th element in the vector `myvector`, we type:

```
myvector[4]
```

```
## [1] 10
```

### 11.4.3 Character vectors

Vectors can contain letters, such as those designating nucleic acids

```r
my.seq <- c("A","T","C","G")
```

They can also contain multi-letter **strings**:

```r
my.oligos <- c("ATCGC","TTTCGC","CCCGCG","GGGCGC")
```

### 11.4.4  Lists

**NOTE**: *below is a discussion of lists in R. This is excellent information, but not necessary if this is your very very first time using R.*

In contrast to a vector, a **list** can contain elements of different types, for example, both numbers and letters.  A list can even include other variables such as a vector. The `list()` function is used to create a list.  For example, we could create a list `mylist` by typing:

```r
mylist <- list(name="Charles Darwin",
               wife="Emma Darwin",
               myvector)
```

We can then print out the contents of the list mylist by typing its name:

```r
mylist
```

```
## $name
## [1] "Charles Darwin"
##
## $wife
## [1] "Emma Darwin"
##
## [[3]]
## [1]  8  6  9 10  5
```

The **elements** in a list are numbered, and can be referred to using **indices**.  We can extract an element of a list by typing the list name with the index of the element given in double **square brackets** (in contrast to a vector, where we only use single square brackets).

We can extract the second element from `mylist` by typing:

```r
mylist[[2]]  # note the double square brackets [[...]]
```

```
## [1] "Emma Darwin"
```

As a baby step towards our next task, we can wrap index values as in the `c()` command like this:

```r
mylist[[c(2)]]  # note the double square brackets [[...]]
```

```
## [1] "Emma Darwin"
```

The number `2` and `c(2)` mean the same thing.

Now, we can extract the second AND third elements from `mylist`. First, we put the indices 2 and 3 into a vector `c(2,3)`, then wrap that vector in double square brackets: `[c(2,3)]`. All together it looks like this.

```
mylist[c(2,3)] # note the double brackets
```

```
## $wife
## [1] "Emma Darwin"
##
## [[2]]
## [1]  8  6  9 10  5
```

Elements of lists may also be named, resulting in a **named lists**. The elements may then be referred to by giving the list name, followed by "$", followed by the element name. For example, mylist$name is the same as mylist[[1]] and mylist$wife is the same as mylist[[2]]:

```
mylist$wife
```

```
## [1] "Emma Darwin"
```

We can find out the names of the named elements in a list by using the `attributes()` function, for example:

```
attributes(mylist)
```

```
## $names
## [1] "name" "wife" ""
```

When you use the `attributes()` function to find the named elements of a list variable, the named elements are always listed under a heading "$names". Therefore, we see that the named elements of the list variable `mylist` are called "name" and "wife", and we can retrieve their values by typing mylist$name and mylist$wife, respectively.

### 11.4.5   Tables

Another type of object that you will encounter in R is a **table**. The `table()` function allows you to total up or tabulate the number of times a value occurs within a vector. Tables are typically used on vectors containing **character data**, such as letters, words, or names, but can work on numeric data data.

#### 11.4.5.1   Tables - The basics

If we made a vector variable "nucleotides" containing the of a DNA molecule, we can use the `table()` function to produce a **table variable** that contains the number of bases with each possible nucleotides:

```r
bases <- c("A", "T", "A", "A", "T", "C", "G", "C", "G")
```

Now make the table

```r
table(bases)
```

```
## bases
## A C G T
## 3 2 2 2
```

We can store the table variable produced by the function `table()`, and call the stored table "bases.table", by typing:

```r
bases.table <- table(bases)
```

Tables also work on vectors containing numbers. First, a vector of numbers.

```r
numeric.vecter <- c(1,1,1,1,3,4,4,4,4)
```

Second, a table, showing how many times each number occurs.

```r
table(numeric.vecter)
```

```
## numeric.vecter
## 1 3 4
## 4 1 4
```

### 11.4.5.2  Tables - further details

To access elements in a table variable, you need to use double square brackets, just like accessing elements in a list. For example, to access the fourth element in the table bases.table (the number of Ts in the sequence), we type:

```r
bases.table[[4]]   # double brackets!
```

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

Alternatively, you can use the name of the fourth element in the table ("John") to find the value of that table element:

```r
bases.table[["T"]]
```

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

## 11.5  Arguments

Functions in R usually require **arguments**, which are input variables (i.e.. objects) that are **passed** to them, which they then carry out some operation on. For example, the `log10()` function is passed a number, and it then calculates the log to the base 10 of that number:

```
log10(100)
```

## [1] 2

There's a more generic function, `log()`, where we pass it not only a number to take the log of, but also the specific **base** of the logarithm. To take the log base 10 with the `log()` function we do this

```
log(100, base = 10)
```

## [1] 2

We can also take logs with other bases, such as 2:

```
log(100, base = 2)
```

## [1] 6.643856

## 11.6   Help files with `help()` and ?

In *R*, you can get help about a particular function by using the `help()` function. For example, if you want help about the `log10()` function, you can type:

```
help("log10")
```

When you use the `help()` function, a box or web pag will show up in one of the panes of RStudio with information about the function that you asked for help with. You can also use the ? next to the function like this

```
?log10
```

Help files are a mixed bag in R, and it can take some getting used to them. An excellent overview of this is Kieran Healy's "How to read an R help page."

## 11.7   Searching for functions with `help.search()` and `RSiteSearch()`

If you are not sure of the name of a function, but think you know part of its name, you can search for the function name using the `help.search()` and `RSiteSearch()` functions. The `help.search()` function searches to see if you already have a function installed (from one of the R packages that you have installed) that may be related to some topic you're interested in. `RSiteSearch()` searches *all* R functions (including those in packages that you haven't yet installed) for functions related to the topic you are interested in.

For example, if you want to know if there is a function to calculate the standard deviation (SD) of a set of numbers, you can search for the names of all installed functions containing the word "deviation" in their description by typing:

```
help.search("deviation")
```

Among the functions that were found, is the function **sd()** in the **stats** package (an R package that comes with the base R installation), which is used for calculating the standard deviation.

Now, instead of searching just the packages we've have on our computer let's search all R packages on CRAN. Let's look for things related to DNA. Note that **RSiteSearch()** doesn't provide output within RStudio, but rather opens up your web browser for you to display the results.

```
RSiteSearch("DNA")
```

The results of the **RSiteSearch()** function will be hits to descriptions of R functions, as well as to R mailing list discussions of those functions.

## 11.8  More on functions

We can perform computations with R using objects such as scalars and vectors. For example, to calculate the average of the values in the vector **myvector** (i.e.. the average of 8, 6, 9, 10 and 5), we can use the **mean()** function:

```
mean(myvector) # note: no " "
```

```
## [1] 7.6
```

We have been using built-in R functions such as mean(), length(), print(), plot(), etc.

### 11.8.1  Writing your own functions

**NOTE**: *Writing your own functions is an advanced skills. New users can skip this section.

We can also create our own functions in R to do calculations that you want to carry out very often on different input data sets. For example, we can create a function to calculate the value of 20 plus square of some input number:

```
myfunction <- function(x) { return(20 + (x*x)) }
```

This function will calculate the square of a number (x), and then add 20 to that value. The **return()** statement returns the calculated value. Once you have typed in this function, the function is then available for use. For example, we can use the function for different input numbers (e.g.. 10, 25):

```
myfunction(10)
```

```
## [1] 120
```

## 11.9   Quiting R

To quit R either close the program, or type:

```
q()
```

## 11.10   Links and Further Reading

Some links are included here for further reading.

For a more in-depth introduction to R, a good online tutorial is available on the "Kickstarting R" website, cran.r-project.org/doc/contrib/Lemon-kickstart.

There is another nice (slightly more in-depth) tutorial to R available on the "Introduction to R" website, cran.r-project.org/doc/manuals/R-intro.html.

Chapter 3 of Danielle Navarro's book is an excellent intro to the basics of R.

# Chapter 12

# DNA descriptive statics - Part 1

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 12.1  Preface

This is a modification of "DNA Sequence Statistics (1)" from Avril Coghlan's *A little book of R for bioinformatics..* The text and code were originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 12.2  Writing TODO:

- Add biology introduction
- Work on flow
- organize intial sections (intro, vocab, preliminaries)

## 12.3  Introduction

## 12.4  Vocabulary

- GC content
- DNA words
- scatterplots, histograms, piecharts, and boxplots

## 12.5   Functions

- `seqinr::GC()`
- `seqinr::count()`

## 12.6   Preliminaries

```
library(compbio4all)
library(seqinr)
```

## 12.7   Converting DNA from FASTA format

In a previous exercise we downloaded and examined DNA sequence in the
FASTA format.  The sequence we worked with is also stored as a data file
within the `compbio4all` pa package and can be brought into memory using the
`data()` command.

```
data("dengueseq_fasta")
```

We can look at this data object with the `str()` command

```
str(dengueseq_fasta)
```

```
##  chr ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACA(
```

This isn't in a format we can work with directly so we'll use the function
fasta_cleaner() to set it up.

```
header. <- ">NC_001477.1 Dengue virus 1, complete genome"
dengueseq_vector <- compbio4all::fasta_cleaner(dengueseq_fasta)
```

Now check it out.

```
str(dengueseq_vector)
```

```
##  chr [1:10735] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T" "C" "T" "A" "C" "G" ...
```

What we have here is each base of the sequence in a seperate slot of our vector.

The first four bases are "AGTT"

We can see the first one like this

```
dengueseq_vector[1]
```

```
## [1] "A"
```

The second one like this

```
dengueseq_vector[2]
```

```
## [1] "G"
```

The first and second like this

```
dengueseq_vector[1:2]
```

```
## [1] "A" "G"
```

and all four like this

```
dengueseq_vector[1:4]
```

```
## [1] "A" "G" "T" "T"
```

## 12.8   Length of a DNA sequence

Once you have retrieved a DNA sequence, we can obtain some simple statistics to describe that sequence, such as the sequence's total length in nucleotides. In the above example, we retrieved the DEN-1 Dengue virus genome sequence, and stored it in the vector variable dengueseq_vector To obtain the length of the genome sequence, we would use the length() function, typing:

```
length(dengueseq_vector)
```

```
## [1] 10735
```

The length() function will give you back the length of the sequence stored in variable dengueseq_vector, in nucleotides. The length() function actually gives the number of **elements** (slots) in the input vector that you passed to it, which in this case in the number of elements in the vector dengueseq_vector. Since each element of the vector dengueseq_vector contains one nucleotide of the DEN-1 Dengue virus sequence, the result for the DEN-1 Dengue virus genome tells us the length of its genome sequence (ie. 10735 nucleotides long).

### 12.8.1   Base composition of a DNA sequence

An obvious first analysis of any DNA sequence is to count the number of occurrences of the four different nucleotides ("A", "C", "G", and "T") in the sequence. This can be done using the the table() function. For example, to find the number of As, Cs, Gs, and Ts in the DEN-1 Dengue virus sequence (which you have put into vector variable dengueseq_vector, using the commands above), you would type:

```
table(dengueseq_vector)
```

```
## dengueseq_vector
##    A    C    G    T
```

```
## 3426 2240 2770 2299
```

This means that the DEN-1 Dengue virus genome sequence has 3426 As occurring throughout the genome, 2240 Cs, and so forth.

### 12.8.2   GC Content of DNA

One of the most fundamental properties of a genome sequence is its **GC content**, the fraction of the sequence that consists of Gs and Cs, ie. the %(G+C).

The GC content can be calculated as the percentage of the bases in the genome that are Gs or Cs. That is, GC content = (number of Gs + number of Cs)*100/(genome length). For example, if the genome is 100 bp, and 20 bases are Gs and 21 bases are Cs, then the GC content is (20 + 21)*100/100 = 41%.*

You can easily calculate the GC content based on the number of As, Gs, Cs, and Ts in the genome sequence. For example, for the DEN-1 Dengue virus genome sequence, we know from using the table() function above that the genome contains 3426 As, 2240 Cs, 2770 Gs and 2299 Ts. Therefore, we can calculate the GC content using the command:

```
(2240+2770)*100/(3426+2240+2770+2299)
```

```
## [1] 46.66977
```

Alternatively, if you are feeling lazy, you can use the GC() function in the SeqinR package, which gives the fraction of bases in the sequence that are Gs or Cs.

```
seqinr::GC(dengueseq_vector)
```

```
## [1] 0.4666977
```

The result above means that the fraction of bases in the DEN-1 Dengue virus genome that are Gs or Cs is 0.4666977. To convert the fraction to a percentage, we have to multiply by 100, so the GC content as a percentage is 46.66977%.

### 12.8.3   DNA words

As well as the frequency of each of the individual nucleotides ("A", "G", "T", "C") in a DNA sequence, it is also interesting to know the frequency of longer **DNA words**, also referred to as **genomic words**. The individual nucleotides are DNA words that are 1 nucleotide long, but we may also want to find out the frequency of DNA words that are 2 nucleotides long (ie. "AA", "AG", "AC", "AT", "CA", "CG", "CC", "CT", "GA", "GG", "GC", "GT", "TA", "TG", "TC", and "TT"), 3 nucleotides long (eg. "AAA", "AAT", "ACG", etc.), 4 nucleotides long, etc.

To find the number of occurrences of DNA words of a particular length, we can use the count() function from the R SeqinR package.

The count() function only works with lower-case letters, so first we have to use the tolower() function to convert our upper class genome to lower case

```
dengueseq_vector <-tolower(dengueseq_vector)
```

Now we can look for words. For example, to find the number of occurrences of DNA words that are 1 nucleotide long in the sequence dengueseq_vector, we type:

```
seqinr::count(dengueseq_vector, 1)
```

```
##
##    a    c    g    t
## 3426 2240 2770 2299
```

As expected, this gives us the number of occurrences of the individual nucleotides. To find the number of occurrences of DNA words that are 2 nucleotides long, we type:

```
seqinr::count(dengueseq_vector, 2)
```

```
##
##   aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
## 1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
```

Note that by default the count() function includes all overlapping DNA words in a sequence. Therefore, for example, the sequence "ATG" is considered to contain two words that are two nucleotides long: "AT" and "TG".

If you type help('count'), you will see that the result (output) of the function count() is a table object. This means that you can use double square brackets to extract the values of elements from the table. For example, to extract the value of the third element (the number of Gs in the DEN-1 Dengue virus sequence), you can type:

```
denguetable_2 <- seqinr::count(dengueseq_vector,2)
denguetable_2[[3]]
```

```
## [1] 890
```

The command above extracts the third element of the table produced by count(dengueseq_vector,1), which we have stored in the table variable denguetable.

Alternatively, you can find the value of the element of the table in column "g" by typing:

```
denguetable_2[["aa"]]
```

```
## [1] 1108
```

Once you have table you can make a basic plot

```
barplot(denguetable_2)
```
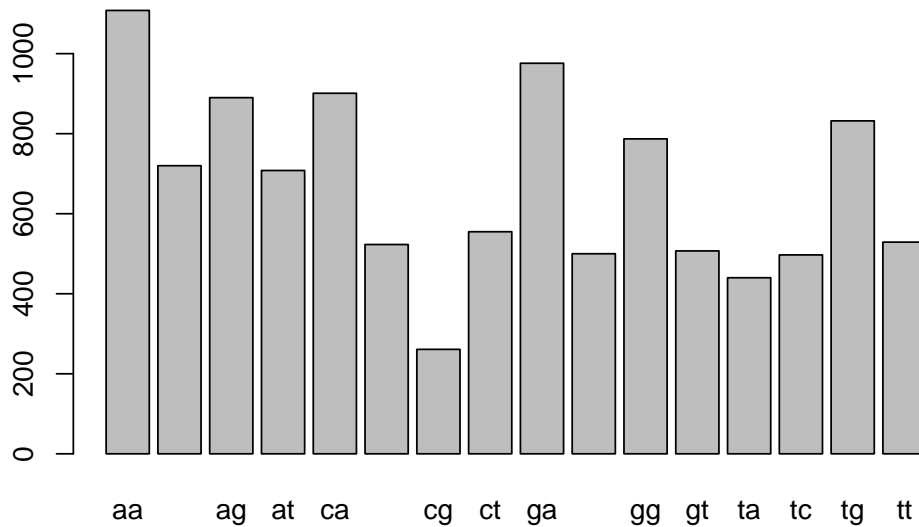


We can sort by the number of words using the sort() command

```
sort(denguetable_2)
```

```
##
##   cg   ta   tc   gc   gt   cc   tt   ct   at   ac   gg   tg   ag   ca   ga   aa
##  261  440  497  500  507  523  529  555  708  720  787  832  890  901  976 1108
```

Let's save over the original object

```
denguetable_2 <- sort(denguetable_2)
```

```
barplot(denguetable_2)
```
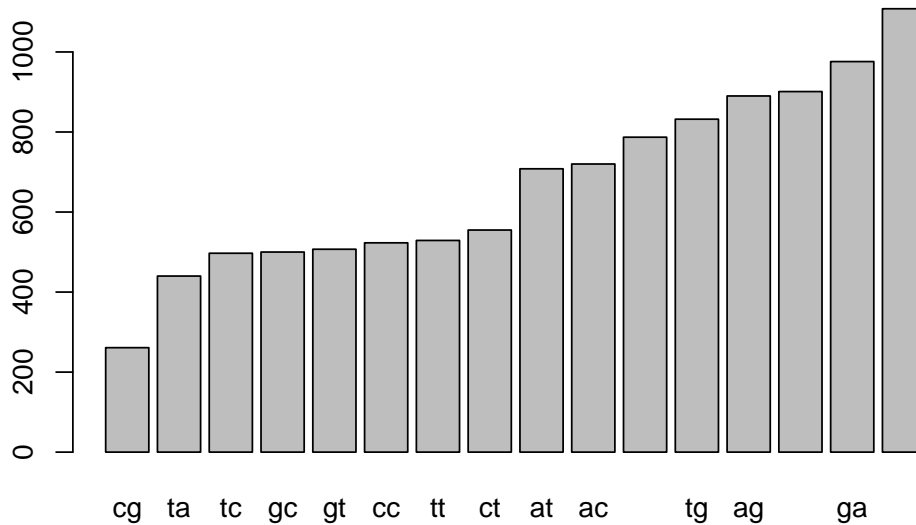
R will automatically try to optimize the appearance of the labels on the graph so you may not see all of them; no worries.

R can also make pie charts. Piecharts only really work when there are a few items being plots, like the four bases.

```
denguetable_1 <- seqinr::count(dengueseq_vector,1)
```

Make a piechart with pie()

```
pie(denguetable_1)
```
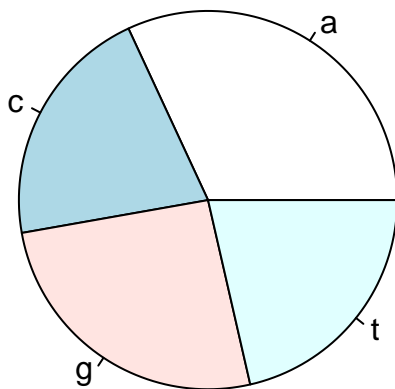


### 12.8.4 Summary

In this practical, have learned to use the following R functions:

length() for finding the length of a vector or list table() for printing out a table of the number of occurrences of each type of item in a vector or list. These

functions belong to the standard installation of R.

You have also learnt the following R functions that belong to the SeqinR package:

GC() for calculating the GC content for a DNA sequence count() for calculating the number of occurrences of DNA words of a particular length in a DNA sequence

## 12.9    Acknowledgements

This is a modification of "DNA Sequence Statistics (1)" from Avril Coghlan's *A little book of R for bioinformatics..* Almost all of text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

In "A little book..." Coghlan noted: "Many of the ideas for the examples and exercises for this chapter were inspired by the Matlab case studies on Haemophilus influenzae (www.computational-genomics.net/case_studies/haemophilus_demo.html) and Bacteriophage lambda (http://www.computational-genomics.net/case_studies/lambdaphage_demo.html) from the website that accompanies the book Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/)."

### 12.9.1    License

The content in this book is licensed under a Creative Commons Attribution 3.0 License.

https://creativecommons.org/licenses/by/3.0/us/

### 12.9.2    Exercises

Answer the following questions, using the R package. For each question, please record your answer, and what you typed into R to get this answer.

Model answers to the exercises are given in Answers to the exercises on DNA Sequence Statistics (1).

1. What are the last twenty nucleotides of the Dengue virus genome sequence?
2. What is the length in nucleotides of the genome sequence for the bacterium Mycobacterium leprae strain TN (accession NC_002677)? Note: Mycobacterium leprae is a bacterium that is responsible for causing leprosy, which is classified by the WHO as a neglected tropical disease. As the genome sequence is a DNA sequence, if you are retrieving its sequence via the NCBI website, you will need to look for it in the NCBI Nucleotide database.

3. How many of each of the four nucleotides A, C, T and G, and any other symbols, are there in the Mycobacterium leprae TN genome sequence? Note: other symbols apart from the four nucleotides A/C/T/G may appear in a sequence. They correspond to positions in the sequence that are are not clearly one base or another and they are due, for example, to sequencing uncertainties. or example, the symbol 'N' means 'aNy base', while 'R' means 'A or G' (puRine). There is a table of symbols at www.bioinformatics.org/sms/iupac.html.

4. What is the GC content of the *Mycobacterium leprae TN* genome sequence, when (i) all non-A/C/T/G nucleotides are included, (ii) non-A/C/T/G nucleotides are discarded? Hint: look at the help page for the GC() function to find out how it deals with non-A/C/T/G nucleotides.

5. How many of each of the four nucleotides A, C, T and G are there in the complement of the Mycobacterium leprae TN genome sequence? *Hint*: you will first need to search for a function to calculate the complement of a sequence. Once you have found out what function to use, remember to use the help() function to find out what are the arguments (inputs) and results (outputs) of that function. How does the function deal with symbols other than the four nucleotides A, C, T and G? Are the numbers of As, Cs, Ts, and Gs in the complementary sequence what you would expect?

6. How many occurrences of the DNA words CC, CG and GC occur in the Mycobacterium leprae TN genome sequence?

7. How many occurrences of the DNA words CC, CG and GC occur in the (i) first 1000 and (ii) last 1000 nucleotides of the Mycobacterium leprae TN genome sequence? 1.How can you check that the subsequence that you have looked at is 1000 nucleotides long?

# Chapter 13

# Programming in R: for loops

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 13.1   Preface

This is a modification of "DNA Sequence Statistics (1)" from Avril Coghlan's *A little book of R for bioinformatics.*. The text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 13.2   Vocab

- for loop
- curly brackets

## 13.3   Functions

- `for()`
- `print()`

## 13.4   Basic for loops in in R

In *R*, just as in programming languages such as **Python**, it is possible to write a **for loop** to carry out the same command several times. For example, say we

have a pressing need to calculate the square the square of each number between 1 and 4. We could write for lines of code like this to do it:

```
1^2
```

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

```
2^2
```

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

```
3^2
```

```
## [1] 9
```

```
4^2
```

```
## [1] 16
```

If we know how to write a for loop, we could do the same think like this:

```
for (i in 1:4) {
  print (i*i)
  }
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

In the for loop above, the variable `i` is a counter or **index** for the number of cycles through the loop. In the first cycle through the loop, the value of `i` is 1, and so `i * i` = 1 is printed out. In the second cycle through the loop, the value of `i` is 2, and so `i * i` = 4 is printed out. In the third cycle through the loop, the value of `i` is 3, and so `i * i` = 9 is printed out. The loop continues until the value of `i` is 4.

Note that the commands that are to be carried out at each cycle of the for loop must be enclosed within **curly brackets** ("{" and "}").

You may be thinking "*ok, so it took four lines of code to do $1^2$ through $4^2$ each on their own, and three lines to do it wit the loop; what's the big deal?*". What if you need to do 1 through 100 squared for some reason? Now the for loop is a lot less work.

You can also give a for loop a vector of numbers containing the values that you want the counter `i` to take in subsequent cycles. For example, you can make a vector containing the numbers 1, 2, 3, and 4, and write a for loop to print out the square of each number in vector avector:

```
## [1] 1
## [1] 4
## [1] 9
```

```
## [1] 16
```

The results should be the same as before.

## 13.5 Challenge: complicated vectors of values

Here's a more complex example. If you don't understand it don't worry, its not something you'd probably do in practice.

Challenge: How can we use a for loop to print out the square of every second number between, say, 1 and 10? The answer is to use the seq() function with "by = 2" to tell the for loop to take every second number between 1 and 10:

```
for (i in seq(1, 10, by = 2)) {
  print (i*i)
  }
```

```
## [1] 1
## [1] 9
## [1] 25
## [1] 49
## [1] 81
```

In the first cycle of this loop, the value of i is 1, and so i * i = 1 is printed out. In the second cycle through the loop, the value of i is 3, and so i * i = 9 is printed out. The loop continues until the value of i is 9. In the fifth cycle through the loop, the value of i is 9, and so i * i = 81 is printed out.

# Chapter 14

# Mini tutorial: Vectors in R

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 14.1   Preface

This is a modification of part of"DNA Sequence Statistics (2)" from Avril Coghlan's *A little book of R for bioinformatics.*. Most of text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 14.2   Vocab

- base R
- scalar, vector, matrix
- regular expressions

# Chapter 15

# Functions

- `seq()`
- `is()`, `is.vector()`, `is.matrix()`
- `gsub()`

## 15.1   Vectors in R

**Variables** in R include **scalars**, **vectors**, and **lists**. **Functions** in R carry out operations on variables, for example, using the `log10()` function to calculate the log to the base 10 of a scalar variable `x`, or using the `mean()` function to calculate the average of the values in a vector variable `myvector`. For example, we can use `log10()` on a scalar object like this:

```
# store value in object
x <- 100

# take log base 10 of object
log10(x)
```

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

Note that while mathematically x is a single number, or a scalar, R considers it to be a vector:

```
is.vector(x)
```

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

There are many "is" commands. What is returned when you run `is.matrix()` on a vector?

```
is.matrix(x)
```

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

Mathematically this is a bit odd, since often a vector is defined as a one-dimensional matrix, e.g., a single column or single row of a matrix. But in *R* land, a vector is a vector, and matrix is a matrix, and there are no explicit scalars.

## 15.2  Math on vectors

Vectors can serve as the input for mathematical operations. When this is done *R* does the mathematical operation separately on each element of the vector. This is a unique feature of *R* that can be hard to get used to even for people with previous programming experience.

Let's make a vector of numbers:

```
myvector <- c(30,16,303,99,11,111)
```

What happens when we multiply `myvector` by 10?

```
myvector*10
```

```
## [1]  300  160 3030  990  110 1110
```

R has taken each of the 6 values, 30 through 111, of `myvector` and multiplied each one by 10, giving us 6 results. That is, what R did was

```
## 30*10     # first value of myvector
## 16*10     # second value of myvector
## 303*10    # ....
## 99*10
## 111*10    # last value of myvector
```

The normal order of operations rules apply to vectors as they do to operations we're more used to. So multiplying `myvector` by 10 is the same whether you put he 10 before or after vector. That is `myvector\*10` is the same as `10\*myvector`.

```
myvector*10
```

```
## [1]  300  160 3030  990  110 1110
```
```
10*myvector
```

```
## [1]  300  160 3030  990  110 1110
```

What happen when you subtract 30 from myvector? Write the code below.

```
myvector-30
```

```
## [1]   0 -14 273  69 -19  81
```

So, what R did was

```
## 30-30    # first value of myvector
## 16-30    # second value of myvector
## 303-30   # ....
## 99-30
## 111-30   # last value of myvector
```

You can also square a vector

```
myvector^2
```

```
## [1]   900   256 91809  9801   121 12321
```

Which is the same as

```
## 30^2    # first value of myvector
## 16^2    # second value of myvector
## 303^2   # ....
## 99^2
## 111^2   # last value of myvector
```

Also you can take the square root of a vector...

```
sqrt(myvector)
```

```
## [1]  5.477226  4.000000 17.406895  9.949874  3.316625 10.535654
```

...and take the log of a vector...

```
log(myvector)
```

```
## [1] 3.401197 2.772589 5.713733 4.595120 2.397895 4.709530
```

...and just about any other mathematical operation. Here we are working on a separate vector object; all of these rules apply to a column in a matrix or a dataframe. This attribution of R is called **vectorization**.

## 15.3   Functions on vectors

We can use functions on vectors. Typically these use the vectors as an input and all the numbers are processed into an output. Call the `mean()` function on the vector we made called `myvector`.

```
mean(myvector)
```

```
## [1] 95
```

Note how we get a single value back - the mean of all the values in the vector. R saw that we had a vector of multiple and knew that the mean is a function that doesn't get applied to single number, but sets of numbers.

The function `sd()` calculates the standard deviation. Apply the `sd()` to myvector:

```
sd(myvector)
```

```
## [1] 110.5061
```

## 15.4   Operations with two vectors

You can also subtract one vector from another vector. This can be a little weird when you first see it. Make another vector with the numbers 5, 10, 15, 20, 25, 30. Call this myvector2:

```
myvector2 <- c(5, 10, 15, 20, 25, 30)
```

Now subtract myvector2 from myvector. What happens?

```
myvector-myvector2
```

```
## [1]   25    6 288   79 -14   81
```

## 15.5   Subsetting vectors

You can extract an **element** of a vector by typing the vector name with the index of that element given in **square brackets**. For example, to get the value of the 3rd element in the vector `myvector`, we type:

```
myvector[3]
```

```
## [1] 303
```

Extract the 4th element of the vector:

```
myvector[4]
```

```
## [1] 99
```

You can extract more than one element by using a vector in the brackets:

First, say I want to extract the 3rd and the 4th element. I can make a vector with 3 and 4 in it:

```
nums <- c(3,4)
```

Then put that vector in the brackets:

```
myvector[nums]
```

```
## [1] 303  99
```

We can also do it directly like this, skipping the vector-creation step:

```r
myvector[c(3,4)]
```

```
## [1] 303  99
```

In the chunk below extract the 1st and 2nd elements:

```r
myvector[c(1,2)]
```

```
## [1] 30 16
```

## 15.6 Sequences of numbers

Often we want a vector of numbers in **sequential order**. That is, a vector with the numbers 1, 2, 3, 4, ... or 5, 10, 15, 20, ... The easiest way to do this is using a colon

```r
1:10
```

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

Note that in R 1:10 is equivalent to c(1:10)

```r
c(1:10)
```

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

Usually to emphasize that a vector is being created I will use c(1:10)

We can do any number to any numbers

```r
c(20:30)
```

```
##  [1] 20 21 22 23 24 25 26 27 28 29 30
```

We can also do it in *reverse*. In the code below put 30 before 20:

```r
c(30:20)
```

```
##  [1] 30 29 28 27 26 25 24 23 22 21 20
```

A useful function in $R$ is the `seq()` function, which is an explicit function that can be used to create a vector containing a sequence of numbers that run from a particular number to another particular number.

```r
seq(1, 10)
```

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

Using `seq()` instead of a `:` can be useful for readability to make it explicit what is going on. More importantly, `seq` has an argument `by = ...` so you can make a sequence of number with any interval between For example, if we want to create the sequence of numbers from 1 to 10 in steps of 1 (i.e.. 1, 2, 3, 4, ... 10), we can type:

```
seq(1, 10,
    by = 1)
```

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

We can change the **step size** by altering the value of the `by` argument given to the function `seq()`. For example, if we want to create a sequence of numbers from 1-100 in steps of 20 (i.e.. 1, 21, 41, … 101), we can type:

```
seq(1, 101,
    by = 20)
```

```
## [1]   1  21  41  61  81 101
```

## 15.7   Vectors can hold numeric or character data

The vector we created above holds numeric data, as indicated by `class()`

```
class(myvector)
```

```
## [1] "numeric"
```

Vectors can also holder character data, like the genetic code:

```
# vector of character data
myvector <- c("A","T","G")

# how it looks
myvector
```

```
## [1] "A" "T" "G"
```

```
# what is "is"
class(myvector)
```

```
## [1] "character"
```

## 15.8   Regular expressions can modify character data

We can use **regular expressions** to modify character data.  For example, change the Ts to Us

```
myvector <- gsub("T", "U", myvector)
```

Now check it out

```
myvector
```

```
## [1] "A" "U" "G"
```

Regular expressions are a deep subject in computing. You can find some more information about them here.

# Chapter 16

# Plotting vectors in base R

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 16.1  Preface

This is a modification of part of"DNA Sequence Statistics (2)" from Avril Coghlan's *A little book of R for bioinformatics.*. Most of text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 16.2  Plotting numeric data

R allows the production of a variety of plots, including **scatterplots**, **histograms**, **piecharts**, and **boxplots**. Usually we make plots from dataframes with 2 or more columns, but we can also make them from two separate vectors. This flexibility is useful, but also can cause some confusion.

For example, if you have two equal-length vectors of numbers `numeric.vect1` and `numeric.vect1`, you can plot a scatterplot of the values in `myvector1` against the values in `myvector2` using the **base R** `plot()`function.

First, let's make up some data in put it in vectors:

```
numeric.vect1 <- c(10, 15, 22, 35, 43)
numeric.vect2 <- c(3, 3.2, 3.9, 4.1, 5.2)
```

Not plot with the base R `plot()` function:

```
plot(numeric.vect1, numeric.vect2)
```
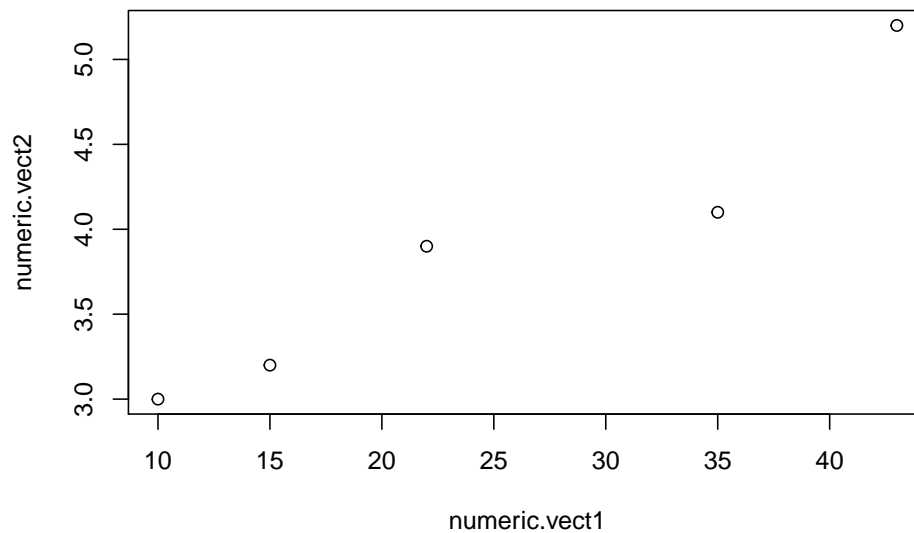
Note that there is a comma between the two vector names. When building plots from dataframes you usually see a tilde (~), but when you have two vectors you can use just a comma.

Also note the order of the vectors within the `plot()` command and which axes they appear on. The first vector is `numeric.vect1` and it appears on the horizontal x-axis.

If you want to label the axes on the plot, you can do this by giving the `plot()` function values for its optional arguments `xlab =` and `ylab =`:

```
plot(numeric.vect1,    # note again the comma, not a ~
     numeric.vect2,
     xlab="vector1",
     ylab="vector2")
```
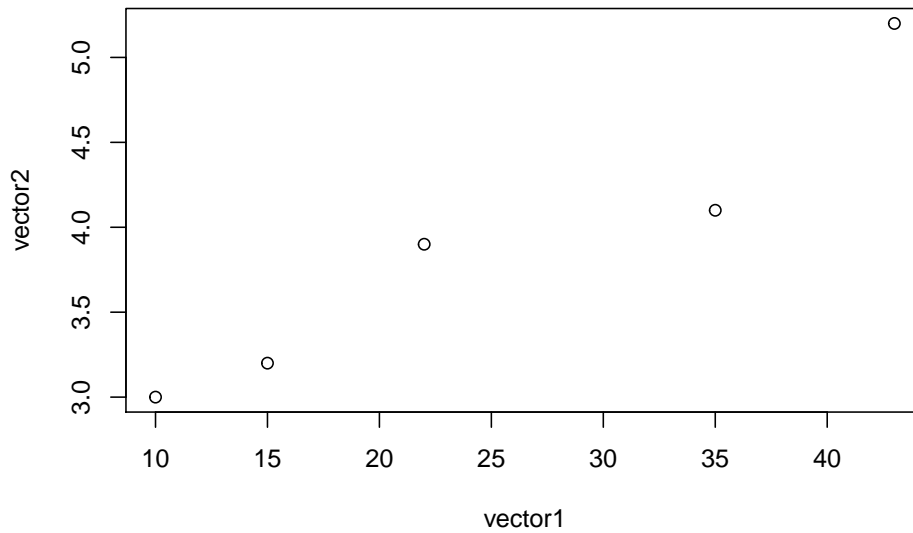
We can store character data in vectors so if we want we could do this to set up our labels:

```
mylabels <-  c("numeric.vect1","numeric.vect2")
```

Then use bracket notation to call the labels from the vector

```
plot(numeric.vect1,
     numeric.vect2,
     xlab=mylabels[1],
     ylab=mylabels[2])
```
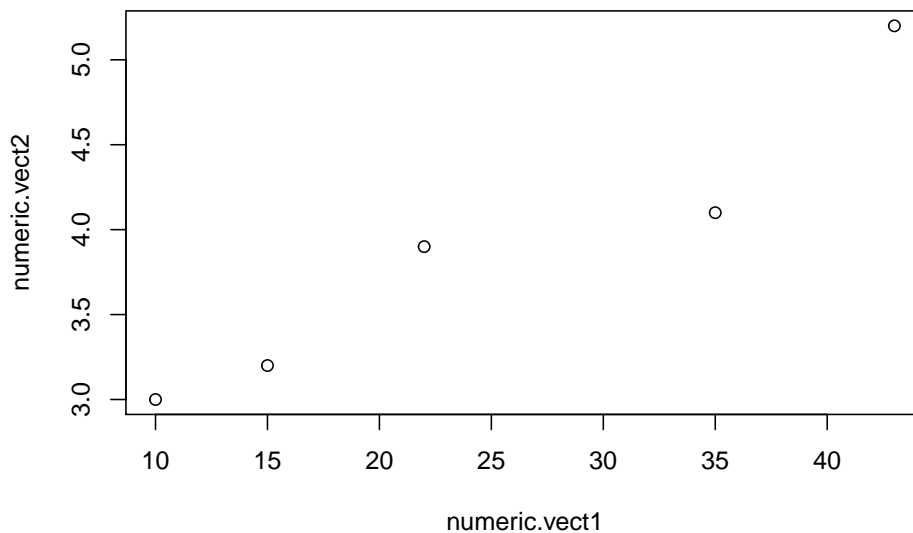


If we want we can use a tilde to make our plot like this:

```
plot(numeric.vect2 ~ numeric.vect1)
```
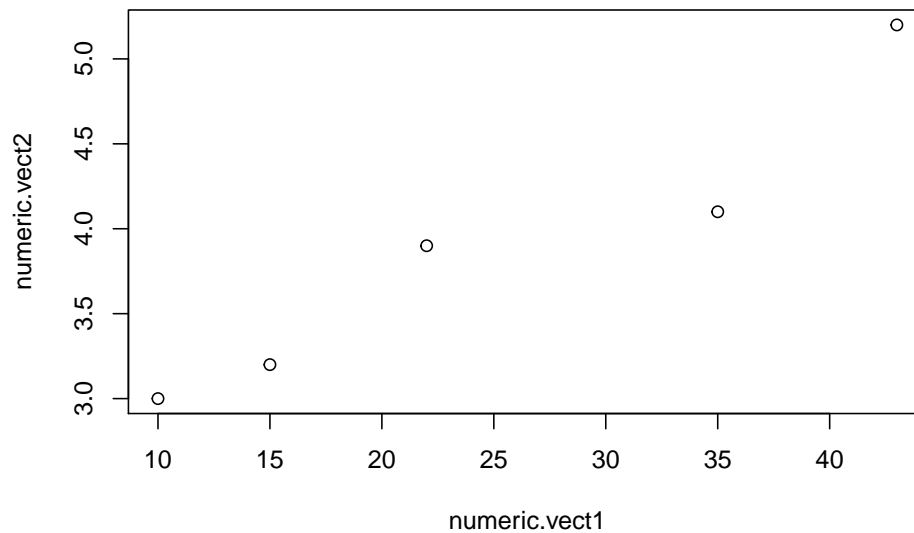


Note that now, `numeric.vect2` is on the left and `numeric.vect1` is on the right. This flexibility can be tricky to keep track of.

We can also combine these vectors into a dataframe and plot the data by referencing the data frame. First, we combine the two separate vectors into a dataframe using the `cbind()` command.

```
df <- cbind(numeric.vect1, numeric.vect2)
```

Then we plot it like this, referencing the dataframe `df` via the `data = ...` argument.

```
plot(numeric.vect2 ~ numeric.vect1, data = df)
```
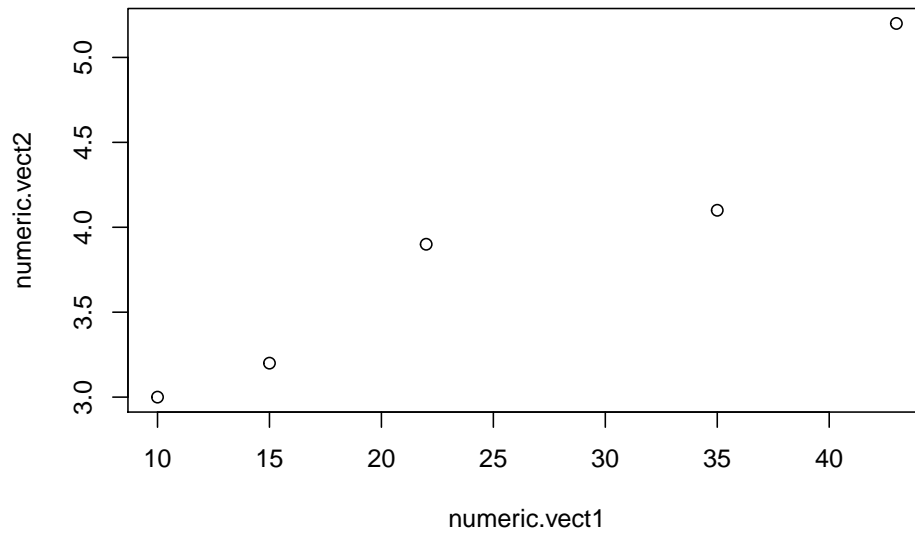
## 16.3 Other plotting packages

Base R has lots of plotting functions; additionally, people have written packages to implement new plotting capabilities. The package `ggplot2` is currently the most popular plotting package, and `ggpubr` is a package which makes `ggplot2` easier to use. For quick plots we'll use base R functions, and when we get to more important things we'll use ggplot2 and ggpubr.

# Chapter 17

# Programming in R: functions

**By**: Avril Coghlan

**Adapted, edited and expanded**: Nathan Brouwer (brouwern@gmail.com) under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 17.1 Preface

This is a modification of "DNA Sequence Statistics (1)" from Avril Coghlan's *A little book of R for bioinformatics.*. The text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 17.2 Vocab

- function
- curly brackets

## 17.3 Functions

- `function()`

## 17.4 Functions in R

We have been using **built-in functions** such as mean(), length(), print(), plot(), etc. We can also create our own functions in R to do calculations that you want

to carry out very often on different input data sets. For example, we can create a function to calculate the value of 20 plus the square of some input number:

```r
myfunction <- function(x) {
  output <- (20 + (x*x))
  return(output)
  }
```

This function will calculate the square of a number (x), and then add 20 to that value. It stores this in a temporary object called output. The return() statement returns the calculated value. Once you have typed in this function, the function is then available for use. For example, we can use the function for different input numbers (e.g.. 10, 25):

```r
myfunction(10)
```

```
## [1] 120
```

```r
myfunction(25)
```

```
## [1] 645
```

You can view the code that makes up a function by typing its name (without any parentheses). For example, we can try this by typing "myfunction":

```r
myfunction
```

```
## function(x) {
##    output <- (20 + (x*x))
##    return(output)
##    }
## <bytecode: 0x7f822cb43c88>
```

## 17.5   Comments in R

When you are typing R, if you want to, you can write comments by writing the comment text after the "#" sign. This can be useful if you want to write some R commands that other people need to read and understand. R will ignore the comments when it is executing the commands. For example, you may want to write a comment to explain what the function log10() does:

```r
x <- 100
log10(x) # Finds the log to the base 10 of variable x.
```

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

# Chapter 18

# FASTA Files

Adapted from Wikipedia: https://en.wikipedia.org/wiki/FASTA_format

"In bioinformatics, the FASTA format is a text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes. The format allows for sequence names and comments to precede the sequences. The format originates from the FASTA alignment software, but has now become a near universal standard in the field of bioinformatics.

"The simplicity of FASTA format makes it easy to manipulate and parse sequences using text-processing tools and scripting languages like the R programming language and Python.

"The first line in a FASTA file starts with a">" (greater-than) symbol and holds summary information about the sequence, often starting with a unique accession number and followed by information like the name of the gene, the type of sequence, and the organism it is from.

"On the next is the sequence itself in a standard one-letter character string. Anything other than a valid character is be ignored (including spaces, tabs, asterisks, etc...).

"A multiple sequence FASTA format can be obtained by concatenating several single sequence FASTA files in a common file (also known as multi-FASTA format).

"Following the header line, the actual sequence is represented. Sequences may be protein sequences or nucleic acid sequences, and they can contain gaps or alignment characters. Sequences are expected to be represented in the standard amino acid and nucleic acid codes. Lower-case letters are accepted and are mapped into upper-case; a single hyphen or dash can be used to represent a gap character; and in amino acid sequences, U and * are acceptable letters.

"FASTQ format is a form of FASTA format extended to indicate information related to sequencing. It is created by the Sanger Centre in Cambridge.

"Bioconductor.org's Biostrings package can be used to read and manipulate FASTA files in R

from https://zhanglab.dcmb.med.umich.edu/FASTA/

"FASTA format is a text-based format for representing either nucleotide sequences or peptide sequences, in which base pairs or amino acids are represented using single-letter codes. A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol in the first column. It is recommended that all lines of text be shorter than 80 characters in length."

## 18.1   Example FASTA file

Here is an example of the contents of a FASTA file. (If your are viewing this chapter in the form of the source .Rmd file, the `cat()` function is included just to print out the content properly and is not part of the FASTA format).

```
cat(">gi|186681228|ref|YP_001864424.1| phycoerythrobilin:ferredoxin oxidoreductase
MNSERSDVTLYQPFLDYAIAYMRSRLDLEPYPIPTGFESNSAVVGKGKNQEEVVTTSYAFQTAKLRQIRA
AHVQGGNSLQVLNFVIFPHLNYDLPFFGADLVTLPGGHLIALDMQPLFRDDSAYQAKYTEPILPIFHAHQ
QHLSWGGDFPEEAQPFFSPAFLWTRPQETAVVETQVFAAFKDYLKAYLDFVEQAEAVTDSQNLVAIKQAQ
LRYLRYRAEKDPARGMFKRFYGAEWTEEYIHGFLFDLERKLTVVK")
```

```
## >gi|186681228|ref|YP_001864424.1| phycoerythrobilin:ferredoxin oxidoreductase
## MNSERSDVTLYQPFLDYAIAYMRSRLDLEPYPIPTGFESNSAVVGKGKNQEEVVTTSYAFQTAKLRQIRA
## AHVQGGNSLQVLNFVIFPHLNYDLPFFGADLVTLPGGHLIALDMQPLFRDDSAYQAKYTEPILPIFHAHQ
## QHLSWGGDFPEEAQPFFSPAFLWTRPQETAVVETQVFAAFKDYLKAYLDFVEQAEAVTDSQNLVAIKQAQ
## LRYLRYRAEKDPARGMFKRFYGAEWTEEYIHGFLFDLERKLTVVK
```

## 18.2   Multiple sequences in a single FASTA file

Multiple sequences can be stored in a single FASTA file, each on separated by a line and have its own headline.

```
cat(">LCBO - Prolactin precursor - Bovine
MDSKGSSQKGSRLLLLLVVSNLLLCQGVVSTPVCPNGPGNCQVSLRDLFDRAVMVSHYIHDLSS
EMFNEFDKRYAQGKGFITMALNSCHTSSLPTPEDKEQAQQTHHEVLMSLILGLLRSWNDPLYHL
VTEVRGMKGAPDAILSRAIEIEEENKRLLEGMEMIFGQVIPGAKETEPYPVWSGLPSLQTKDED
ARYSAFYNLLHCLRRDSSKIDTYLKLLNCRIIYNNNC*

>MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken
MADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAELQDMINEVDADGNGTID
```

```
FPEFLTMMARKMKDTDSEEEIREAFRVFDKDGNGYISAAELRHVMTNLGEKLTDEEVDEMIREA
DIDGDGQVNYEEFVQMMTAK*

>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY")
```

```
## >LCBO - Prolactin precursor - Bovine
## MDSKGSSQKGSRLLLLLVVSNLLLCQGVVSTPVCPNGPGNCQVSLRDLFDRAVMVSHYIHDLSS
## EMFNEFDKRYAQGKGFITMALNSCHTSSLPTPEDKEQAQQTHHEVLMSLILGLLRSWNDPLYHL
## VTEVRGMKGAPDAILSRAIEIEEENKRLLEGMEMIFGQVIPGAKETEPYPVWSGLPSLQTKDED
## ARYSAFYNLLHCLRRDSSKIDTYLKLLNCRIIYNNNC*
##
## >MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken
## MADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAELQDMINEVDADGNGTID
## FPEFLTMMARKMKDTDSEEEIREAFRVFDKDGNGYISAAELRHVMTNLGEKLTDEEVDEMIREA
## DIDGDGQVNYEEFVQMMTAK*
##
## >gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
## LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
## EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
## LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
## GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
## IENY
```

## 18.3 Multiple sequence alignments can be stored in FASTA format

**Aligned FASTA format** can be used to store the output of **Multiple Sequence Alignment (MSA)**. This format contains

1. Multiple entries, each with their own header line
2. **Gaps** inserted to align sequences are indicated by .
3. Each spaces added to the beginning and end of sequences that vary in length are indicated by ~

In the sample FASTA file below, the `example1` sequence has a gap of 8 near its beginning. The `example2` sequence has numerous ~ indicating that this sequence is missing data from its beginning that are present in the other sequences. The `example3` sequence has numerous ~ at its end, indicating that this sequence is shorter than the others.

```
cat(">example1
MKALWALLLVPLLTGCLA........EGELEVTDQLPGQSDQP.WEQALNRFWDYLRWVQ
GNQARDRLEEVREQMEEVRSKMEEQTQQIRLQAEIFQARIKGWFEPLVEDMQRQWANLME
KIQASVATNSIASTTVPLENQ
>example2
~~~~~~~~~~~~~~~~~~~~~~~~~KVQQELEPEAGWQTGQP.WEAALARFWDYLRWVQ
SSRARGHLEEMREQIQEVRVKMEEQADQIRQKAEAFQARLKSWFEPLLEDMQRQWDGLVE
KVQAAVAT.IPTSKPVEEP~~
>example3
MRSLVVFFALAVLTGCQARSLFQAD..............APQPRWEEMVDRFWQYVSELN
AGALKEKLEETAENL...RTSLEGRVDELTSLLAPYSQKIREQLQEVMDKIKEATAALPT
QA~~~~~~~~~~~~~~~~~~")
```

```
## >example1
## MKALWALLLVPLLTGCLA........EGELEVTDQLPGQSDQP.WEQALNRFWDYLRWVQ
## GNQARDRLEEVREQMEEVRSKMEEQTQQIRLQAEIFQARIKGWFEPLVEDMQRQWANLME
## KIQASVATNSIASTTVPLENQ
## >example2
## ~~~~~~~~~~~~~~~~~~~~~~~~~KVQQELEPEAGWQTGQP.WEAALARFWDYLRWVQ
## SSRARGHLEEMREQIQEVRVKMEEQADQIRQKAEAFQARLKSWFEPLLEDMQRQWDGLVE
## KVQAAVAT.IPTSKPVEEP~~
## >example3
## MRSLVVFFALAVLTGCQARSLFQAD..............APQPRWEEMVDRFWQYVSELN
## AGALKEKLEETAENL...RTSLEGRVDELTSLLAPYSQKIREQLQEVMDKIKEATAALPT
## QA~~~~~~~~~~~~~~~~~~
```

## 18.4   FASTQ Format

Adapted from Wikipedia: https://en.wikipedia.org/wiki/FASTQ_format

"FASTQ format is a text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores. Both the sequence letter and quality score are each encoded with a single ASCII character for brevity.

"It was originally developed at the Wellcome Trust Sanger Institute to bundle a FASTA formatted sequence and its quality data, but has recently become the de facto standard for storing the output of high-throughput sequencing instruments such as the Illumina Genome Analyzer.

"A FASTQ file normally uses four lines per sequence.

- Line 1 begins with a @ character and is followed by a sequence identifier and an optional description (like a FASTA title line).
- Line 2 is the raw sequence letters.
- Line 3 begins with a + character and is optionally followed by the same sequence identifier (and any description) again.

- Line 4 encodes the **quality values** for the sequence in Line 2 of the file, and must contain the same number of symbols as letters in the sequence.

"A FASTQ file containing a single sequence might look like this:"

```
cat("@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((*** +))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65")
```

```
## @SEQ_ID
## GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
## +
## !''*((((*** +))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

"Here are the quality value characters in left-to-right increasing order of quality (ASCII):"

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

FASTQ files typically do not include line breaks and do not wrap around when they reach the width of a normal page or file.

## Chapter 19

# Downloading DNA sequences as FASTA files in R

This is a modification of "DNA Sequence Statistics" from Avril Coghlan's *A little book of R for bioinformatics.*. Most of the text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

**NOTE**: There is some reduncnacy in this current draft that needs to be eliminated.

### 19.0.1 Functions

- library()
- help()
- length
- table
- seqinr::GC()
- seqinr::count()
- seqinr::write.fasta()

### 19.0.2 Software/websites

- www.ncbi.nlm.nih.gov
- Text editors (e.g. Notepad++, TextWrangler)

### 19.0.3 R vocabulary

- list

- library
- package
- CRAN
- wrapper

### 19.0.4   File types

- FASTA

### 19.0.5   Bioinformatics vocabulary

- accession, accession number
- NCBI
- NCBI Sequence Database
- EMBL Sequence Database
- FASTA file

### 19.0.6   Organisms and Sequence accessions

- Dengue virus: DEN-1, DEN-2, DEN-3, and DEN-4.

The NCBI accessions for the DNA sequences of the DEN-1, DEN-2, DEN-3, and DEN-4 Dengue viruses are NC_001477, NC_001474, NC_001475 and NC_002640, respectively.

According to Wikipedia

> "Dengue virus (DENV) is the cause of dengue fever. It is a mosquito-borne, single positive-stranded RNA virus … Five serotypes of the virus have been found, all of which can cause the full spectrum of disease. Nevertheless, scientists' understanding of dengue virus may be simplistic, as rather than distinct … groups, a continuum appears to exist." https://en.wikipedia.org/wiki/Dengue_virus

### 19.0.7   Preliminaries

```
library(rentrez)
library(compbio4all)
```

## 19.1   DNA Sequence Statistics: Part 1

### 19.1.1   Using R for Bioinformatics

The chapter will guide you through the process of using R to carry out simple analyses that are common in bioinformatics and computational biology. In particular, the focus is on computational analysis of biological sequence data

such as genome sequences and protein sequences. The programming approaches, however, are broadly generalizable to statistics and data science.

The tutorials assume that the reader has some basic knowledge of biology, but not necessarily of bioinformatics. The focus is to explain simple bioinformatics analysis, and to explain how to carry out these analyses using *R*.

### 19.1.2 R packages for bioinformatics: Bioconductor and SeqinR

Many authors have written *R* packages for performing a wide variety of analyses. These do not come with the standard *R* installation, but must be installed and loaded as "add-ons".

Bioinformaticians have written numerous specialized packages for *R*. In this tutorial, you will learn to use some of the function in the `SeqinR` package to to carry out simple analyses of DNA sequences. (`SeqinR` can retrieve sequences from a DNA sequence database, but this has largely been replaced by the functions in the package `rentrez`)

Many well-known bioinformatics packages for *R* are in the Bioconductor set of *R* packages (www.bioconductor.org), which contains packages with many *R* functions for analyzing biological data sets such as microarray data. The `SeqinR` package is from CRAN, which contains R functions for obtaining sequences from DNA and protein sequence databases, and for analyzing DNA and protein sequences.

We will also use functions from the `rentrez` and `ape` packages.

Remember that you can ask for more information about a particular *R* command by using the `help()` function. For example, to ask for more information about the `library()`, you can type:

```
help("library")
```

You can also do this

```
?library
```

### 19.1.3 FASTA file format

The FASTA format is a simple and widely used format for storing biological (e.g. DNA or protein) sequences. It was first used by the FASTA program for sequence alignment in the 1980s and has been adopted as standard by many other programs.

FASTA files begin with a single-line description starting with a greater-than sign > character, followed on the next line by the sequences. Here is an example of a FASTA file. (If you're looking at the source script for this lesson you'll see

the `cat()` command, which is just a text display function used format the text when you run the code).

```
## >A06852 183 residues MPRLFSYLLGVWLLLSQLPREIPGQSTNDFIKACGRELVRLWVEICGSVSWGRTALSLEEPQI
```

### 19.1.4   The NCBI sequence database

The US National Centre for Biotechnology Information (NCBI) maintains the **NCBI Sequence Database**, a huge database of all the DNA and protein sequence data that has been collected. There are also similar databases in Europe, the European Molecular Biology Laboratory (EMBL) Sequence Database, and Japan, the DNA Data Bank of Japan (DDBJ). These three databases exchange data every night, so at any one point in time, they contain almost identical data.

Each sequence in the NCBI Sequence Database is stored in a separate **record**, and is assigned a unique identifier that can be used to refer to that record. The identifier is known as an **accession**, and consists of a mixture of numbers and letters.

For example, Dengue virus causes Dengue fever, which is classified as a **neglected tropical disease** by the World Health Organization (WHO), is classified by any one of four types of Dengue virus: DEN-1, DEN-2, DEN-3, and DEN-4. The NCBI accessions for the DNA sequences of the DEN-1, DEN-2, DEN-3, and DEN-4 Dengue viruses are

- NC_001477
- NC_001474
- NC_001475

- NC_002640

Note that because the NCBI Sequence Database, the EMBL Sequence Database, and DDBJ exchange data every night, the DEN-1 (and DEN-2, DEN-3, DEN-4) Dengue virus sequence are present in all three databases, but they have different accessions in each database, as they each use their own numbering systems for referring to their own sequence records.

### 19.1.5   Retrieving genome sequence data using rentrez

You can retrieve sequence data from NCBI directly from *R* using the `rentrez` package.  The DEN-1 Dengue virus genome sequence has NCBI accession NC_001477. To retrieve a sequence with a particular NCBI accession, you can use the function `entrez_fetch()` from the `rentrez` package. Note that to be specific where the function comes from I write it as `package::function()`.

```
dengueseq_fasta <- rentrez::entrez_fetch(db = "nucleotide",
                                          id = "NC_001477",
                                          rettype = "fasta")
```

Note that the "*_*" *in the name is just an arbitrary way to separate two words. Another common format would be* **dengueseq.fasta**. *Some people like* **dengueseqFasta**, *called* **camel case** *because the capital letter makes a hump in the middle of the word. Underscores are becoming most common and are favored by developers associated with RStudio and the* **tidyverse** *of packages that many data scientists use. I switch between "." and ""* as separators, usually favoring "_" for function names and "." for objects; I personally find camel case harder to read and to type.

Ok, so what exactly have we done when we made `dengueseq_fasta`? We have an R object `dengueseq_fasta` which has the sequence linked to the accession number "NC_001477." So where is the sequence, and what is it?

First, what is it?

```
is(dengueseq_fasta)
```

```
## [1] "character"              "vector"              "data.frameRowLabels"
## [4] "SuperClassMethod"
```

```
class(dengueseq_fasta)
```

```
## [1] "character"
```

How big is it? Try the `dim()` and `length()` commands and see which one works. Do you know why one works and the other doesn't?

```
dim(dengueseq_fasta)
```

```
## NULL
```

```
length(dengueseq_fasta)
```

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

The size of the object is 1. Why is this? This is the genomics sequence of a virus, so you'd expect it to be fairly large. We'll use another function below to explore that issue. Think about this first: how many pieces of unique information are in the `dengueseq` object? In what sense is there only *one* piece of information?

If we want to actually see the sequence we can type just type `dengueseq_fasta` and press enter. This will print the WHOLE genomic sequence out but it will probably run of your screen.

```
dengueseq_fasta
```

This is a whole genome sequence, but its stored as single entry in a vector, so the `length()` command just tells us how many entries there are in the vector, which is just one! What this means is that the entire genomic sequence is stored in a single entry of the vector `dengueseq_fasta`. (If you're not following along with this, no worries - its not essential to actually working with the data)

If we want to actually know how long the sequence is, we need to use the function `nchar()`.

```
nchar(dengueseq_fasta)
```

```
## [1] 10935
```

The sequence is 10935 bases long. All of these bases are stored as a single **character string** with no spaces in a single entry of our `dengueseq_fasta` vector. This isn't actually a useful format for us, so below were' going to convert it to something more useful.

If we want to see just part of the sequence we can use the `strtrim()` function. Before you run the code below, predict what the 100 means.

```
strtrim(dengueseq_fasta, 100)
```

```
## [1] ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACAG
```

Note that at the end of the name is a slash followed by an n, which indicates to the computer that this is a **newline**; this is read by text editor, but is ignored by R in this context.

```
strtrim(dengueseq_fasta, 45)
```

```
## [1] ">NC_001477.1 Dengue virus 1, complete genome\n"
```

After the `\n` begins the sequence, which will continue on for a LOOOOOONG way. Let's just print a little bit.

```
strtrim(dengueseq_fasta, 52)
```

```
## [1] ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTT"
```

Let's print some more. Do you notice anything beside A, T, C and G in the sequence?

```
strtrim(dengueseq_fasta, 200)
```

```
## [1] ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACAG
```

Again, there are the `\n` newline characters, which tell text editors and word-processors how to display the file.

Now that we a sense of what we're looking at let's explore the `dengueseq_fasta` a bit more.

We can find out more information about what it is using the `class()`command.

```
class(dengueseq_fasta)
```

```
## [1] "character"
```

As noted before, this is character data.

Many things in R are vectors so we can ask *R* `is.vector()`

```
is.vector(dengueseq_fasta)
```

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

Yup, that's true.

Ok, let's see what else. A handy though often verbose command is `is()`, which tells us what an object, well, what it is:

```
is(dengueseq_fasta)
```

```
## [1] "character"          "vector"            "data.frameRowLabels"
## [4] "SuperClassMethod"
```

There is a lot here but if you scan for some key words you will see "character" and "vector" at the top. The other stuff you can ignore. The first two things, though, tell us the dengueseq_fasta is a **vector** of the class **character**: that is, a **character vector**.

Another handy function is `str()`, which gives us a peak at the context and structure of an *R* object. This is most useful when you are working in the R console or with dataframes, but is a useful function to run on all *R* objects. How does this output differ from other ways we've displayed dengueseq_fasta?

```
str(dengueseq_fasta)
```

```
##  chr ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCG
```

We know it contains character data - how many characters? `nchar()` for "number of characters" answers that:

```
nchar(dengueseq_fasta)
```

```
## [1] 10935
```

## 19.2 OPTIONAL: Saving FASTA files

We can save our data as .fasta file for safe keeping. The `write()` function will save the data we downloaded as a plain text file.

```
write(dengueseq_fasta,
      file="dengueseq.fasta")
```

If you do this, you'll need to figure out where *R* is saving things, which requires and understanding *R's* **file system**, which can take some getting used to, especially if you're new to programming. As a start, you can see where *R* saves things by using the `getwd()` command, which tells you where on your harddrive R currently is using as its home base for files.

```
getwd()
```

```
## [1] "/Users/nlb24/google_backup_sync_nlb24/lbrb"
```

## 19.3 Next steps

On their own, FASTA files in R are not directly useful. In the next lesson we'll process our `dengueseq_fasta` file so that we can use it in analyses.

# Chapter 20

# Downloading DNA sequences as FASTA files in R

This is a modification of "DNA Sequence Statistics" from Avril Coghlan's *A little book of R for bioinformatics..* Most of the text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

## 20.1   Preliminaries

We'll need the `dengueseq_fasta` FASTA data object, which is in the `compbio4all` package. We'll also use the `stringr` package for cleaning up the FASTA data, which can be downloaded with `install.packages("stringr")`

```
# compbio4all, which has dengueseq_fasta
library(compbio4all)
data(dengueseq_fasta)

# stringr, for data cleaning
library(stringr)
```

## 20.2   Convert FASTA sequence to an R variable

We can't actually do much with the contents of the `dengueseq_fasta` we downloaded with the `rentrez` package except read them. If we want do address some biological questions with the data we need is to convert it into a data structure *R* can work with.

There are several things we need to remove:

1. The **meta data** line `>NC_001477.1 Dengue virus 1, complete genome` (metadata is "data" about data, such as where it came from, what it is, who made it, etc.).
2. All the `\n` that show up in the file (these are the **line breaks**).
3. Put each nucleotide of the sequence into its own spot in a vector.

There are functions that can do this automatically, but

1. I haven't found one I like, and
2. walking through this will help you understand the types of operations you can do on text data.

The first two steps involve removing things from the existing **character string** that contains the sequence. The third step will split the single continuous character string like "AGTTGTTAGTCTACGT…" into a **character vector** like `c("A","G","T","T","G","T","T","A","G","T","C","T","A","C","G","T"...)`, where each element of the vector is a single character a stored in a separate slot in the vector.

## 20.2.1 Removing unwanted characters

The second item is the easiest to take care of. *R* and many programming languages have tools called **regular expressions** that allow you to manipulate text. R has a function called `gsub()` which allows you to substitute or delete character data from a string. First I'll remove all those `\n` values.

The regular expression function `gsub()` takes three arguments: 1. `pattern = ....` This is what we need it to find so we can replace it. 1. `replacement = ....` The replacement. 1. `x = ....` A character string or vector where `gsub()` will do its work.

We need to get rid of the `\n` so that we are left with only A, T and G, which are the actually information of the sequence. We want `\n` completely removed, so the replacement will be `""`, which is as set of quotation marks with nothing in the middle, which means "delete the target pattern and put nothing in its place."

One thing that is tricky about regular expressions is that many characters have special meaning to the functions, such as slashes, dollar signs, and brackets. So, if you want to find and replace one of these specially designated characters you need to put a slash in front of them. So when we set the pattern, instead of setting the pattern to a slash before an n `\n`, we have to give it two slashes `\\n`.

Here is the regular expression to delete the newline character \n.

```r
# note: we want to find all the \n, but need to set the pattern as \\n
dengueseq_vector <- gsub(pattern = "\\n",
```

```
                              replacement = "",
                              x = dengueseq_fasta)
```

We can use **strtrim()** to see if it worked

```
strtrim(dengueseq_vector, 80)
```

```
## [1] ">NC_001477.1 Dengue virus 1, complete genomeAGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTC"
```

Now for the metadata header. This is a bit complex, but the following code is going to take all the that occurs before the beginning of the sequence ("AGTTGT-TAGTC") and delete it.

First, I'll define what I want to get rid of in an *R* object. This will make the **call** to **gsub()** a little cleaner to read

```
seq.header <- ">NC_001477.1 Dengue virus 1, complete genome"
```

Now I'll get rid of the header with with **gsub()**.

```
dengueseq_vector <- gsub(pattern = seq.header, # object defined above
                         replacement = "",
                         x = dengueseq_vector)
```

See if it worked:

```
strtrim(dengueseq_vector, 80)
```

```
## [1] "AGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCGGAAGCTTGCTTAACGTAGTTCTAACAGTTTTTTATTAG"
```

## 20.2.2  Spliting unbroken strings in character vectors

Now the more complex part. We need to split up a continuous, unbroken string of letters into a vector where each letter is on its own. This can be done with the **str_split()** function ("string split") from the **stringr** package. The notation **stringr::str_split()** mean "use the **str_split** function from from the **stringr** package." More specifically, it temporarily loads the **stringr** package and gives R access to just the **str_split** function. These allows you to call a single function without loading the whole library.

There are several arguments to **str_split**, and I've tacked a **[[1]]** on to the end.

First, run the command

```
dengueseq_vector_split <- stringr::str_split(dengueseq_vector,
                                             pattern = "",
                                             simplify = FALSE)[[1]]
```

Look at the output with str()

```
str(dengueseq_vector_split)
```

```
##  chr [1:10735] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T" "C" "T" "A" "C" "G" ...
```

We can explore what the different arguments do by modifying them. Change
`pattern = ""` to `pattern = "A"`. Can you figure out what happened?

```
# re-run the command with "pattern  = "A"
dengueseq_vector_split2 <- stringr::str_split(dengueseq_vector,
                                     pattern = "A",
                                     simplify = FALSE)[[1]]
str(dengueseq_vector_split2)
```

```
##  chr [1:3427] "" "GTTGTT" "GTCT" "CGTGG" "CCG" "C" "" "G" "" "C" "GTTTCG" ...
```

And try it with `pattern = ""` to `pattern = "G"`.

```
# re-run the command with "pattern  = "G"
dengueseq_vector_split3 <- stringr::str_split(dengueseq_vector,
                                     pattern = "G",
                                     simplify = FALSE)[[1]]
str(dengueseq_vector_split3)
```

```
##  chr [1:2771] "A" "TT" "TTA" "TCTAC" "T" "" "ACC" "ACAA" "AACA" "TTTC" ...
```

Run this code to compare the two ways we just used `str_split` (don't worry
what it does). Does this help you see what's up?

```
options(str = strOptions(vec.len = 10))
str(list(dengueseq_vector_split[1:20],
    dengueseq_vector_split2[1:10],
    dengueseq_vector_split3[1:10]))
```

```
## List of 3
##  $ : chr [1:20] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T" ...
##  $ : chr [1:10] "" "GTTGTT" "GTCT" "CGTGG" "CCG" "C" "" "G" "" "C"
##  $ : chr [1:10] "A" "TT" "TTA" "TCTAC" "T" "" "ACC" "ACAA" "AACA" "TTTC"
```

So, what does the `pattern = ...` argument do? For more info open up the
help file for `str_split` by calling `?str_split`.

Something cool which we will explore in the next exercise is that we can do
summaries on vectors of nucleotides, like this:

```
table(dengueseq_vector_split)
```

```
## dengueseq_vector_split
##    A    C    G    T
## 3426 2240 2770 2299
```

# Chapter 21

# Downloading protein sequences in *R*

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 21.1 Preliminaries

```
library(compbio4all)
```

## 21.2 Retrieving a UniProt protein sequence using rentrez

We can use `entrez_fetch()` to download protein sequences.

For example to retrieve the protein sequences for UniProt accessions Q9CD83 and A0PQ23, we type in R:

```
# sequence 1: Q9CD83
leprae_fasta <- rentrez::entrez_fetch(db = "protein",
                        id = "Q9CD83",
                         rettype = "fasta")
# sequence 2: OIN17619.1
ulcerans_fasta <- rentrez::entrez_fetch(db = "protein",
                        id = "OIN17619.1",
                         rettype = "fasta")
```

Display the contents of the `lepraeseq` FASTA file.

```
leprae_fasta
```

```
## [1] ">sp|Q9CD83.1|PHBS_MYCLE RecName: Full=Chorismate pyruvate-lyase; AltName: Full=
```

Let's clean these up to remove the header and new line characters usin the function `fasta_cleaner()`.

```
leprae_vector   <- fasta_cleaner(leprae_fasta)
ulcerans_vector <- fasta_cleaner(ulcerans_fasta)
```

Examine the output using `length()`, `class()`, and `head()`:

```
length(leprae_vector)
class(leprae_vector)
head(leprae_vector, 20)
```

# Chapter 22

# Sequence dotplots in *R*

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

**NOTE: I've added some new material that is rather terse and lacks explication.**

## 22.1 Preliminaries

```r
library(compbio4all)
```

### 22.1.1 Download sequences

As we did in the previous lesson on dotplots, we'll look at two sequences.

```r
# sequence 1: Q9CD83
leprae_fasta <- rentrez::entrez_fetch(db = "protein",
                          id = "Q9CD83",
                           rettype = "fasta")
# sequence 2: OIN17619.1
ulcerans_fasta <- rentrez::entrez_fetch(db = "protein",
                          id = "OIN17619.1",
                          rettype = "fasta")

leprae_vector   <- fasta_cleaner(leprae_fasta)
ulcerans_vector <- fasta_cleaner(ulcerans_fasta)
```

## 22.2   Visualzing two identical sequences

To help build our intuition about dotplots we'll first look at some artificial examples. First, we'll see what happens when we make a dotplot comparing the alphabet versus itself. The build-in `LETTERS` object in R contains the alphabet from A to Z. This is a sequence with no repeats.

```
seqinr::dotPlot(LETTERS,
                LETTERS)
```
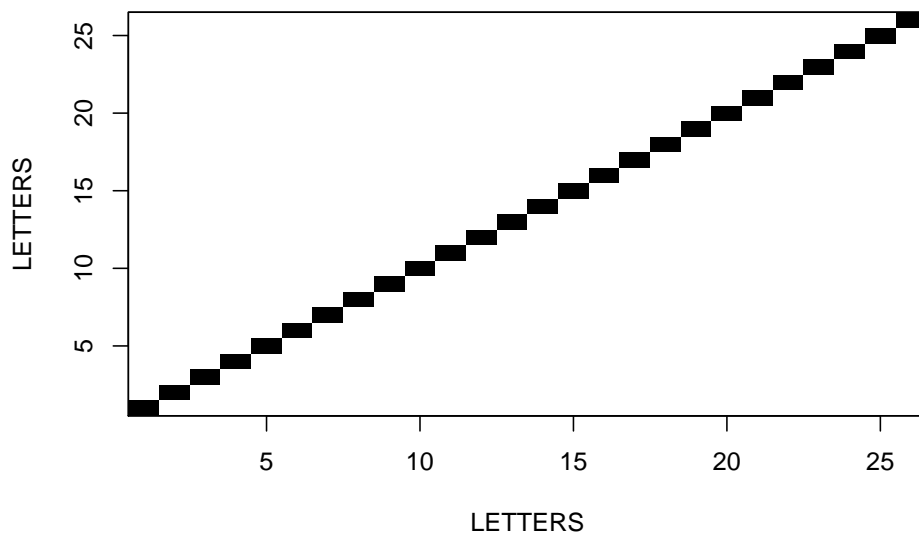


What we get is a perfect diagonal line.

## 22.3   Visualizing repeats

Now lets' make a sequence where the alphabet gets repeats twice

```
LETTERS.2.times <- c(LETTERS,LETTERS)

seqinr::dotPlot(LETTERS.2.times,
                LETTERS.2.times)
```
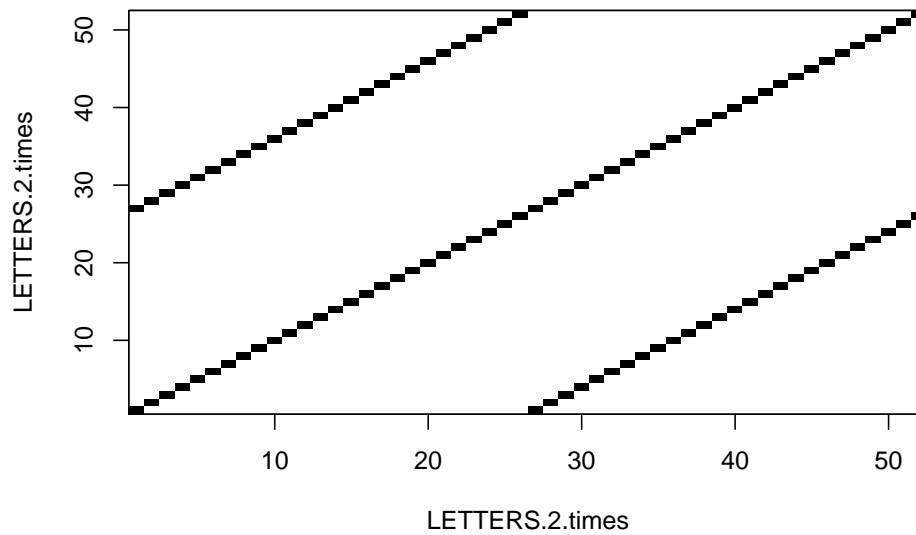
Note the diagonal lines.

Now 3 repeats

```
LETTERS.3.times <- c(LETTERS,LETTERS,LETTERS)

seqinr::dotPlot(LETTERS.3.times,
                LETTERS.3.times)
```
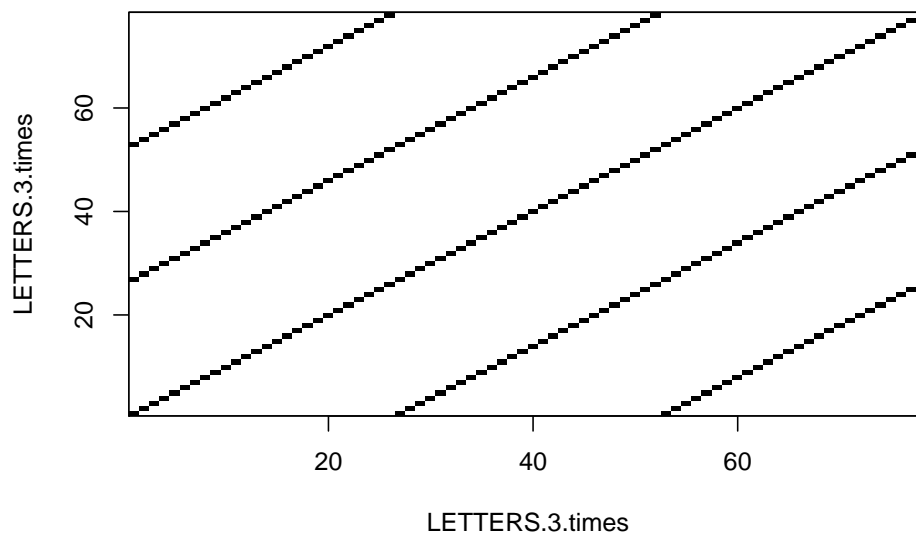


Here's another example of repeats.

Create sequence with repeats:

```r
seq.repeat <- c("A","C","D","E","F","G","H","I")
seq1 <- rep(seq.repeat,3)
```

Make the dotplot:

```r
seqinr::dotPlot(seq1,
                seq1)
```
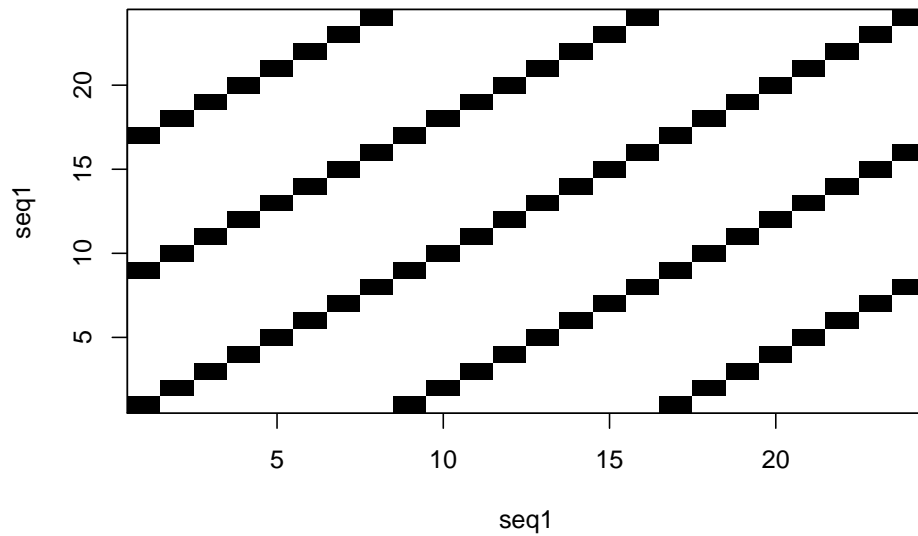


## 22.4   Inversions

See if you can figure out what's going on here.

```r
LETTERS.3.times.with.invert <- c(LETTERS,rev(LETTERS),LETTERS)

seqinr::dotPlot(LETTERS.3.times,
                LETTERS.3.times.with.invert)
```
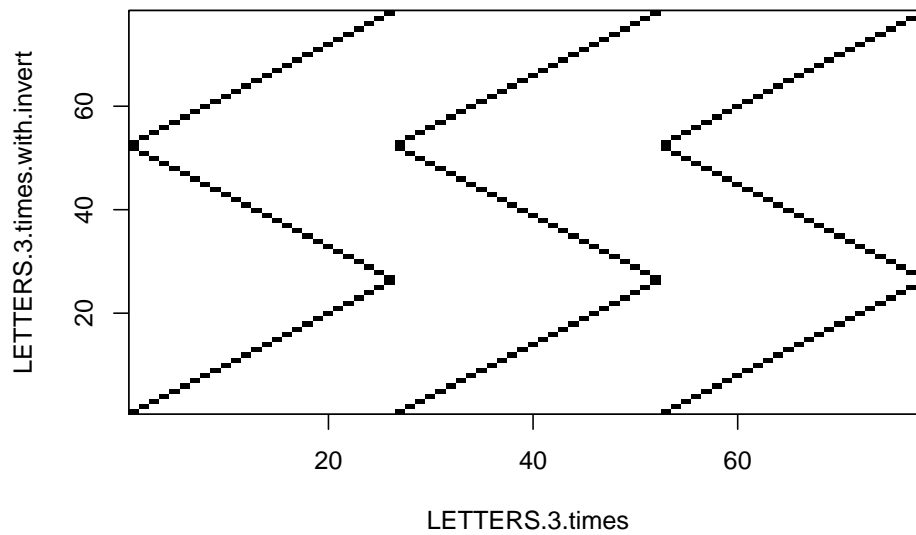
## 22.5  Translocations

See if you can figure out what's going on here.

```
seg1 <- LETTERS[1:8]
seg2 <- LETTERS[9:18]
seg3 <- LETTERS[18:26]

LETTERS.with.transloc <- c(seg1,seg3,seg2)

seqinr::dotPlot(LETTERS,
                LETTERS.with.transloc)
```
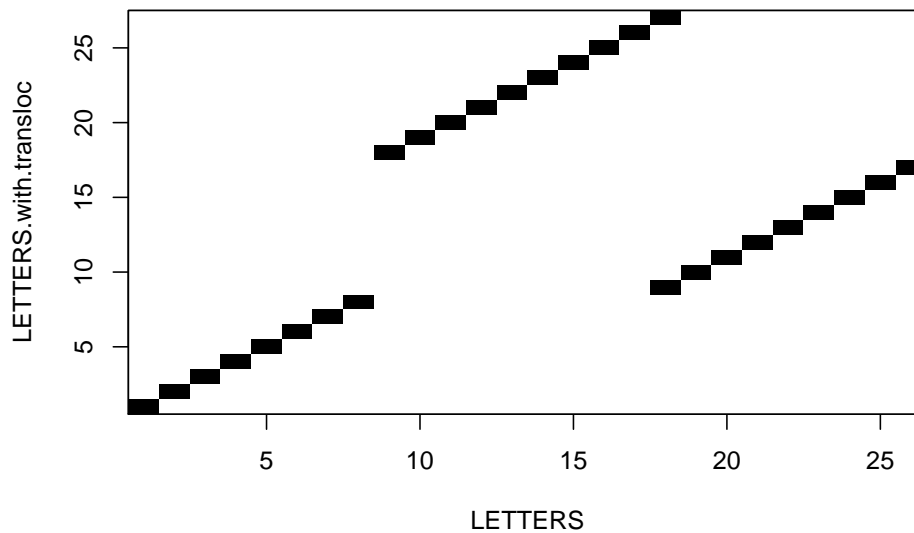
## 22.6  Random sequence

```
letters.rand1 <- sample(x = LETTERS, size = 26, replace = F)
letters.rand2 <- sample(x = LETTERS, size = 26, replace = F)


seqinr::dotPlot(letters.rand1,
                letters.rand2)
```
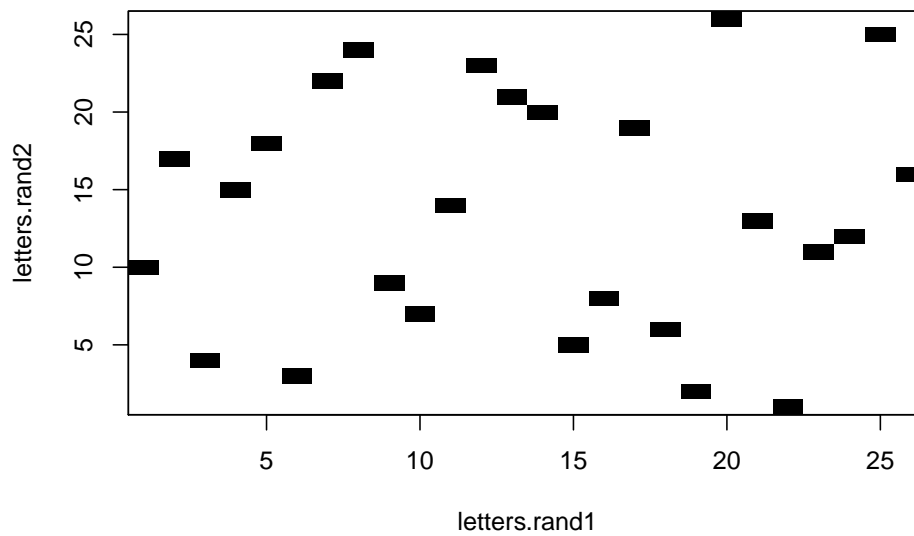
## 22.7 Comparing two real sequences using a dotplot

As a first step in comparing two protein, RNA or DNA sequences, it is a good idea to make a **dotplot**. A **dotplot** is a graphical method that allows the comparison of two protein or DNA sequences and identify regions of close similarity between them. A dotplot is essentially a two-dimensional matrix (like a grid), which has the sequences of the proteins being compared along the vertical and horizontal axes.

In order to make a simple dotplot to represent of the similarity between two sequences, individual cells in the matrix can be shaded black if residues are identical, so that matching sequence segments appear as runs of diagonal lines across the matrix. Identical proteins will have a line exactly on the main diagonal of the dotplot, that spans across the whole matrix.

For proteins that are not identical, but share regions of similarity, the dotplot will have shorter lines that may be on the **main diagonal**, or off the main diagonal of the matrix. In essence, a dotplot will reveal if there are any regions that are clearly very similar in two protein (or DNA) sequences.

We can create a dotplot for two sequences using the `dotPlot()` function in the `seqinr` package.

First, let's look at a dotplot created using only a single sequence. You'd never do this in practice, but it will give you a sense of what dotplots are doing.

```
seqinr::dotPlot(leprae_vector,
                leprae_vector)
```
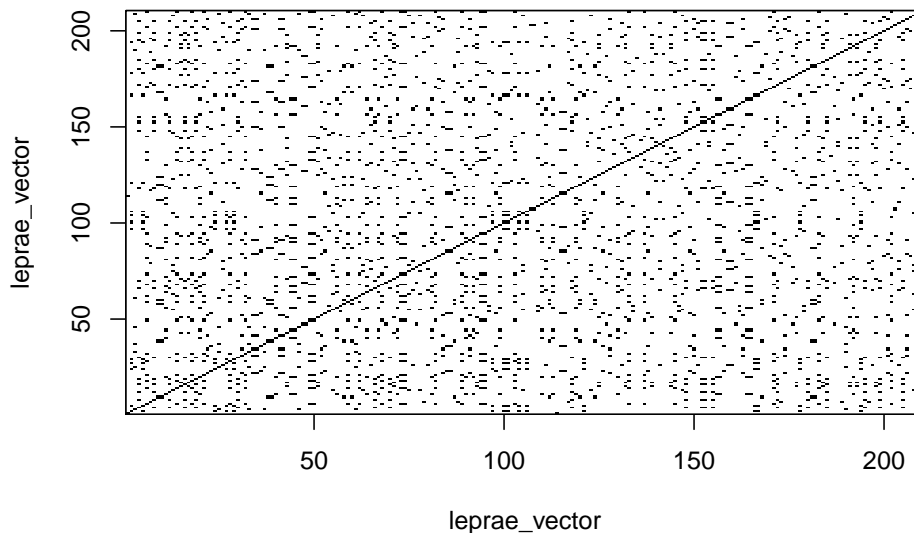


These two sequences are identical, so we have a very distinct diagonal line. But

there's also other

Now we'll make a real dotplot of the chorismate lyase proteins from two closely related species, *Mycobacterium leprae* and *Mycobacterium ulcerans*.

```
seqinr::dotPlot(leprae_vector,
                ulcerans_vector)
```
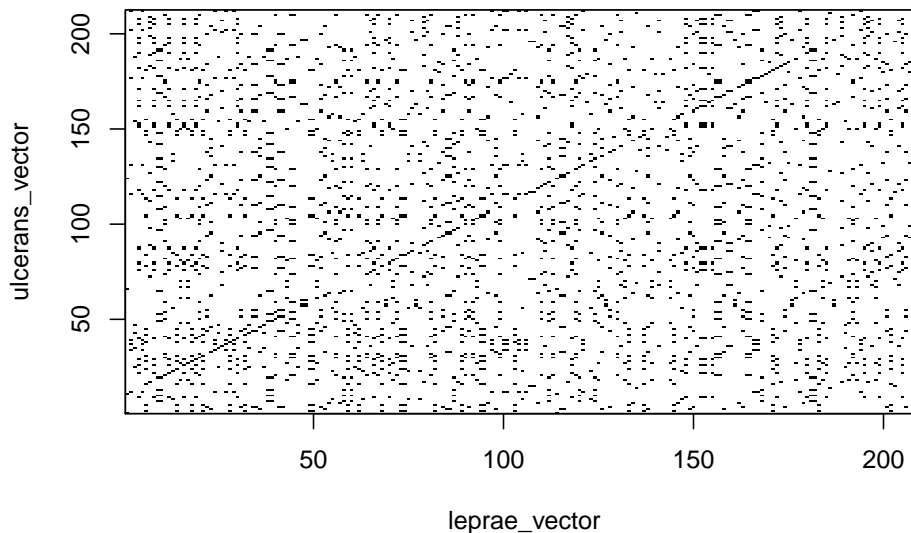


In the dotplot above, the *M. leprae* sequence is plotted along the x-axis (horizontal axis), and the *M. ulcerans* sequence is plotted along the y-axis (vertical axis). The dotplot displays a dot at points where there is an identical amino acid in the two sequences.

For example, if amino acid 53 in the *M. leprae* sequence is the same amino acid (eg. "W") as amino acid 70 in the M. ulcerans sequence, then the dotplot will show a dot the position in the plot where x =50 and y =53.

In this case you can see a lot of dots along a diagonal line, which indicates that the two protein sequences contain many identical amino acids at the same (or very similar) positions along their lengths. This is what you would expect, because we know that these two proteins are **homologs** (related proteins) because they share a close evolutionary history.

# Chapter 23

# Global proteins aligments in *R*

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 23.1   Preliminaries

```
library(compbio4all)

library(Biostrings)
```

```
## Loading required package: BiocGenerics

## Loading required package: parallel

##
## Attaching package: 'BiocGenerics'

## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB

## The following objects are masked from 'package:stats':
##
##     IQR, mad, sd, var, xtabs

## The following objects are masked from 'package:base':
```

```
##
##      anyDuplicated, append, as.data.frame, basename, cbind, colnames,
##      dirname, do.call, duplicated, eval, evalq, Filter, Find, get, grep,
##      grepl, intersect, is.unsorted, lapply, Map, mapply, match, mget,
##      order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##      rbind, Reduce, rownames, sapply, setdiff, sort, table, tapply,
##      union, unique, unsplit, which, which.max, which.min

## Loading required package: S4Vectors

## Loading required package: stats4

##
## Attaching package: 'S4Vectors'

## The following object is masked from 'package:base':
##
##      expand.grid

## Loading required package: IRanges

## Loading required package: XVector

##
## Attaching package: 'Biostrings'

## The following object is masked from 'package:seqinr':
##
##      translate

## The following object is masked from 'package:base':
##
##      strsplit
```

### 23.1.1   Download sequences

As we did in the previous lesson on dotplots, we'll look at two sequences.

```
# Download
## sequence 1: Q9CD83
leprae_fasta <- rentrez::entrez_fetch(db = "protein",
                         id = "Q9CD83",
                          rettype = "fasta")
## sequence 2: OIN17619.1
ulcerans_fasta <- rentrez::entrez_fetch(db = "protein",
                         id = "OIN17619.1",
                          rettype = "fasta")


# clean
leprae_vector   <- fasta_cleaner(leprae_fasta)
ulcerans_vector <- fasta_cleaner(ulcerans_fasta)
```

## 23.2 Pairwise global alignment of DNA sequences using the Needleman-Wunsch algorithm

If you are studying a particular pair of genes or proteins, an important question is to what extent the two sequences are similar.

To quantify similarity, it is necessary to **align** the two sequences, and then you can calculate a similarity score based on the alignment.

There are two types of alignment in general. A **global alignment** is an alignment of the *full* length of two sequences from beginning to end, for example, of two protein sequences or of two DNA sequences. A **local alignment** is an alignment of part of one sequence to part of another sequence; the parts the end up getting aligned are the most similar, and determined by the alignment algorithm.

The first step in computing a alignment (global or local) is to decide on a **scoring system**. For example, we may decide to give a score of +2 to a match and a penalty of -1 to a mismatch, and a penalty of -2 to a **gap** due to an **indexl**. Thus, for the alignment:

`## [1] "G A A T T C"`

`## [1] "G A T T - A"`

we would compute a score of

1. G vs G = matchb = 2
2. A vs A = match = 2
3. A vs T = mismatch = -1
4. T vs T = match = 2
5. T vs - = gap = -2
6. C vs A = mismatch = 2

So, the scores is 2 + 2 -1 + 2 -2 - 1 = 2.

Similarly, the score for the following alignment is 2 + 2 -2 + 2 + 2 -1 = 5:

`## [1] "G A A T T C"`

`## [1] "G A - T T A"`

The **scoring system** above can be represented by a **scoring matrix** (also known as a **substitution matrix**). The scoring matrix has one row and one column for each possible letter in our alphabet of letters (e.g. 4 rows and 4 columns for DNA and RNA sequences, 20 x 20 for amino acids). The (i,j) element of the matrix has a value of +2 in case of a match and -1 in case of a mismatch.

We can make a scoring matrix in R by using the `nucleotideSubstitutionMatrix()` function in the `Biostrings` package. `Biostrings` is part of a set of R packages for bioinformatics analysis known as Bioconductor (www.bioconductor.org/).

The arguments (inputs) for the `nucleotideSubstitutionMatrix()` function are the score that we want to assign to a match and the score that we want to assign to a mismatch. We can also specify that we want to use only the four letters representing the four nucleotides (ie. A, C, G, T) by setting `baseOnly=TRUE`, or whether we also want to use the letters that represent **ambiguous cases** where we are not sure what the nucleotide is (e.g. 'N' = A/C/G/T; ambiguous cases occur in some sequences due to sequencing errors or ambiguities).

To make a scoring matrix which assigns a score of +2 to a match and -1 to a mismatch, and store it in the variable `sigma`, we type:

```
# make the matrix
sigma <- nucleotideSubstitutionMatrix(match = 2,
                                      mismatch = -1,
                                      baseOnly = TRUE)
# Print out the matrix
sigma
```

```
##     A  C  G  T
## A   2 -1 -1 -1
## C  -1  2 -1 -1
## G  -1 -1  2 -1
## T  -1 -1 -1  2
```

Instead of assigning the same penalty (e.g. -8) to every gap position, it is common instead to assign a **gap opening penalty** to the first position in a gap (e.g. -8), and a smaller **gap extension penalty** to every subsequent position in the same gap.

The reason for doing this is that it is likely that adjacent gap positions were created by the same insertion or deletion event, rather than by several independent insertion or deletion events. Therefore, we don't want to penalize a 3-letter gap (AAA—AAA) as much as we would penalize three separate 1-letter gaps (AA-A-A-AA), as the 3-letter gap may have arisen due to just one insertion or deletion event, while the 3 separate 1-letter gaps probably arose due to three independent insertion or deletion events.

For example, if we want to compute the score for a global alignment of two short DNA sequences 'GAATTC' and 'GATTA', we can use the **Needleman-Wunsch** algorithm to calculate the highest-scoring alignment using a particular scoring function.

The `pairwiseAlignment()` function in the Biostrings package finds the score for the optimal global alignment between two sequences using the Needleman-Wunsch algorithm, given a particular scoring system.

As arguments (inputs), `pairwiseAlignment()` takes

1. the two sequences that you want to align,
2. the scoring matrix,
3. the gap opening penalty, and
4. the gap extension penalty.

You can also tell the function that you want to just have the optimal global alignment's score by setting `scoreOnly = TRUE`, or that you want to have both the optimal global alignment and its score by setting `scoreOnly = FALSE`.

For example, let's find the score for the optimal global alignment between the sequences 'GAATTC' and 'GATTA'.

First, we'll store the sequences as **character vectors**:

```
s1 <- "GAATTC"
s2 <- "GATTA"
```

Now we'll align them:

```
globalAligns1s2 <- Biostrings::pairwiseAlignment(s1, s2,
                                          substitutionMatrix = sigma,
                                          gapOpening = -2,
                                          gapExtension = -8,
                                          scoreOnly = FALSE)
```

The output:

```
globalAligns1s2
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: GAATTC
## subject: GA-TTA
## score: -3
```

The above commands print out the optimal global alignment for the two sequences and its score.

**Note** we set `gapOpening` to be -2 and `gapExtension` to be -8, which means that the first position of a gap is assigned a score of -8 - 2= -10, and every subsequent position in a gap is given a score of -8. Here the alignment contains four matches, one mismatch, and one gap of length 1, so its score is $(4*2)+(1*-1)+(1*-10) = -3$.

## 23.3 Pairwise global alignment of protein sequences using the Needleman-Wunsch algorithm

As well as DNA alignments, it is also possible to make alignments of protein sequences. In this case it is necessary to use a scoring matrix for amino acids rather than for nucleotides.

### 23.3.1 Protein score matrices

There are several well known scoring matrices that come with *R*, such as the **BLOSUM** series of matrices. Different BLOSUM matrices exist, named with different numbers. BLOSUM with high numbers are designed for comparing closely related sequences, while BLOSUM with low numbers are designed for comparing evolutionarily distantly related sequences. For example, **BLOSUM62** is used for **less divergent alignments** (alignments of sequences that differ little), and **BLOSUM30** is used for more divergent alignments (alignments of sequences that differ a lot).

Many *R* packages come with example data sets or data files and you use the `data()` function is used to load these data files. You can use the `data()` function to load a data set of BLOSUM matrices that comes with `Biostrings`

To load the BLOSUM50 matrix, we type:

```
data(BLOSUM50)
BLOSUM50 # Print out the data
```

```
##     A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
## A   5 -2 -1 -2 -1 -1 -1  0 -2 -1 -2 -1 -1 -3 -1  1  0 -3 -2  0 -2 -1 -1 -5
## R  -2  7 -1 -2 -4  1  0 -3  0 -4 -3  3 -2 -3 -3 -1 -1 -3 -1 -3 -1  0 -1 -5
## N  -1 -1  7  2 -2  0  0  0  1 -3 -4  0 -2 -4 -2  1  0 -4 -2 -3  4  0 -1 -5
## D  -2 -2  2  8 -4  0  2 -1 -1 -4 -4 -1 -4 -5 -1  0 -1 -5 -3 -4  5  1 -1 -5
## C  -1 -4 -2 -4 13 -3 -3 -3 -3 -2 -2 -3 -2 -2 -4 -1 -1 -5 -3 -1 -3 -3 -2 -5
## Q  -1  1  0  0 -3  7  2 -2  1 -3 -2  2  0 -4 -1  0 -1 -1 -1 -3  0  4 -1 -5
## E  -1  0  0  2 -3  2  6 -3  0 -4 -3  1 -2 -3 -1 -1 -1 -3 -2 -3  1  5 -1 -5
## G   0 -3  0 -1 -3 -2 -3  8 -2 -4 -4 -2 -3 -4 -2  0 -2 -3 -3 -4 -1 -2 -2 -5
## H  -2  0  1 -1 -3  1  0 -2 10 -4 -3  0 -1 -1 -2 -1 -2 -3  2 -4  0  0 -1 -5
## I  -1 -4 -3 -4 -2 -3 -4 -4 -4  5  2 -3  2  0 -3 -3 -1 -3 -1  4 -4 -3 -1 -5
## L  -2 -3 -4 -4 -2 -2 -3 -4 -3  2  5 -3  3  1 -4 -3 -1 -2 -1  1 -4 -3 -1 -5
## K  -1  3  0 -1 -3  2  1 -2 -3 -3 -3  6 -2 -4 -1  0 -1 -3 -2 -3  0  1 -1 -5
## M  -1 -2 -2 -4 -2  0 -2 -3 -1  2  3 -2  7  0 -3 -2 -1 -1  0  1 -3 -1 -1 -5
## F  -3 -3 -4 -5 -2 -4 -3 -4 -1  0  1 -4  0  8 -4 -3 -2  1  4 -1 -4 -4 -2 -5
## P  -1 -3 -2 -1 -4 -1 -1 -2 -2 -3 -4 -1 -3 -4 10 -1 -1 -4 -3 -3 -2 -1 -2 -5
## S   1 -1  1  0 -1  0 -1  0 -1 -3 -3  0 -2 -3 -1  5  2 -4 -2 -2  0  0 -1 -5
## T   0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  2  5 -3 -2  0  0 -1  0 -5
## W  -3 -3 -4 -5 -5 -1 -3 -3 -3 -3 -2 -3  1 -4 -4 -3 15  2 -3 -5 -2 -3 -5
```

```
## Y -2 -1 -2 -3 -3 -1 -2 -3  2 -1 -1 -2  0  4 -3 -2 -2  2  8 -1 -3 -2 -1 -5
## V  0 -3 -3 -4 -1 -3 -3 -4 -4  4  1 -3  1 -1 -3 -2  0 -3 -1  5 -4 -3 -1 -5
## B -2 -1  4  5 -3  0  1 -1  0 -4 -4  0 -3 -4 -2  0  0 -5 -3 -4  5  2 -1 -5
## Z -1  0  0  1 -3  4  5 -2  0 -3 -3  1 -1 -4 -1  0 -1 -2 -2 -3  2  5 -1 -5
## X -1 -1 -1 -1 -2 -1 -1 -2 -1 -1 -1 -1 -1 -2 -2 -1  0 -3 -1 -1 -1 -1 -1 -5
## * -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5  1
```

You can get a list of the available scoring matrices that come with the Biostrings package by using the data() function, which takes as an argument the name of the package for which you want to know the data sets that come with it:

```
data(package="Biostrings")
```

Another well-known series of scoring matrices are the **PAM** matrices developed by Margaret Dayhoff and her team. These have largely been replaced by BLOSUM but are important for historical reasons because they represent one of the first major bioinformatics, computational biology, and phyolgenetics projects ever.

## 23.3.2    Example protein alignment

Let's find the optimal global alignment between the protein sequences "PAWHEAE" and "HEAGAWGHEE" using the Needleman-Wunsch algorithm using the BLOSUM50 matrix.

First, load the scoring matrix BLOSUM50 and make vectors for the sequence

```
# matrix
data(BLOSUM50)


# sequences
s3 <- "PAWHEAE"
s4 <- "HEAGAWGHEE"
```

Now do the alignments.

```
globalAligns3s4 <- pairwiseAlignment(s3, s4,
                                     substitutionMatrix = "BLOSUM50",
                                     gapOpening = -2,
                                     gapExtension = -8,
                                     scoreOnly = FALSE)
```

Look at the results:

```
globalAligns3s4 # Print out the optimal global alignment and its score


## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: P---AWHEAE
## subject: HEAGAWGHEE
## score: -5
```

We set `gapOpening` to be -2 and `gapExtension` to be -8, which means that the first position of a gap is assigned a score of -8-2=-10, and every subsequent position in a gap is given a score of -8. This means that the gap will be given a score of -10-8-8 = -26.

## 23.4   Aligning UniProt sequences

We discussed previously how you can search for UniProt accessions and retrieve the corresponding protein sequences, either via the UniProt website or using the `rentrez` package.

In the examples given above, we learned how to retrieve the sequences for the chorismate lyase proteins from *Mycobacterium leprae* (UniProt Q9CD83) and *Mycobacterium ulcerans* (UniProt A0PQ23), and read them into R, and store them as vectors lepraeseq and ulceransseq.

You can align these sequences using `pairwiseAlignment()` from the Biostrings package.

As its input, the `pairwiseAlignment()`function requires that the sequences be in the form of a single string (e.g. "ACGTA"), rather than as a vector of characters (e.g. a vector with the first element as "A", the second element as "C", etc.). Therefore, to align the *M. leprae* and *M. ulcerans* chorismate lyase proteins, we first need to convert the vectors lepraeeq and ulceransseq into strings. We can do this using the `paste()` function:

```
# convert leprae_vector to an object lepraeseq_string
lepraeseq_string <-paste(leprae_vector,collapse = "")

# convert ulcerans_vector to an object ulceransseq_string
ulceransseq_string <-paste(ulcerans_vector,collapse = "")
```

Furthermore, pairwiseAlignment() requires that the sequences be stored as uppercase characters. Therefore, if they are not already in uppercase, we need to use the `toupper()` function to convert `lepraeseq_string` and `ulceransseq_string` to uppercase:

```
lepraeseq_string   <- toupper(lepraeseq_string)
ulceransseq_string <- toupper(ulceransseq_string)
```

Check the output

```
lepraeseq_string # Print out the content of "lepraeseq_string"
```

```
## [1] "MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLLDVAPKIPELENLKIGRILQRDILLKGQKS
```

We can now align the the *M. leprae* and *M. ulcerans* chorismate lyase protein sequences using the `pairwiseAlignment()` function:

```
globalAlignLepraeUlcerans <- Biostrings::pairwiseAlignment(lepraeseq_string,
                                              ulceransseq_string,
                                              substitutionMatrix = BLOSUM50,
                                              gapOpening = -2,
                                              gapExtension = -8,
                                              scoreOnly = FALSE)
```

The output:

```
globalAlignLepraeUlcerans # Print out the optimal global alignment and its score
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: MT-----NR--T---LSREEIRKLDRDLRILVAT...DTPREELDRCQYSNDIDTRSGDRFVLHGRVFKNL
## subject: MLAVLPEKREMTECHLSDEEIRKLNRDLRILIAT...DNSREEPIRHQRS--VGT-SA-R---SGRSICT-
## score: 627
```

As the alignment is very long, when you type `globalAlignLepraeUlcerans`, you only see the start and the end of the alignment. Therefore, we need to have a function to print out the whole alignment (see below).

## 23.5 Viewing a long pairwise alignment

If you want to view a long pairwise alignment such as that between the *M. leprae* and *M. ulerans* chorismate lyase proteins, it is convenient to print out the alignment in blocks.

The R function `print_pairwise_alignment()` below will do this for you:

```
print_pairwise_alignment(globalAlignLepraeUlcerans, 60)
```

```
## [1] "MT-----NR--T---LSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLL 50"
## [1] "MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEIVVEIVKQQIQ 60"
## [1] " "
## [1] "DVAPKIPELENLKIGRILQRDILLKGQKSGILFVAAESLIVIDLLPTAITTYLTKTHHPI 110"
## [1] "DAAPEMDGCDHSSIGRVLRRDIVLKGRRSGIPFVAAESFIAIDLLPPEIVASLLETHRPI 120"
## [1] " "
## [1] "GEIMAASRIETYKEDAQVWIGDLPCWLADYGYWDLPKRAVGRRYRIIAGGQPVIITTEYF 170"
## [1] "GEVMAASCIETFKEEAKVWAGESPAWLELDRRRNLPPKVVGRQYRVIAEGRPVIIITEYF 180"
## [1] " "
## [1] "LRSVFQDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKN 230"
## [1] "LRSVFEDNSREEPIRHQRS--VGT-SA-R---SGRSICT 233"
## [1] " "
```

The position in the protein of the amino acid that is at the end of each line of the printed alignment is shown after the end of the line. For example, the first line of the alignment above finishes at amino acid position 50 in the *M. leprae* protein and also at amino acid position 60 in the *M. ulcerans* protein. Becaues there as a difference of 60-50 = 10 bases, there must be 10 insertions in the *M.*

*leprae* to get it to line up.  Count the number of dashes in the sequence to see
how many there are.

# Chapter 24

# Local protein aligments in *R*

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 24.1   Preliminaries

```
library(compbio4all)
library(Biostrings)
```

### 24.1.1   Download sequences

As we did in the previous lesson on dotplots, we'll look at two sequences.

```
# Download
## sequence 1: Q9CD83
leprae_fasta <- rentrez::entrez_fetch(db = "protein",
                            id = "Q9CD83",
                            rettype = "fasta")
## sequence 2: OIN17619.1
ulcerans_fasta <- rentrez::entrez_fetch(db = "protein",
                            id = "OIN17619.1",
                            rettype = "fasta")

# clean
leprae_vector   <- fasta_cleaner(leprae_fasta)
ulcerans_vector <- fasta_cleaner(ulcerans_fasta)
```

```
# convert leprae_vector to an object lepraeseq_string
lepraeseq_string <-paste(leprae_vector,collapse = "")

# convert ulcerans_vector to an object ulceransseq_string
ulceransseq_string <-paste(ulcerans_vector,collapse = "")
```

## 24.2  Pairwise local alignment of protein sequences using the Smith-Waterman algorithm

You can use the `pairwiseAlignment()` function to find the optimal **local alignment** of two sequences, that is the best alignment of parts (**subsequences**) of those sequences, by using the `type=local` argument in `pairwiseAlignment()`. This uses the **Smith-Waterman algorithm** for local alignment. This is the classic bioinformatics algorithm for finding optimal local alignments. (We'll discuss updated approaches when we get into **database searches** with **BLAST**, the **Basic, Local Alignment Search Tool* that is the workhorse of many day-to-day bioinformatics tasks).

For example, to find the best local alignment between the *M. leprae* and *M. ulcerans* chorismate lyase proteins, we can run:

```
# load scoring matrix
data(BLOSUM50)

# run alignment
localAlignLepraeUlcerans <- pairwiseAlignment(lepraeseq_string,
                                              ulceransseq_string,
                                              substitutionMatrix = BLOSUM50,
                                              gapOpening = -2,
                                              gapExtension = -8,
                                              scoreOnly = FALSE,
                                              type="local")  # <= type = "local !
```

Print out the optimal local alignment and its score

```
localAlignLepraeUlcerans
```

```
## Local PairwiseAlignmentsSingleSubject (1 of 1)
## pattern:  [1] MTNRTLSREEIRKLDRDLRILVATNGTLTRVL...TEYFLRSVFQDTPREELDRCQYSNDIDTRSG
## subject: [11] MTECHLSDEEIRKLNRDLRILIATNGTLTRIL...TEYFLRSVFEDNSREEPIRHQRSVGTSARSG
## score: 761
```

As before, we can print out the full alignment with `print_pairwise_alignment()`:

```
print_pairwise_alignment(localAlignLepraeUlcerans, 60)
```

```
## [1] "MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLLDVAPKIPELE 60"
## [1] "MTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEIVVEIVKQQIQDAAPEMDGCD 60"
## [1] " "
## [1] "NLKIGRILQRDILLKGQKSGILFVAAESLIVIDLLPTAITTYLTKTHHPIGEIMAASRIE 120"
## [1] "HSSIGRVLRRDIVLKGRRSGIPFVAAESFIAIDLLPPEIVASLLETHRPIGEVMAASCIE 120"
## [1] " "
## [1] "TYKEDAQVWIGDLPCWLADYGYWDLPKRAVGRRYRIIAGGQPVIITTEYFLRSVFQDTPR 180"
## [1] "TFKEEAKVWAGESPAWLELDRRRNLPPKVVGRQYRVIAEGRPVIIITEYFLRSVFEDNSR 180"
## [1] " "
## [1] "EELDRCQYSNDIDTRSG 240"
## [1] "EEPIRHQRSVGTSARSG 240"
## [1] " "
```

We see that the optimal local alignment is quite similar to the optimal global alignment in this case, except that it excludes a short region of poorly aligned sequence at the start and at the ends of the two proteins.

# Chapter 25

# Matrix vocabulary

**By** Nathan Brouwer

## 25.1 Introduction

In this exercise we will practice manipulating matrices and vector by exploring the structure of **scoring matrices** and **alignments**.

## 25.2 Preliminaries

### 25.2.1 Packages

- BiocManager
- Biostrings
- combio4all

### 25.2.2 Vocab

#### 25.2.2.1 Math / R vocab

- triangular matrix
- square matrix
- lower triangle
- upper triangle
- symmetric matrix
- matrix diagonal

#### 25.2.2.2 Bioinformatics vocab

- scoring matrix

- BLOSUM scoring matrix

## 25.2.3   Functions used

### 25.2.3.1   Base R functions

- `data()`
- `is()`
- `nrow()`, `ncol()`,
- `dim()`,
- `names()`, `colnames()`, `rownames()`,
- `head()`, `tail()`

### 25.2.3.2   Specific function

- combio4all::tri_print()
- combio4all::diag_show()

## 25.3   Load packages

Load basic packages

```
library(flextable)
```

```
##
## Attaching package: 'flextable'

## The following object is masked from 'package:Biostrings':
##
##     width

## The following object is masked from 'package:XVector':
##
##     width

## The following object is masked from 'package:IRanges':
##
##     width

## The following object is masked from 'package:S4Vectors':
##
##     width

## The following object is masked from 'package:BiocGenerics':
##
##     width
```

```
library(webshot)
```

```
##
## Attaching package: 'webshot'

## The following object is masked from 'package:IRanges':
##
##      resize
```

```
library(compbio4all)
```

Data

```
data(BLOSUM62)
```

Install the Biostrings package from Bioconductor; you don't have to run this if
you already happen to have downloaded Biostrings before

In the *R* console the matrix looks like this.

```
BLOSUM62
```

```
##    A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
## A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4
## R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1 -2  0 -1 -4
## N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  4 -3  0 -1 -4
## D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4 -3  1 -1 -4
## C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -1 -3 -1 -4
## Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0 -2  4 -1 -4
## E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1 -3  4 -1 -4
## G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -4 -2 -1 -4
## H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0 -3  0 -1 -4
## I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3  3 -3 -1 -4
## L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4  3 -3 -1 -4
## K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0 -3  1 -1 -4
## M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3  2 -1 -1 -4
## F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3  0 -3 -1 -4
## P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -3 -1 -1 -4
## S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0 -2  0 -1 -4
## T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -1 -4
## W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -2 -2 -1 -4
## Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -1 -2 -1 -4
## V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3  2 -2 -1 -4
## B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4 -3  0 -1 -4
## J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2 -3  3 -3 -1 -4
## Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2  0 -3  4 -1 -4
## X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -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  1
```

The function `tri_print()` will print just the lower triangle, with all the other
values left empty. (This might be a bit slow)

```
tri_print(BLOSUM62, as.image = T)
```

| x | A | R | N | D | C | Q | E |
|---|---|---|---|---|---|---|---|
| A | 4 | | | | | | |
| R | -1 | 5 | | | | | |
| N | -2 | 0 | 6 | | | | |
| D | -2 | -2 | 1 | 6 | | | |
| C | 0 | -3 | -3 | -3 | 9 | | |
| Q | -1 | 1 | 0 | 0 | -3 | 5 | |
| E | -1 | 0 | 0 | 2 | -4 | 2 | 5 |
| G | 0 | -2 | 0 | -1 | -3 | -2 | -2 |
| H | -2 | 0 | 1 | -1 | -3 | 0 | 0 |
| I | -1 | -3 | -3 | -3 | -1 | -3 | -3 |
| L | -1 | -2 | -3 | -4 | -1 | -2 | -3 |
| K | -1 | 2 | 0 | -1 | -3 | 1 | 1 |
| M | -1 | -1 | -2 | -3 | -1 | 0 | -2 |
| F | -2 | -3 | -3 | -3 | -2 | -3 | -3 |
| P | -1 | -2 | -2 | -1 | -3 | -1 | -1 |
| S | 1 | -1 | 1 | 0 | -1 | 0 | 0 |
| T | 0 | -1 | 0 | -1 | -1 | -1 | -1 |
| W | -3 | -3 | -4 | -4 | -2 | -2 | -3 |
| Y | -2 | -2 | -2 | -3 | -2 | -1 | -2 |
| V | 0 | -3 | -3 | -3 | -1 | -2 | -2 |
| B | -2 | -1 | 4 | 4 | -3 | 0 | 1 |
| J | -1 | -2 | -3 | -3 | -1 | -2 | -3 |
| Z | -1 | 0 | 0 | 1 | -3 | 4 | 4 |
| X | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| * | -4 | -4 | -4 | -4 | -4 | -4 | -4 |

We can also look at the upper triangle.

```
tri_print(BLOSUM62,triangle = "upper", as.image = T)
```

| x | A | R | N | D | C | Q | E | G |
|---|---|---|---|---|---|---|---|---|
| A | 4 | -1 | -2 | -2 | 0 | -1 | -1 | 0 |
| R |   | 5 | 0 | -2 | -3 | 1 | 0 | -2 |
| N |   |   | 6 | 1 | -3 | 0 | 0 | 0 |
| D |   |   |   | 6 | -3 | 0 | 2 | -1 |
| C |   |   |   |   | 9 | -3 | -4 | -3 |
| Q |   |   |   |   |   | 5 | 2 | -2 |
| E |   |   |   |   |   |   | 5 | -2 |
| G |   |   |   |   |   |   |   | 6 |
| H |   |   |   |   |   |   |   |   |
| I |   |   |   |   |   |   |   |   |
| L |   |   |   |   |   |   |   |   |
| K |   |   |   |   |   |   |   |   |
| M |   |   |   |   |   |   |   |   |
| F |   |   |   |   |   |   |   |   |
| P |   |   |   |   |   |   |   |   |
| S |   |   |   |   |   |   |   |   |
| T |   |   |   |   |   |   |   |   |
| W |   |   |   |   |   |   |   |   |
| Y |   |   |   |   |   |   |   |   |
| V |   |   |   |   |   |   |   |   |
| B |   |   |   |   |   |   |   |   |
| J |   |   |   |   |   |   |   |   |
| Z |   |   |   |   |   |   |   |   |
| X |   |   |   |   |   |   |   |   |
| * |   |   |   |   |   |   |   |   |

## 25.4   Matrix structure

The BLOSUM62 matrix is a **square matrix** since the number of rows equals
the number of columns. It is also a symmetrical matrix since the **lower triangle**
of the matrix is the same as the **upper triangle**. The `compbio4all` package
has functions for displaying these.

*R* shows us the full **symmetric matrix**, though in books usually they just
show the **lower triangle**. A symmetric matrix is one where the upper and
lower triangles are identical.

## 25.5   Diagonal of a matrix

The **diagonal of a matrix** is the set of numbers that falls directly in a diagonal
line from the upper left corner to the lower right.

The diagonal of a scoring matrix represents an amino acid that hasn't changed
or diverged between two sequences (in theory it could change and then change
back). We can access the diagonal in *R* `diag()`.

What does `diag(BLOSUM62)` tell you?

```
diag(BLOSUM62)
```

```
## A R N D C Q E G H I L K M F P S T W Y V B J Z X *
## 4 5 6 6 9 5 5 6 8 4 4 5 5 6 7 4 5 11 7 4 4 3 4 -1 1
```

This is showing you the entries that fall along the diagonal of the matrix. But
the output is just a string of numbers. We can look at the diagonal in context
using the `diag_show()` function from the compbio4all package.

```
diag_show(BLOSUM62)
```

| .  | A | R | N | D | C | Q | E |
|----|---|---|---|---|---|---|---|
| A  | 4 |   |   |   |   |   |   |
| R  |   | 5 |   |   |   |   |   |
| N  |   |   | 6 |   |   |   |   |
| D  |   |   |   | 6 |   |   |   |
| C  |   |   |   |   | 9 |   |   |
| Q  |   |   |   |   |   | 5 |   |
| E  |   |   |   |   |   |   | 5 |
| G  |   |   |   |   |   |   |   |
| H  |   |   |   |   |   |   |   |

| . | A | R | N | D | C | Q | E | G |
|---|---|---|---|---|---|---|---|---|
| I | | | | | | | | |
| L | | | | | | | | |
| K | | | | | | | | |
| M | | | | | | | | |
| F | | | | | | | | |
| P | | | | | | | | |
| S | | | | | | | | |
| T | | | | | | | | |
| W | | | | | | | | |
| Y | | | | | | | | |
| V | | | | | | | | |
| B | | | | | | | | |
| J | | | | | | | | |
| Z | | | | | | | | |
| X | | | | | | | | |
| * | | | | | | | | |

# Chapter 26

# Alignment in R

**By**: Nathan Brouwer

## 26.1 Preliminaries

### 26.1.1 Packages

```
library(Biostrings)
data(BLOSUM62)
```

### 26.1.2 Data

```
hShrm3  <- "EPEREPEWRDRPGSP"

mShrm3   <- "EAEREASWSEDRPGT"
```

We can do a pairwise alignment directly in *R* using `pairwiseAlignment()` from Biostrings, a package from Bioconductor. We'll use this function to check the math on our alignment calculations from the previous lesson.

Normally this `pairwiseAlignment()` will insert gaps. Alignment algorithms allow you to set a **penalty** for first creating a gap, and for how large it is. As noted above, usually the penalty is largest for first allowing a gap initially, but less for each additional space. So a gap of 1 might have a penalty of -14 (-10 + -4), but a gap of 2 might have total penalty of -18 (-10 + -4 + -4).

In `pairwiseAlignment()` the arguments related to gaps are `gapOpening = ` and `gapExtension =`, where `gapOpening` is the for starting the gap and `gapExtension` is the penalty for the size of the gap (*including* the first insertion).

In *R* you can get information about a function using the ?    function.
If we want to know about the `pairwiseAlignment()` function we run
`?pairwiseAlignment`.

Call up the help file for `pairwiseAlignment` and see if you can see what the
**defaults** are for gap opening (aka gap creation) and gap extension. Hint: Its
int the first 50 lines of the help file - don't scroll too far down).

*R* help files are pretty dense so this might be hard. Note that the penalties are
set as positive numbers in the help file but get converted to negative values.

```r
# Call up the help file for pairwiseAlignment()


# What is the gap opening penalty? Replace the "NA" with the penalty

gapOpen <- NA


# What is the gap extension ? Replace the "NA" with the penalty

gapExtend <- NA
```

We can force the alignment to NOT allow gaps by setting the penalties for gaps
to be REALLY large.

```r
nogap <- pairwiseAlignment(hShrm3,
                  mShrm3,
                  gapOpening = -100,
                  gapExtension = -100,
                  substitutionMatrix = "BLOSUM62")
```

We can check the alignment by calling up the object

We can get the score directly using the score() function

We can get the percent identity using the pid() function

Note that In the code above I've set the penalties to be negative, which is more
intuitive. Confirm that positive numbers for these penalties result in the same
score by changing the 100 values to -100. Copy the code from above and make
the appropriate changes.

Why does setting the gap penalties high make the algorithm *not* insert gaps?
We haven't covered this directly yet, but the goal of an alignment algorithm
is to maximize the score by lining up the two sequences and adding gaps and
deletions as necessary. Recalling what's along the diagonal of the BLOSUM
matrix, direct matches (identity) between the residues on two sequences result
in the highest scores, and differences have lower scores based on how different
the residues are. Look at the BLOSUM matrix and see if there are any entries
that are as low as -10, the gap penalty.

We can allow gaps by removing the gapOpening and gapExtension arguments (deleting them). Copy the code from above, remove those arguments, and assign the output to an object called gap.

When we don't assign anything to gapOpening and gapExtension arguments in the code above, the function uses the defaults. We can also specify the gap penalties directly. COpy the code again, leave the arguments in but change the penalties to -10 for gap creation and -4 for gap extension.

Look at the alignment; How many indels were added on the "pattern" strand?

What is the score of this alignment? Use the appropriate function to access it directly and save into an object called score.gap

```
# remove the NA and add the appropriate code
score.gap <- NA
```

What is the percent identity? Use the appropriate function to access it directly and save it to and object called pid.gap

```
# remove the NA and add the appropriate code
pid.gap <- NA
```

# Chapter 27

# Manipulating matrices and vectors: worked example

**By** Nathan Brouwer

## 27.1  Introduction

In this exercise we will practice manipulating matrices and vectors by exploring the structure of **scoring matrices**.

## 27.2  Preliminaries

### 27.2.1  Packages

- BiocManager
- Biostrings
- combio4all

### 27.2.2  Vocab

#### 27.2.2.1  Math / R vocab

- triangular matrix
- square matrix
- lower triangle
- upper triangle
- symmetric matrix
- matrix diagonal
- named vectors
- named matrices

- accessing items in named vectors or matrices
- square bracket notation

#### 27.2.2.2 Bioinformatics vocab

- scoring matrix
- BLOSUM scoring matrix

### 27.2.3 Functions used

#### 27.2.3.1 Base R functions

- `data()`
- `is()`
- `nrow()`, `ncol()`,
- `dim()`,
- `names()`, `colnames()`, `rownames()`,
- `head()`, `tail()`

### 27.2.4 Packages

```
library(Biostrings)
```

### 27.2.5 Data

```
data(BLOSUM62)
```

## 27.3 Matrix elements can be accessed with square bracket notation

We often want to access just subsets of data from a matrix or a dataframe. This can take some getting used to. We can use **square brackets** to get certain subsets or ranges of cells. If we want just the upper left-hand cell we can do this:

```
BLOSUM62[1,1]
```

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

If we want the first four cells in the upper left hand corner we can do this

```
BLOSUM62[c(1:4), c(1:4)]
```

```
##    A  R  N  D
## A  4 -1 -2 -2
## R -1  5  0 -2
```

```
## N -2  0  6  1
## D -2 -2  1  6
```

If we want to get rid of the **ambiguity code** cells on the bottom which are B, J, Z, X and * we can specify that we want elements 1 through 20. Let me step through this.

First, run the code below; what happens?

```
1:4
```

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

Now do the same thing for 1 to 20:

Note that this next line should provide the same result as what you just did.

```
c(1:20)
```

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

If we *really* wanted to type all of this out, we would have to do this:

```
c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
```

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

We can get just the first row of the BLOSUM matrix by telling *R* we want row 1 and columns 1 to 20.

We get just *row* 1 like this:

```
BLOSUM62[1, ]
```

```
##  A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
##  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4
```

We get columns 1:20 like this:

```
BLOSUM62[ , c(1:20)]
```

```
##    A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
## A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0
## R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3
## N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3
## D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3
## C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1
## Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2
## E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2
## G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3
## H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3
## I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3
## L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1
## K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2
```

```
## M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1
## F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1
## P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2
## S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2
## T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0
## W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3
## Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1
## V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4
## B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3
## J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2
## Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2
## X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
## * -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4
```

Note that leaving the first part blank gives us ALL the rows.

If we want just the first row AND the first 20 columns, we do this:

```
BLOSUM62[1 , c(1:20)]
```

```
##  A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
##  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0
```

Assign the output of the code `c(1:20)` to an object called `i` using the assignment operator `<-`. The code should like like this `i <- c(1:20)`.

```
i <- c(1:20)
```

This thing `i` we just made is a **vector** of numbers. Vectors are a 1-dimensional sequence of numbers. You can think of a matrix as a bunch of vectors stacked on top of each other.

Unfortunately, if you call `is()` in the `i` object you don't get a totally clear picture of what it is. Where does the word `vector` show up?

I'm not sure why `vector` isn't the first thing to be printed. We can check whether `i` is a vector more directly by asking it "hey i, are you a vector" using the `is.vector()` command

```
is.vector(i)
```

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

Its a vector, so it shouldn't be a matrix, but we can check with `is.matrix()`

```
is.matrix(i)
```

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

Math books may tell you that a vector is a 1-dimensional matrix, but in *R* land vectors are distinct from matrices.

Vectors show up *everywhere* in R. In this case I've defined a vector, `i`, which is holding the row and column numbers I want to isolated from the BLOSUM62 matrix.

I can call up just these rows and columns like this

```
BLOSUM62[i , i]
```

```
##    A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
## A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0
## R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3
## N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3
## D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3
## C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1
## Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2
## E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2
## G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3
## H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3
## I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3
## L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1
## K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2
## M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1
## F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1
## P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2
## S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2
## T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0
## W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3
## Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1
## V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4
```

If I hadn't defined the vector `i`, I could write

```
BLOSUM62[c(1:20) ,c(1:20)  ]
```

```
##    A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
## A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0
## R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3
## N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3
## D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3
## C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1
## Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2
## E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2
## G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3
## H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3
## I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3
## L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1
## K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2
## M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1
```

```
## F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1
## P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2
## S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2
## T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0
## W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3
## Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1
## V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4
```

Let's isolate just the first 20 rows and columns and put them into a new object called BLOSUM62.subset.

```
BLOSUM62.subset <-  BLOSUM62[i ,i ]
```

We can get the scores for when there is no change in an amino acid from the diagonal with diag()

```
diag(BLOSUM62.subset)
```

```
##  A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
##  4  5  6  6  9  5  5  6  8  4  4  5  5  6  7  4  5 11  7  4
```

If you're scoring an alignment by hand you can pull up the diagonal of the matrix this way so you don't have to squint at the whole matrix. We can make it even easier if we alphabetize thing, though this requires some extra code which you don't need to worry about. (I'm making a new vector, i2, which will do the alphabetizing).

```
# get column names
n <- colnames(BLOSUM62.subset)

# sort
i2 <- sort(n)
```

Now things are in alphabetical order

```
diag(BLOSUM62.subset)[i2]
```

```
##  A  C  D  E  F  G  H  I  K  L  M  N  P  Q  R  S  T  V  W  Y
##  4  9  6  5  6  6  8  4  5  4  5  6  7  5  5  4  5  4 11  7
```

Save this alphabetized diagonal to an R object called BLOSUM62.diag using the assignment operator <-

```
BLOSUM62.diag <- diag(BLOSUM62.subset)[i2]
```

What are the names attached to this BLOSUM62.diag object? Unfortunately *R* is picky about how you do this so you need to figure out whether the functions names(), rownames(), or colnames() gets you want you want

```
rownames(BLOSUM62.diag)
```

```
## NULL
```

```
colnames(BLOSUM62.diag)
```

```
## NULL
```

```
names(BLOSUM62.diag)
```

```
##  [1] "A" "C" "D" "E" "F" "G" "H" "I" "K" "L" "M" "N" "P" "Q" "R" "S" "T" "V" "W"
## [20] "Y"
```

## 27.4  R object names

When an *R* object has "names" assigned to it we can use the name to call up elements of the object.

### 27.4.1  Accessing items in names vectors

We can call up just the score for an E to E transition stored in our matrix diagonal like this, which is a way of saying "Hey R, give me the value in this `BLOSUM62.diag` object that is in the slot labeled"E".

```
BLOSUM62.diag["E"]
```

```
## E
## 5
```

E is the 4th slot so we can also get this value like this with the slot number.

```
BLOSUM62.diag[4]
```

```
## E
## 5
```

We could get the first four values using `c(1:4)`. Try it

What type of object is this diagonal thingy anyway? The command starts with "i".

### 27.4.2  Accessing items in names matrices

If we want to get something from the main matrix (the full BLOSUM matrix, not the diagonal) we can also specify things using the row and column names. This will give us the value for an E to E transition:

```
BLOSUM62.subset["E","E"]
```

```
## [1] 5
```

How could you get the whole "E" *column*?

How could you get the whole "E" *row*?

We can use numbers too if we want. "E" is in the 7th column, so we can get the E to E value like this:

```
BLOSUM62.subset[7,7]
```

```
## [1] 5
```

How would you get the whole "E" column?

How would you get the whole "E" row?

```
BLOSUM62.subset[7, ]
```

```
##  A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
## -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2
```

We can of course specify any score we want. For a P to A transition we could do this:

```
BLOSUM62.subset["P","A"]
```

```
## [1] -1
```

Or specify the row and column numbers

```
BLOSUM62.subset[15,1]
```

```
## [1] -1
```

What happens when you do ["A","P"] instead (reverse of what was above?

So, if you are doing an alignment by hand, you can quickly query the matrix and pull up the scores. If you have a sequence "EPEERPEWRDRPGSP" and "EAEREASWSEDRPGT" you can get the score for the E to E matching again like this

```
BLOSUM62.subset["E","E"]
```

```
## [1] 5
```

and P to A like this

```
BLOSUM62.subset["P","A"]
```

```
## [1] -1
```

and so on.

# Chapter 28

# Alignment by eye in R

**By** Nathan Brouwer

## 28.1 Preliminaries

### 28.1.1 Packages

```
library(compbio4all)
library(Biostrings)
```

### 28.1.2 Data

```
data("BLOSUM62")
```

Let's do an alignment between two parts of a shroom protein. We'll look at part of the ASD2 domain of Shroom 3. We'll look at human shroom (hShrm3) and call this the "Query" sequence. We'll use mouse shroom (mShrm3) as the "subject" or "target". (The accession number for human shroom 3 is NP_065910.3)

```
hShrm3   <- "EPEREPEWRDRPGSP"

mShrm3    <- "EAEREASWSEDRPGT"
```

Use nchar() to see how many characters are in them

We can make a little matrix and look at how they align using the rbind() function, which stands for "row bind"

```
rbind(hShrm3,
      mShrm3)
```

```
##          [,1]
```

```
## hShrm3 "EPEREPEWRDRPGSP"
## mShrm3 "EAEREASWSEDRPGT"
```

Assign this to an object called shrms

```
shrms <- rbind(hShrm3, mShrm3)
```

What type of object is shrms?

How big is it? See if you can guess first before using R to check

If we want to do an alignment by eye it would help if instead of having all the letters stuck together they were separated. We would do this by hand but it would require a lot of typing:

```
hShrm3_alt <- c("E","P","E","R","E","P","E","W","R","D","R","P","G","S","P")

mShrm3_alt <- c("E","A","E","R","E","A","S","W","S","E","D","R","P","G","T")
```

One thing that is annoying about R is that vectors have "length" but no dimension. Compare the output of the length() command and dim() command when called on hShrm3_alt

```
# run length()


# run dim()
```

Use is, is.vector, length, is.character, and n.char on hShrm3 and hShrm3_alt. How is hShrm3 ("EPEREPEWRDRPGSP") different from hShrm3_alt? Can you figure out why nchar() is doing what its doing?

```
# run is() on hShrm3 AND hShrm3_alt


# run is.vector on hShrm3 AND hShrm3_alt


# do this for the rest of the commands
```

Typing all this stuff out is a pain. Instead of typing all those commas to make hShrm3_alt we could take hShrm3 and have *R* do the work for us. Lots of people work with **character data** (data make up of alphabetic and other characters that can NOT be interpreted as numbers) so there are lots of functions for manipulating characters. We'll use the `strsplit()` function.

If we just all strsplit() on the hShrm3 object we get something a little annoying.

```
strsplit(hShrm3, split = "")
```

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

```
##  [1] "E" "P" "E" "R" "E" "P" "E" "W" "R" "D" "R" "P" "G" "S" "P"
```

Note that the output above is on too lines, and the first line is `[[1]]`.

Ok, what's up?

Assign the thing we just made to an object called hShrm3_vec for "hShrm3 vector".

```
hShrm3_vec <- strsplit(hShrm3, split = "")
```

Now figure out what the heck it "is":

If you used the right command to figure out what it **is** (hint) the first thing you see is "list." Lists are rather complex **data structures** in R because they aren't the kind of thing you run in to if you just have experience working in a spreadsheet. Lists allow you to make collections of different R objects. For now you just need to know that they exist; we'll run into them again later. For now, we want to get rid of this listy-ness of our R object. We can do this with the function unlist(). Call unlist() on your hShrm3_vec object and re-assign it to the same object hShrm3_vec, eg, run hShrm3_vec <- unlist(hShrm3_vec)

```
hShrm3_vec <- unlist(hShrm3_vec)
```

Now we need to repeat these steps for the mouse shroom sequence.

```
# split up mShrm3 with strsplit()
mShrm3_vec <- strsplit(mShrm3, split = "")

# unlist it with unlist()
mShrm3_vec <- unlist(mShrm3_vec)
```

Something that is useful once you get good at reading R code is that you can wrap functions within functions. So instead of doing this in separate steps I could just do this:

```
hShrm3_vec <- unlist(strsplit(hShrm3, split = ""))
mShrm3_vec <- unlist(strsplit(mShrm3, split = ""))
```

Before moving on confirm that these two factors are the same length. If they aren't R will get angry

```
# length of hShrm3_vec

# length of mShrm3_vec
```

Now let's make a little matrix so we can think about aligning these sequences. The rbind() function binds two rows together into a matrix.

```
rbind(hShrm3_vec, mShrm3_vec)
```

```
##            [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
```

```
## hShrm3_vec "E"   "P"   "E"   "R"   "E"   "P"   "E"   "W"   "R"   "D"    "R"    "P"    "G"
## mShrm3_vec "E"   "A"   "E"   "R"   "E"   "A"   "S"   "W"   "S"   "E"    "D"    "R"    "P"
##               [,14] [,15]
## hShrm3_vec "S"   "P"
## mShrm3_vec "G"   "T"
```

For what we want to do next aligning them vertically will actually work better.
We can align them as columns using cbind(), which stands for "column bind"

```
cbind(hShrm3_vec, mShrm3_vec)
```

```
##         hShrm3_vec mShrm3_vec
##  [1,] "E"        "E"
##  [2,] "P"        "A"
##  [3,] "E"        "E"
##  [4,] "R"        "R"
##  [5,] "E"        "E"
##  [6,] "P"        "A"
##  [7,] "E"        "S"
##  [8,] "W"        "W"
##  [9,] "R"        "S"
## [10,] "D"        "E"
## [11,] "R"        "D"
## [12,] "P"        "R"
## [13,] "G"        "P"
## [14,] "S"        "G"
## [15,] "P"        "T"
```

Save that to an object called shrm

```
cbind(hShrm3_vec, mShrm3_vec)
```

```
##         hShrm3_vec mShrm3_vec
##  [1,] "E"        "E"
##  [2,] "P"        "A"
##  [3,] "E"        "E"
##  [4,] "R"        "R"
##  [5,] "E"        "E"
##  [6,] "P"        "A"
##  [7,] "E"        "S"
##  [8,] "W"        "W"
##  [9,] "R"        "S"
## [10,] "D"        "E"
## [11,] "R"        "D"
## [12,] "P"        "R"
## [13,] "G"        "P"
## [14,] "S"        "G"
## [15,] "P"        "T"
```

```
shrm <- cbind(hShrm3_vec, mShrm3_vec)
```

The most basic way to do an alignment is to determine the which bases are identical and to score those as 1, and anything that is mismatched score as 0. From this you can determine **percent identity**.

A handy way to do this quickly in R is to use the ifelse() command. What you can do is tell it to do this: "IF a base in the first sequence the same as the aligned base in the second sequence, return a value of 1, ELSE return a value of 0."

I'll do a really transparent example. First, let me make some very simple objects with just a single letter in them.

```
aa1 <- "A"
aa2 <- "A"
aa3 <- "W"
```

What kind of object have I just made? Use is, is.vector, length, dim, and is.matrix. Can you understand why you get the results that you do?

Now let me use ifelse(). First, let me check if aa1 is the same as aa2, an if they R, return a value of 1, else return a value of 0

```
ifelse(aa1 == aa2, yes =  1, no = 0)
```

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

The first part of the function asks a true/false question: is aa1 the same as aa2? Note that it is TWO equals signs

This is a **logical comparison** or **logical test** in R. It would work on its own outside of ifelse, like this:

```
aa1 == aa2
```

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

Now compare aa1 and aa2. First try the logical test

Now try the ifelse() command

Something that is very hand in R is that it can process things in a series. First, let's turn our single amino acids into some sequences.

```
seq1 <- c(aa1, aa2, aa3)
seq2 <- c(aa3, aa2, aa1)
```

Do you know what kind of R object we just made? Run an appropriate check

Now let's compare these two sequences. First a logical comparison. Can you tell what this code is doing?

```
seq1 == seq2
```

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

What is going on here?  It might help to line everything up using rbind() to make a little matrix.  First, save the output of the logical comparison to an R object

```
identical <- seq1==seq2
```

Now stack the two sequences and the logical comparison into a matrix

```
rbind(seq1, seq2, identical)
```

```
##            [,1]    [,2]   [,3]
## seq1      "A"     "A"    "W"
## seq2      "W"     "A"    "A"
## identical "FALSE" "TRUE" "FALSE"
```

We can also run ifelse() on the two sequences

```
ifelse(seq1 == seq2, yes = 1, no = 0)
```

```
## [1] 0 1 0
```

Save this output to an object called align.score for "alignment score"

```
align.score <- ifelse(seq1 == seq2, yes = 1, no = 0)
```

Now stack everything up into a matrix using rbind

```
rbind(seq1, seq2, identical, align.score )
```

```
##             [,1]    [,2]   [,3]
## seq1        "A"     "A"    "W"
## seq2        "W"     "A"    "A"
## identical   "FALSE" "TRUE" "FALSE"
## align.score "0"     "1"    "0"
```

Now let's try this on our shroom vectors.  First, do a logical comparison of the hShrm3_vec vector and the mShrm3_vec vector.  Remember there are two equals signs (==)

Now use ifelse() to assign a 1 to a match and a 0 to a mismatch.

Again, can you describe what's going on here?

Again we can make a little matrix with rbind().  I'm going to embed the ifelse() function within rbind(); this might be a little dense but see if you can figure out what each separate command is.

```
rbind(hShrm3_vec,
      mShrm3_vec,
```

```
    identical = hShrm3_vec == mShrm3_vec,
    score= ifelse(hShrm3_vec == mShrm3_vec,yes = 1, no = 0 ))
```

```
##              [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]    [,9]
## hShrm3_vec "E"     "P"     "E"     "R"     "E"     "P"     "E"     "W"     "R"
## mShrm3_vec "E"     "A"     "E"     "R"     "E"     "A"     "S"     "W"     "S"
## identical  "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "FALSE" "FALSE" "TRUE"  "FALSE"
## score      "1"     "0"     "1"     "1"     "1"     "0"     "0"     "1"     "0"
##             [,10]   [,11]   [,12]   [,13]   [,14]   [,15]
## hShrm3_vec "D"     "R"     "P"     "G"     "S"     "P"
## mShrm3_vec "E"     "D"     "R"     "P"     "G"     "T"
## identical  "FALSE" "FALSE" "FALSE" "FALSE" "FALSE" "FALSE"
## score      "0"     "0"     "0"     "0"     "0"     "0"
```

If we want the total score we can assign the results of ifelse() to an object

```
scores <-ifelse(hShrm3_vec == mShrm3_vec,yes = 1, no = 0 )
```

I can then total up the score with sum()

```
sum(scores)
```

```
## [1] 5
```

I can easily call up the number of amino acids with length(). Do that below

Percent identity is a common statistic when comparing sequences. If my score is 5 and my total number of residues is 15 my percent identity is:

```
5/15
```

```
## [1] 0.3333333
```

I can do this directly on the objects like this

```
sum(scores)/length(scores)
```

```
## [1] 0.3333333
```

We can do this sort of scoring on sequences in a matrix or data frame. Remember that we made a matrix called shrm with our two sequences in it. Matrices and dataframes are very similar, but its usually easier to work with dataframes. We can convert the matrix to a dataframe like this

```
shrm <- data.frame(shrm, stringsAsFactors = F)
```

There's a lot relate to the "stringsAsFactors = F"; basically it makes sure we are working with raw character data.

Calling summary() on the shrm dataframe can confirm that its character data:

```
summary(shrm)
```

```
##   hShrm3_vec          mShrm3_vec
##  Length:15           Length:15
##  Class :character    Class :character
##  Mode  :character    Mode  :character
```

You can call up the human shroom 3 columns with this code:  shrm[, "hShrm3_vec"]. try it below

How would you call up the other column with mouse shroom?

One tricky thin about R is that you can access the column of dataframes in more than one way. You can get the first column also using the dollar sign

```
shrm$hShrm3_vec
```

```
## [1] "E" "P" "E" "R" "E" "P" "E" "W" "R" "D" "R" "P" "G" "S" "P"
```

Note that even though we're calling a column, R prints it out left to right like a row.

We can do a logical comparisons of the two columns like this:

```
shrm[, "hShrm3_vec"] == shrm[,"mShrm3_vec"]
```

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

Can you re-write this using the dollar sign notation? Try it below

Now let's use ifelse() on these two columns

```
ifelse(shrm[, "hShrm3_vec"] == shrm[,"mShrm3_vec"], yes = 1, no = 0)
```

```
## [1] 1 0 1 1 1 0 0 1 0 0 0 0 0 0 0
```

Rewrite the code above using dollar sign notation

We can add this information to the dataframe like this

```
shrm$identical <- ifelse(shrm[, "hShrm3_vec"] == shrm[,"mShrm3_vec"], yes = 1, no = 0)
```

Now what do we have? Call up the dataframe shrm we just made:

Now call summary on it. What types of data are each variable?

Comparing an alignment to a scoring matrix by hand is a lot of work, so there's a function called `score_alignment()` in the `compbio4all` package which can do basic comparisons. The function has the following arguments

- seq.df = A dataframe with aligned sequences in columns
- seq1 = the name of the column with a sequence in it; name must be in quotes

- seq2 = the other, aligned sequence
- gap.penalty = -10

Running the function returns a dataframe with a new column called "score".

```
shrm <-score_alignment(seq.df = shrm,
                seq1=  "hShrm3_vec",  #note the quotation marks
                seq2 = "mShrm3_vec",
                sub.mat = BLOSUM62,
                gap.penalty =  -10)
```

We can call the sum() function on the score column of the shrm dataframe to calculate the overall score for the alignment. This is a metric that tells use how good the alignment is based on the number of identical residues in the sequence, how different any mutations are from the original residue, and how many gaps there are.

```
sum(shrm$score)
```

```
## [1] 23
```

I happen to know that we can make this alignment better if we add a gap. I'll do it by hand because I don't have a function yet to do it automatically. I'll put a gap into the human shroom 3 sequence (hShrm3_alt) after the "W" that's about in the middle of the sequence.

```
hShrm3_alt  <- c("E","P","E","R","E","P","E","W","-","R","D","R","P","G","S","P")
```

Using an appropriate function, confirm that the insertion has made the sequence longer (hint: its not nchar)

Now we'll do the other sequence. To make this work we need to add an insertion to the end of the mouse sequence (mShrm3_alt) so that the two sequences are the same length.

```
mShrm3_alt  <- c("E","A","E","R","E","A","S","W","S","E","D","R","P","G","T","-")
```

Again, check that this sequence is longer than it was before

A handy trick is to do a **logical comparison** to check that the two vectors are the same. Run the following code and see if you can interpret what is going on. Remember that the double equals sign == carries out a logical comparison that returns TRUE if two values are the same.

```
length(hShrm3_alt) == length(mShrm3_alt)
```

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

Now, for a challenge, see if you can figure out what's going on here

```
hShrm3_alt == mShrm3_alt
```

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

```
## [13]  TRUE  TRUE FALSE FALSE
```

It might be easier if we stack thing in to a little matrix with rbind(). I've put a lot of stuff in the rbind() function but see if you can figure it out.

```r
rbind(hShrm3_alt = hShrm3_alt,
      mShrm3_alt = mShrm3_alt,
      identical = hShrm3_alt == mShrm3_alt)
```

```
##              [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]    [,9]
## hShrm3_alt "E"     "P"     "E"     "R"     "E"     "P"     "E"     "W"     "-"
## mShrm3_alt "E"     "A"     "E"     "R"     "E"     "A"     "S"     "W"     "S"
## identical  "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "FALSE" "FALSE" "TRUE"  "FALSE"
##              [,10]   [,11]   [,12]   [,13]   [,14]   [,15]   [,16]
## hShrm3_alt "R"     "D"     "R"     "P"     "G"     "S"     "P"
## mShrm3_alt "E"     "D"     "R"     "P"     "G"     "T"     "-"
## identical  "FALSE" "TRUE"  "TRUE"  "TRUE"  "TRUE"  "FALSE" "FALSE"
```

Now let's compare these two sequences by scoring them. First, we'll put them into a dataframe using data.frame(). We'll call the object shrm.gap.

```r
shrm.gap <-data.frame(hShrm3_alt,
                      mShrm3_alt,
                      stringsAsFactors = F)
```

Run an appropriate function to confirm that this is a dataframe

Check what the size of the dataframe is:

Now look at the full dataframe

Now run a function to just look at the top of the dataframe:

Run a function to just look at the bottom of the dataframe

Ok, we have a sense of what we're working with. Let's calculate the score

```r
shrm.gap <- score_alignment(shrm.gap,
                      seq1=  "hShrm3_alt",
                      seq2 = "mShrm3_alt",
                      sub.mat= BLOSUM62,
                      gap.penalty =-10)
```

Look at the dataframe and make sure it looks right. Note that each gap ("-") has a score of 10 as defined by the gap.penalty.

Now calculate the total score

```r
sum(shrm.gap$score)
```

```
## [1] 34
```

There is one issue here which I haven't yet resolved. When alignment are scored we take into account two things: the creation of a gap (yes there is a gap there) and its length (how long it it).

Each gap that get's created gets scores -10 for occurring and -4 for each insertion in the gap. So a gap of 1 insertion ("-") gets score -10 + -4

```
-10 + -4
```

```
## [1] -14
```

A gap of 2 insertions ("–") gets a score of

```
(-10 + -4) + -4
```

```
## [1] -18
```

A gap of 3 insertions ("—") gets a score of

```
(-10 + -4) + -4 + -4
```

```
## [1] -22
```

Our alignment has two separate insertions, one in the middle and one on the end. In the logic of alignment this is two separate insertions each of length one, so each one gets scored -10 + -4 = -14

I haven't written a function yet to implement counting up the length of the insertions, so the score returned by score_alignment() is not correct. Based on the description above, what is the correct score? Write out R code using sum() to calculate the correct score.

# Chapter 29

# Retrieving multiple sequences in R

```
library(compbio4all)
```

**By**: Avril Coghlan.
Multiple Alignment and Phylogenetic trees https://a-little-book-of-r-for-bioinf ormatics.readthedocs.io/en/latest/src/chapter5.html

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 29.1 Prelminaries

```
## package
library(compbio4all)
library(rentrez)    # still needed?
```

## 29.2 Retrieving a set of sequences from UniProt

Using websites or *R* you can search fort DNA or protein sequences in sequence databases such as the **NCBI** database and **UniProt**. Oftentimes, it is useful to retrieve several sequences at once. The *R* function `entrez_fetch()` from the *rentrez* package is useful for this purpose. Other packages can also, such as `sequinr` this but *rentrez* has the cleanest interface.

We'll retrieve the protein sequences for these UniProt accessions

1. P06747: rabies virus phosphoprotein

159

  2. P0C569: Mokola virus phosphoprotein
  3. O56773: Lagos bat virus phosphoprotein
  4. Q5VKP1: Western Caucasian bat virus phosphoprotein

Rabies virus is the virus responsible for rabies, which is classified by the WHO as a **neglected tropical disease**. Rabies is not a major human pathogen in the USA and Europe, but is problem in Africa. [Mokola virus](https://en.wikip edia.org/wiki/Mokola_lyssavirus() and rabies virus are closely related viruses that both belong to a group of viruses called the Lyssaviruses. Mokola virus causes a rabies-like infection in mammals including humans.

You can type make a vector containing the names of the sequences. Note that the accessions aren't numbers but are **quoted character strings**:

```
seqnames <- c("P06747",
              "P0C569",
              "O56773",
              "Q5VKP1")
```

Confirm that we are working with character data using `is.character()`

```
is.character(seqnames)
```

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

We can access the first element of the vector, P06747, using **bracket notation** like this:

```
seqnames[1]
```

```
## [1] "P06747"
```

The code to access the second and third elements of the vector of accessions is:

```
# 2nd accession
seqnames[2]
```

```
## [1] "P0C569"
```

```
# 3rd accession
seqnames[3]
```

```
## [1] "O56773"
```

Now let's use this vector of accessions to download sequence data. To make sure we understand what we're doing, first we'll download just the sequences one by one. This code retrieves the first sequence and store them in vector variable `seqs`.

```
seq1 <- rentrez::entrez_fetch(db = "protein",
                              id = seqnames[1],
                              rettype = "fasta")
```

We can do the next two sequences by change `id = ...` to `seqnames[2]` and `seqnames[3]`.

```r
# sequence two using seqnames[2]
seq2 <- rentrez::entrez_fetch(db = "protein",
                              id = seqnames[2],
                              rettype = "fasta")

# sequence two using seqnames[3]
seq3 <- rentrez::entrez_fetch(db = "protein",
                              id = seqnames[3],
                              rettype = "fasta")

# sequence two using seqnames[4]
seq4 <- rentrez::entrez_fetch(db = "protein",
                              id = seqnames[4],
                              rettype = "fasta")
```

Each of these is in raw FASTA format

```r
seq1
```

```
## [1] ">sp|P06747.1|PHOSP_RABVP RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Pro
```

To do an analysis of these we need to clean each of these vectors using `fasta_cleaner()`.

```r
seq1 <- fasta_cleaner(seq1, parse = T)
seq2 <- fasta_cleaner(seq2, parse = T)
seq3 <- fasta_cleaner(seq3, parse = T)
seq4 <- fasta_cleaner(seq4, parse = T)
```

Print out the first 20 letters of the first sequence

```r
seq1[1:20]
```

```
##  [1] "M" "S" "K" "I" "F" "V" "N" "P" "S" "A" "I" "R" "A" "G" "L" "A" "D" "L" "E"
## [20] "M"
```

Print out the first 20 letters of the second sequence

```r
seq2[1:20]
```

```
##  [1] "M" "S" "K" "D" "L" "V" "H" "P" "S" "L" "I" "R" "A" "G" "I" "V" "E" "L" "E"
## [20] "M"
```

We have these four sequences, each cleaned and in its own vector. We can therefore do dotplots, alignments, and other analyses. Creating all these separate vectors is a bit laborious. Luckily `entrez_fetch()` can download multiple sequences for us.

## 29.3   Downloading sequences in bulk

Previously we were giving `entrez_fetch` the name of just one sequence at
a time by using square brackets on our `seqnames` vector, e.g. `seqnames[1]`,
`seqnames[2]` etc. If don't include square brackets, `entrez_fetch()` will down-
load *all* of the sequences in succession and package them up into a single long,
formatted string. This may take a second or two, depending on your internet
connection and how busy the NCBI servers are.

```
seq_1_2_3_4 <- rentrez::entrez_fetch(db = "protein",
                                     id = seqnames,
                                     rettype = "fasta")
```

We can view what we have in a nice format using the `cat()` function.

```
cat(seq_1_2_3_4)
```

```
## >sp|P06747.1|PHOSP_RABVP RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full
## MSKIFVNPSAIRAGLADLEMAEETVDLINRNIEDNQAHLQGEPIEVDNLPEDMGRLHLDDGKSPNPGEMA
## KVGEGKYREDFQMDEGEDPSLLFQSYLDNVGVQIVRQIRSGERFLKIWSQTVEEIISYVAVNFPNPPGKS
## SEDKSTQTTGRELKKETTPTPSQRESQSSKARMAAQTASGPPALEWSATNEEDDLSVEAEIAHQIAESFS
## KKYKFPSRSSGILLYNFEQLKMNLDDIVKEAKNVPGVTRLARDGSKLPLRCVLGWVALANSKKFQLLVES
## NKLSKIMQDDLNRYTSC
##
## >sp|P0C569.1|PHOSP_MOKV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full
## MSKDLVHPSLIRAGIVELEMAEETTDLINRTIESNQAHLQGEPLYVDSLPEDMSRLRIEDKSRRTKTEEE
## ERDEGSSEEDNYLSEGQDPLIPFQNFLDEIGARAVKRLKTGEGFFRVWSALSDDIKGYVSTNIMTSGERD
## TKSIQIQTEPTASVSSGNESRHDSESMHDPNDKKDHTPDHDVVPDIESSTDKGEIRDIEGEVAHQVAESF
## SKKYKFPSRSSGIFLWNFEQLKMNLDDIVKAAMNVPGVERIAEKGGKLPLRCILGFVALDSSKRFRLLAD
## NDKVARLIQEDINSYMARLEEAE
##
## >sp|O56773.1|PHOSP_LBV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=
## MSKGLIHPSAIRSGLVDLEMAEETVDLVHKNLADSQAHLQGEPLNVDSLPEDMRKMRLTNAPSEREIIEE
## DEEEYSSEDEYYLSQGQDPMVPFQNFLDELGTQIVRRMKSGDGFFKIWSAASEDIKGYVLSTFMKPETQA
## TVSKPTQTDSLSVPRPSQGYTSVPRDKPSNSESQGGGVKPKKVQKSEWTRDTDEISDIEGEVAHQVAESF
## SKKYKFPSRSSGIFLWNFEQLKMNLDDIVKTSMNVPGVDKIAEKGGKLPLRCILGFVSLDSSKRFRLLAD
## TDKVARLMQDDIHNYMTRIEEIDHN
##
## >sp|Q5VKP1.1|PHOSP_WCBV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full
## MSKSLIHPSDLRAGLADIEMADETVDLVYKNLSEGQAHLQGEPFDIKDLPEGVSKLQISDNVRSDTSPNE
## YSDEDDEEGEDEYEEVVYDPVSAFQDFLDETGSYLISKLKKGEKIKKTWSEVSRVIYSYVMSNFPPRPPKP
## TTKDIAVQADLKKPNEIQKISEHKSKSEPSPREPVVEMHKHATLENPEDDEGALESEIAHQVAESYSKKY
## KFPSKSSGIFLWNFEQLKMNLDDIVQVARGVPGISQIVERGGKLPLRCMLGYVGLETSKRFRSLVNQDKL
## CKLMQEDLNAYSVSSNN
```

This is a good way to store FASTA files, but we can't work with them in this
format - they need to be in vectors. To give us FASTA data in a usable form the
`compbio4all` package has a function called `entrez_fetch_list()`, which is a

**wrapper** for `entrez_fetch()` which returns each sequence in its own separate slot in a list.

```
seq_1_2_3_4 <- entrez_fetch_list(db = "protein",
                                 id = seqnames,
                                 rettype = "fasta")
```

Let's look at the output

```
seq_1_2_3_4
```

```
## $P06747
## [1] ">sp|P06747.1|PHOSP_RABVP RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Pro
##
## $P0C569
## [1] ">sp|P0C569.1|PHOSP_MOKV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Prot
##
## $O56773
## [1] ">sp|O56773.1|PHOSP_LBV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Prote
##
## $Q5VKP1
## [1] ">sp|Q5VKP1.1|PHOSP_WCBV RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Prot
```

Note that before each sequence is its name, preceded by a dollar sign, e.g. `$P06747`.

This is the name of each element of our list. We can confirm that we have a list using `is.list()`.

```
is.list(seq_1_2_3_4)
```

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

The size of the list is the number of elements it contains, not the amount of data in each element. There are 4 sequences, so 4 elements, so length = 4.

```
length(seq_1_2_3_4)
```

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

We can access each element of the list by name, like this:

```
seq_1_2_3_4$P06747
```

```
## [1] ">sp|P06747.1|PHOSP_RABVP RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Pro
```

We can also access it by its index number, like this, using **double-bracket notation.**

```
seq_1_2_3_4[[1]] #NOTE: double brackets
```

```
## [1] ">sp|P06747.1|PHOSP_RABVP RecName: Full=Phosphoprotein; Short=Protein P; AltName: Full=Pro
```

Each element of the list is a vector. We can check this using `is.vector()` like this

```
is.vector(seq_1_2_3_4$P06747)
```

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

or using double brackets like this

```
is.vector(seq_1_2_3_4[[1]])
```

Its character data, which we can confirm with `class()`

```
class(seq_1_2_3_4$P06747) # dollar sign notation
class(seq_1_2_3_4[[1]])    # double-bracket notation
```

# Chapter 30

# Multiple sequence alignment in R

**By**: Nathan Brouwer, with some content adapted Coghlan (2011) Multiple Alignment and Phylogenetic trees and under the Creative Commons 3.0 Attribution License (CC BY 3.0). Functions print_msa() and clean_alignment() adapted from (Coglan 2011).

## 30.1 Preliminaries

### 30.1.1 Packages

We'll be using the package `ggmsa` for the first time and you will have to install it using `install.packages("ggmsa")`. You may be asked to re-restart R more then once during the installation process.

```r
# new packages
## Only install once
# install.packages("ggmsa")
library(ggmsa)         # visualize MSA

# other packages
library(compbio4all)
library(Biostrings)  # convert FASTA to AAStringSet
library(msa)           # multiple sequence alignment
```

### 30.1.2 Functions

The following key functions from compbio4all are used in this lesson

- fasta_cleaner()
- entrez_fetch_list()
- print_msa()
- clean_alignment()

## 30.2   Multiple sequence alignment (MSA)

A common task in bioinformatics is to download a set of related sequences from a database, and then to align those sequences using multiple alignment software. This is the first step in almost all phylogenetic analyses using sequence data.

## 30.3   Make MSA with msa()

WE'll use a package called `msa` (Bodenhofer et al. 2015). There are several packages that can do multiple sequence alignment in R, but they all require loading an external piece of alignment software that is just accessed via R. The `msa` package actually runs the alignment algorithms entirely in R, making workflows simpler.

### 30.3.1   Data preparation

The data is stored in object in combio4all

```
data(seq_1_2_3_4)
```

This data is in the form of a list

```
is.list(seq_1_2_3_4)
```

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

First we need to clean each one using `fasta_cleaner()`.

```
seq_1_2_3_4[[1]] <- fasta_cleaner(seq_1_2_3_4[[1]])
seq_1_2_3_4[[2]] <- fasta_cleaner(seq_1_2_3_4[[2]])
seq_1_2_3_4[[3]] <- fasta_cleaner(seq_1_2_3_4[[3]])
seq_1_2_3_4[[4]] <- fasta_cleaner(seq_1_2_3_4[[4]])
```

We need to put it into the form of a vector; in particular a **named vector**.

```
seq_1_2_3_4_vector <- c(P06747 = paste(seq_1_2_3_4[[1]], collapse = ""),
                P0C569 = paste(seq_1_2_3_4[[2]], collapse = ""),
                O56773 = paste(seq_1_2_3_4[[3]], collapse = ""),
                Q5VKP1 = paste(seq_1_2_3_4[[4]], collapse = ""))
```

We'll need need to convert our set of sequences to a particular format in preparation for alignment. This is done with the `AAStringSet()` function from `Biostrings`.

```
seq_1_2_3_4_stringset <- Biostrings::AAStringSet(seq_1_2_3_4_vector)
```

This just puts things in a format that makes the software happy. Doing this is a theme of bioinformatics work!

```
seq_1_2_3_4_stringset
```

```
## AAStringSet object of length 4:
##     width seq                                          names
## [1]     1 M                                            P06747
## [2]     1 M                                            P0C569
## [3]     1 M                                            O56773
## [4]     1 M                                            Q5VKP1
```

Next, we can run the alignment algorithm with the `msa()` function. There are many algorithms and pieces software for building alignments. The `msa` packages implements three major ones:

1. ClustalW
2. ClustalOmega
3. Muscle

We'll use ClustalW. Depending on the size and number of sequences this may take a little bit of time.

```
virusaln <- msa(inputSeqs = seq_1_2_3_4_stringset,
                method = "ClustalW")
```

```
## use default substitution matrix
```

We can view a snapshot of the alignment.

```
virusaln
```

```
## CLUSTAL 2.1
##
## Call:
##    msa(inputSeqs = seq_1_2_3_4_stringset, method = "ClustalW")
##
## MsaAAMultipleAlignment with 4 rows and 1 column
##     aln     names
## [1] M       P06747
## [2] M       P0C569
## [3] M       Q5VKP1
## [4] M       O56773
## Con M       Consensus
```

Each sequence is on a line. The **consensus sequence** indicates something similar to the average of all the sequences and is on the bottom and labeled `Con`. Question marks indicate that the software could not determine a consensus.

Dashes indicate either **indels** (**insertions** or **deletions**), or are added at the begining and end of sequences of unequal length so that they line up.

The output from `msa()` is a particular class of R object, a `MsaAAMultipleAlignment`.

```
is(virusaln)
```

```
## [1] "MsaAAMultipleAlignment" "AAMultipleAlignment"     "MsaMetaData"
## [4] "MultipleAlignment"
```

```
class(virusaln)
```

```
## [1] "MsaAAMultipleAlignment"
## attr(,"package")
## [1] "msa"
```

Next we'll want to visualize our alignment. In order to do further work with the MSA we're going to - I bet you can guess what happens next - make a conversion to the object. In this case we're going to make a subtle change by calling up the `class()` of the alignment and changing it from `MsaAAMultipleAlignment` (with "Msa" at the beginning) to `AAMultipleAlignment` (no "Msa"). (This is an annoying step and is needed because the folks who wrote the `msa` package have yet to collaobrate with the folks who wrote another package we'll use in a little bit).

Don't worry if you don't understand what's going on here - just run the code.

```
class(virusaln) <- "AAMultipleAlignment"
```

## 30.4   Viewing your MSA

There are several ways to view and explore your MSA

1. Within the R console using `compbio4all::print_msa()`
2. As an R plot using `ggmsa::ggmas()`
3. OPTIONAL: As a PDF using `msa::msaPrettyPrint()`

### 30.4.1   Viewing a long multiple alignment in the R console.

If you want to view a long multiple alignment within the R console, it is convenient to view the multiple alignment in blocks.

The function `print_msa()` (Coglan 2011)below will do this for you.  As its inputs, the function `print_msa()` takes the two things

1. `alignment`: input alignment
2. `chunksize`: the number of columns to print out in each block.

To use `print_msa()` we first need to do a little format conversion:

```r
virusaln_seqinr <- msaConvert(virusaln, type = "seqinr::alignment")
```

Then we can print it out like this, making the alignment 60 bases wide:

```r
print_msa(alignment = virusaln_seqinr,
          chunksize = 60)
```

```
## [1] "M 59"
## [1] "M 59"
## [1] "M 59"
## [1] "M 59"
## [1] " "
```

## 30.4.2 Visualizing alignments as an R plot

A powerful tool for visualizing focal parts of an alignment is `ggmsa`. If you haven't already, download it with `install.packages("ggmsa")` and load it with `library(ggmsa)`.

`ggmsa` prints a sequence alignment out within RStudio. Alignments can be large, so its important to select a subset of the alignment for visualzation.

First, let's look at the first 20 bases of our alignment. Note that we are using `virusaln`, NOT `virusaln_seqinr` (sorry for the back and forth between objects.)

```r
ggmsa(virusaln,    # virusaln, NOT virusaln_seqinr
      start = 1,
      end = 20)
```

0.500.751.001.251.50

### 30.4.2.1   OPTIONAL: File types used by `ggmsa`

The `ggmsa` packages currently only works with certain types of alignment output. We can see what these are with `available_msa()`.

```
available_msa()
```

```
## files currently available:
```

```
## .fasta
```

```
## XStringSet objects from 'Biostrings' package:
```

```
## DNAStringSet RNAStringSet AAStringSet BStringSet DNAMultipleAlignment RNAMultipleAl
```

```
## bin objects:
```

```
## DNAbin AAbin
```

As you can see there are a number of ways multiple sequence alignments can be represented in *R*. This has to do with the facts that i) There are many pieces of software / algorithms for making MSAs, and many bioinformatics packages that work with them.,

You can see that `AAMultipleAlignment` is listed, which the the format we set previously using the `class()` command.

The `msa` packages has a function `msaConvert()` which can change formats between different ways of representing MSAs which may be useful.

### 30.4.3  OPTIONAL: Print MSA to PDF

The `msa` package has a fabulous function, `msaPrettyPrint()` for rendering an MSA to PDF. It can take a little bit to run, and in order to vioew the PDF you need to locate the output. (Again, we'll use `virusaln`, not `virusaln_seqinr`).

```
msaPrettyPrint(virusaln,      # virusaln, NOT virusaln_seqinr
               file = "my_msa.pdf",
               askForOverwrite = F)
```

On a Mac usually searching in Finder will locate the file even after is just created. You can ask R where it is saving thing using `getwd()`.

```
getwd()
```

```
## [1] "/Users/nlb24/google_backup_sync_nlb24/lbrb"
```

You can change where R is saving things using the RStudio menu by clicking on Session -> Set Working Directory -> Choose directory...

## 30.5  Discarding very poorly conserved regions from an alignment

It is often a good idea to discard very **poorly conserved** regions from a multiple sequence alignment before visualizing it or building a phylogenetic tree, as the very poorly conserved regions are likely to be regions that are either **non-homologous** between the sequences being considered (and so do not have any phylogenetic signal), or are homologous but are so **diverged** that they are very difficult to align accurately (and so may add noise to the phylogenetic analysis, and decrease the accuracy of the inferred tree).

To discard very poorly conserved regions from a multiple alignment, you can use the following R function, `clean_alignment()` ((Coglan 2011))

The function `clean_alignment()` takes three arguments (inputs):

1. the input alignment;
2. minpcnongap: the minimum percent of letters in an alignment column that must be non-gap characters for the column to be kept; and
3. minpcid: the minimum percent of pairs of letters in an alignment column that must be identical for the column to be kept.

For example, if we have a single column (**locus**) with letters "T", "A", "T", "-" (in four sequences), then 75% of the letters are non-gap characters; and the pairs of letters between the three non-gap sequences are

- 1 versus 2: "T,A",
- 1 versus 3: "T,T",
- 2 versus 3: "A,T",

Therefore 33% of the pairs of letters are identical (**PID**) for that position in the alignment.

If you look at the multiple alignment for the virus phosphoprotein sequences (which we printed out using function `print_msa()`, see above), we can see that the last few columns are poorly aligned (contain many gaps and mismatches), and probably add noise to the phylogenetic analysis.

Let's cleave off anything with less than 30% non-gap and less than 30% PID. NOTE: we're bac to using `virusaln_seqinr`, not `virusaln`.

```
virusaln_seqinr_clean <- clean_alignment(alignment = virusaln_seqinr,  # virusaln_seqi
                                         minpcnongap = 30,
                                         minpcid = 30)
```

In this case, we required that at least 30% of letters in a column are not gap characters for that column to be kept, and that at least 30% of pairs of letters in an alignment column must be identical for the column to be kept.

We can print out the filtered alignment by typing:

```
print_msa(virusaln_seqinr_clean)
```

The filtered alignment is shorter, and is missing some of the poorly conserved regions of the original alignment.

Note that it is not a good idea to filter out too much of your alignment, as if you are left with few columns in your filtered alignment, you will be basing your phylogenetic tree upon a very short alignment (little data), and so the tree may be unreliable. Therefore, you need to achieve a balance between discarding the dodgy (poorly aligned) parts of your alignment, and retaining enough columns of the alignment that you will have enough data to based your tree upon.

# Chapter 31

# The BLOSUM scoring matrix in R

**By** Nathan Brouwer

```
library(compbio4all)
```

## 31.1   Introduction

In this lesson we will introduce the basic structure of the BLOSUM scoring matrix.

## 31.2   Preliminaries

### 31.2.1   Packages

- BiocManager
- Biostrings
- combio4all

### 31.2.2   Vocab

#### 31.2.2.1   Math / R vocab

- triangular matrix
- square matrix
- lower triangle
- upper triangle
- symmetric matrix
- matrix diagonal

173

- named vectors
- named matrices
- accessing items in named vectors or matrices
- square bracket notation

#### 31.2.2.2  Bioinformatics vocab

- scoring matrix
- BLOSUM scoring matrix
- ambiguity codes

### 31.2.3  Functions used

#### 31.2.3.1  Base R functions

- `data()`
- `is()`
- `nrow()`, `ncol()`,
- `dim()`,
- `names()`, `colnames()`, `rownames()`,
- `head()`, `tail()`

#### 31.2.3.2  Specific function

- combio4all::tri_print()
- combio4all::diag_show()

## 31.3  Load packages

Load basic packages

```
library(flextable)
library(webshot)
```

Install the Biostrings package from Bioconductor; you don't have to run this if you already happen to have downloaded Biostrings before

```
library(Biostrings)
```

## 31.4  The BLOSUM matrix

When data is contained within an R package we can load it using the data() function. We'll work with a version of the BLOSUM62 matrix in the Biostrings package

Load BLOSUM62 amino acid substitution matrix using data()

```
data(BLOSUM62)
```

Bioinformatics uses a lot of difference data structures, including dataframes, matrices, vectors, etc.

What is the **data structure** of BLOSUM62? Use is()

Take a look at the whole matrix by just calling up the object on its own:

Whenever you start working with data its important to get a sense of what's there, how much of its there, and if there is anything goofy. Since it can be hard to see a dataframe or matrix on a screen its important to explore it with various commands, including `nrow()`, `ncol()`, `dim()`, `colnames()`, `rownames()`, `head()`, and `tail()`

Use `nrow()` and `ncol()` to determine the number of rows and columns.

This is a **square matrix** since the number of rows equals the number of columns. It is also a symmetrical matrix since the **lower triangle** of the matrix is the same as the **upper triangle**. The compbio4all package has functions for displaying these.

*R* shows us the full **symmetric matrix**, though in books usually they just show the **lower triangle**. A symmetric matrix is one where the upper and lower triangles are identical.

```
BLOSUM62
```

```
##     A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
## A   4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4
## R  -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1 -2  0 -1 -4
## N  -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  4 -3  0 -1 -4
## D  -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4 -3  1 -1 -4
## C   0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -1 -3 -1 -4
## Q  -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0 -2  4 -1 -4
## E  -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1 -3  4 -1 -4
## G   0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -4 -2 -1 -4
## H  -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0 -3  0 -1 -4
## I  -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3  3 -3 -1 -4
## L  -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4  3 -3 -1 -4
## K  -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0 -3  1 -1 -4
## M  -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3  2 -1 -1 -4
## F  -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3  0 -3 -1 -4
## P  -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -3 -1 -1 -4
## S   1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0 -2  0 -1 -4
## T   0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -1 -4
## W  -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -2 -2 -1 -4
## Y  -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -1 -2 -1 -4
## V   0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2  0 -3 -1  4 -3  2 -2 -1 -4
```

```
## B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4 -3  0 -1 -4
## J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2 -3  3 -3 -1 -4
## Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2  0 -3  4 -1 -4
## X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -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  1
```

The function `tri_print()` will print just the lower triangle, with all the other
values left empty. (This might be a bit slow)

```
tri_print(BLOSUM62, as.image = T)
```

| x | A | R | N | D | C | Q | E |
|---|---|---|---|---|---|---|---|
| A | 4 | | | | | | |
| R | -1 | 5 | | | | | |
| N | -2 | 0 | 6 | | | | |
| D | -2 | -2 | 1 | 6 | | | |
| C | 0 | -3 | -3 | -3 | 9 | | |
| Q | -1 | 1 | 0 | 0 | -3 | 5 | |
| E | -1 | 0 | 0 | 2 | -4 | 2 | 5 |
| G | 0 | -2 | 0 | -1 | -3 | -2 | -2 |
| H | -2 | 0 | 1 | -1 | -3 | 0 | 0 |
| I | -1 | -3 | -3 | -3 | -1 | -3 | -3 |
| L | -1 | -2 | -3 | -4 | -1 | -2 | -3 |
| K | -1 | 2 | 0 | -1 | -3 | 1 | 1 |
| M | -1 | -1 | -2 | -3 | -1 | 0 | -2 |
| F | -2 | -3 | -3 | -3 | -2 | -3 | -3 |
| P | -1 | -2 | -2 | -1 | -3 | -1 | -1 |
| S | 1 | -1 | 1 | 0 | -1 | 0 | 0 |
| T | 0 | -1 | 0 | -1 | -1 | -1 | -1 |
| W | -3 | -3 | -4 | -4 | -2 | -2 | -3 |
| Y | -2 | -2 | -2 | -3 | -2 | -1 | -2 |
| V | 0 | -3 | -3 | -3 | -1 | -2 | -2 |
| B | -2 | -1 | 4 | 4 | -3 | 0 | 1 |
| J | -1 | -2 | -3 | -3 | -1 | -2 | -3 |
| Z | -1 | 0 | 0 | 1 | -3 | 4 | 4 |

| x | A | R | N | D | C | Q | E | G |
|---|---|---|---|---|---|---|---|---|
| X | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| * | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 |

**BLOSUM62** is a matrix for scoring differences between sequences of amino acids. Is the size of this *R* object therefore a bit odd? We should check out what these rows and columns actually are.

What are the row names of the matrix? Use rownames() to determine this.

What are the column names? Run the appropriate command below to determine this.

Look at the top of the matrix using `head()`.

Look at the bottom of the matrix using `tail()`.

What have we learned? There's more than 20 rows and columns (why is 20 the reference point?). The B, J, Z, X and * represent more **ambiguity codes**. These usually show up due to sequencing errors or lack of full resolution of an amino acid in the sequence because they are so close chemically (e.g. Asparagine versus aspartic acid) .

- **B** means its ambiguous whether the amino acid is asparagine or aspartici acid (D or N)
- **J** is used when its ambiguous whether its I or L.
- **Z** means its ambiguous whether the amino acid is glutamine or glutamic acid (E or Q).
- **X** is a stand in for any or an unknown code. This might occur if there is a sequencing error.
- The **asterisk** "*" is used to represent **stop codons**.

We'll be working with sequences without any ambiguities. In the following sections of code we'll remove these ambiguity codes.

## 31.5   Visualizing the BLOSUM matrix

We can visualize the BLOSUM matrix by first drawing all 20 amino acids in a circle.

```
##
## Attaching package: 'igraph'

## The following object is masked from 'package:flextable':
##
##     compose
```

```
## The following object is masked from 'package:Biostrings':
##
##     union


## The following object is masked from 'package:IRanges':
##
##     union


## The following object is masked from 'package:S4Vectors':
##
##     union


## The following objects are masked from 'package:BiocGenerics':
##
##     normalize, path, union


## The following objects are masked from 'package:stats':
##
##     decompose, spectrum


## The following object is masked from 'package:base':
##
##     union
```
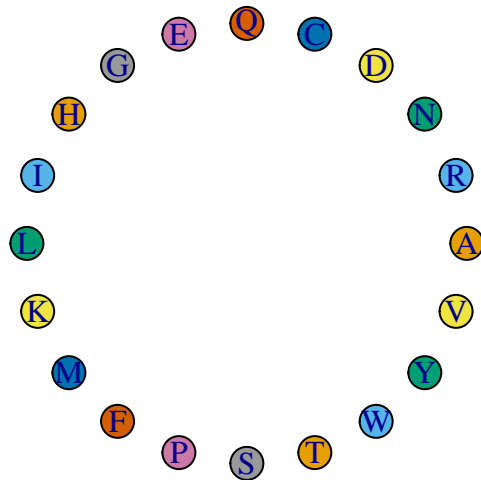


In theory, all mutations can change any codon into any other codon. We can represent mutations that result in a changed in the amino acid by connecting each amino acid with an arrow. We'll do this for adenine. Thicker arrows represent more common transitions.

**Histogram of BLOSUM62_all_pos_A[1, ]**



Often, there will be no change in a codon, the changes don't impact which amino acid, or a mutation to a different codon is reversed due to a **back mutation**. We can represent the fact that an adenine can stay as adenine with a **self loop**. You can see that the most common "transition" is for adenine to remain as adenine.

```
blosum.graph.A <- graph_from_adjacency_matrix(BLOSUM62_all_pos_A,
                                              diag = T,
                                              weighted = T,
                                              mode = "directed")
```

```
plot.igraph(blosum.graph.A ,
            layout = blosum.graph.circle,
            edge.arrow.size =0.5,
            vertex.color= 1:20,
            edge.color = 1:20,
            edge.width = E(blosum.graph.A)$weight/10
            )
```
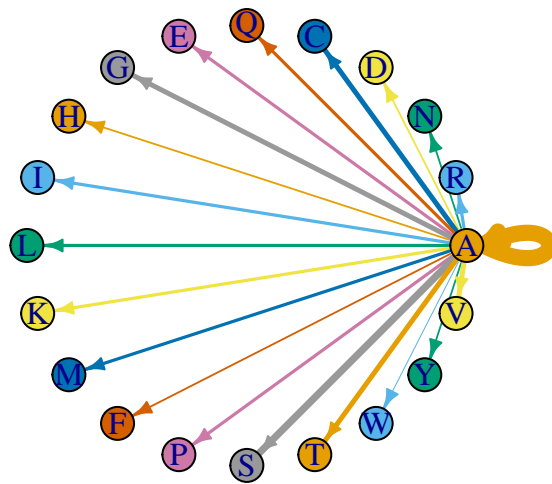


Instead of laying things out in circle, we can have them oriented so that the amino acids adenine is most likely to transition to are closest to it.

```
blosum.graph.A <- graph_from_adjacency_matrix(BLOSUM62_all_pos_A,
                                              diag = F,
                                              weighted = T,
                                              mode = "directed")
plot.igraph(blosum.graph.A ,
            #layout = blosum.graph.circle,
            edge.arrow.size =0.5,
            vertex.color= 1:20,
            edge.color = 1:20,
            edge.width = E(blosum.graph.A)$weight/10
            )
```
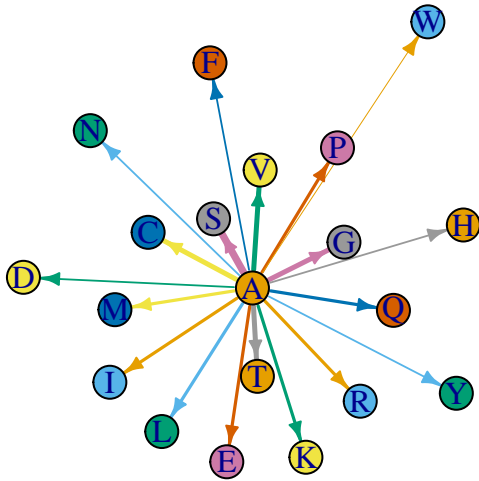
We can extend this by considering all possible transition and making a plot that tries to orient each amino acid close to the other amino acids its likely to transition to, and far from those its unlikely to. This is a bit tricky, since all transitions are now being consider.

```
blosum.graph.all <- graph_from_adjacency_matrix(BLOSUM62_all_pos,
                                                 diag = F,
                                                 weighted = T,
                                                 mode = "directed")

plot.igraph(blosum.graph.all,
            edge.arrow.size = 0,
            vertex.color= 1:20,
            edge.color = 1)
```
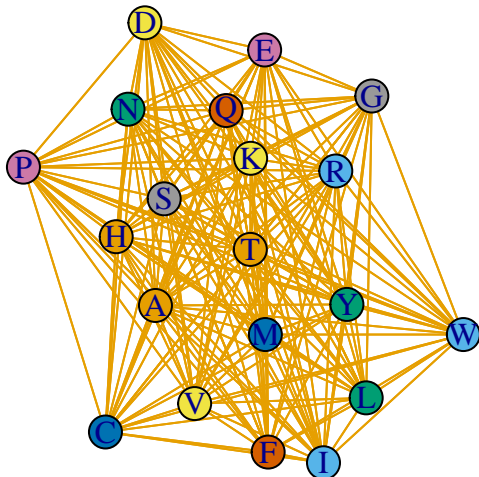
# Chapter 32

# Calculating genetic distances between sequences

**By**: Nathan Brouwer, with some content adapted Coghlan (2011) Multiple Alignment and Phylogenetic trees and under the Creative Commons 3.0 Attribution License (CC BY 3.0). Functions print_msa() and clean_alignment() adapted from (Coglan 2011).

## 32.1 Preliminaries

```
library(compbio4all)
library(msa)
library(seqinr)
library(ape)
```

## 32.2 Introduction

A common first step in performing a **phylogenetic analysis** is to calculate the **pairwise genetic distances** between sequences. The **genetic distance** is an estimate of the evolutionary **divergence** between two sequences, and is usually measured in quantity of evolutionary change, e.g., an estimate of the number of mutations that have occurred since the two sequences shared a **common ancestor**.

We can calculate the genetic distances between protein sequences using the `dist.alignment()` function in the `seqinr` package. The `dist.alignment()`

function takes a multiple sequence alignment (MSA) as input.  Based on the MSA that you give it, `dist.alignment()` calculates the genetic distance between each *pair* of proteins in the multiple alignment, yielding pairwise distances.  For example, to calculate genetic distances between the virus phosphoproteins based on the multiple sequence alignment stored in virusaln, we type:

```
data(virusaln)
virusdist <- seqinr::dist.alignment(virusaln_seqinr)  # Calculate the genetic distance

virusdist # Print out the genetic distance matrix
```

```
##          P06747 P0C569 Q5VKP1
## P0C569      0
## Q5VKP1      0       0
## O56773      0       0       0
```

**NOTE** My result are different from the original results, shown here: I need to check the settings used for the MSA

P0C569 O56773 P06747 O56773 0.4142670 P06747 0.4678196 0.4714045 Q5VKP1 0.4828127 0.5067117 0.5034130

The genetic distance matrix above shows the genetic distance between each pair of proteins.  The sequences are referred to by their **UniProt accessions**.  Recall that

- P06747 = rabies virus phosphoprotein
- P0C569 is Mokola virus phosphoprotein
- O56773 is Lagos bat virus phosphoprotein
- Q5VKP1 is Western Caucasian bat virus phosphoprotein.

Based on the genetic **distance matrix** above, we can see that the genetic distance between Lagos bat virus phosphoprotein (O56773) and Mokola virus phosphoprotein (P0C569) is smallest (about 0.414).  Similarly, the genetic distance between Western Caucasian bat virus phosphoprotein (Q5VKP1) and Lagos bat virus phosphoprotein (O56773) is the biggest (about 0.507).

The larger the genetic distance between two sequences, the more amino acid changes (such as change from Asp to Met) or **indels** that have occurred since they shared a common ancestor, and the longer ago their **common ancestor** probably lived.  (The relationship between number of mutations and time, however, depends on the mutation rate and generation time of the organism).

## 32.3   Calculating   genetic   distances   between DNA/mRNA sequences

Just like for protein sequences, you can calculate genetic distances between DNA (or mRNA) sequences based on an alignment of the sequences.  The RefSeq DNA accession numbers for the proteins we've been using are:

- AF049118 = mRNA sequence for Mokola virus phosphoprotein,
- AF049114 = mRNA sequence for Mokola virus phosphoprotein,
- AF049119 = mRNA sequence for Lagos bat virus phosphoprotein,
- AF049115 = mRNA sequence for Duvenhage virus phosphoprotein.

We can retrieve these DNA sequences using `entrez_fetch_list()`. Some notes about how we'll use this function works:

1. `db` is short for "database"
2. the database is called `nuccore` (not `genebank` or `gene`)
3. the argument `rettype` is "tt"; I think it stands for "REtURN type" (I also forget the second "t")

```r
# put accessions in vector
accessions_mrna <- c("AF049118","AF049114","AF049119","AF049115")

# get sequences
virus_mrna_list <- entrez_fetch_list(db = "nuccore",  # "nuccore" db for DNA
                                     id = accessions_mrna,
                                     rettype = "FASTA") # rettype has two t
```

We can clean these three sequences using a simple `for()` loop. We set `parse = F` so we get things back as single character string.

```r
for(i in 1:length(virus_mrna_list)){
  virus_mrna_list[[i]] <- fasta_cleaner(virus_mrna_list[i],
                                        parse = F)
}
```

Now we need to convert each of these into a named vector of sequences.

```r
mra_seq_vector <- c(AF049118 =  virus_mrna_list[[1]] ,
                    AF049114 = virus_mrna_list[[2]] ,
                    AF049119 = virus_mrna_list[[3]] ,
                    AF049115 = virus_mrna_list[[4]])
```

Finally convert this to a "stringset" using `Biostrings::DNAStringSet()`.

```r
dna_seq_stringset <- Biostrings::DNAStringSet(mra_seq_vector)
```

Let's see what we got

```r
dna_seq_stringset
```

```
## DNAStringSet object of length 4:
##     width seq                                           names
## [1]  1083 TCAGAGAGCTCCTTTGCAAAGGA...AAAATGAAAAAAACATTTAACAT AF049118
## [2]  1016 CACTTCGAACGGTACATATTAGT...AAATTCTCAAAAATGAGCTCTCC AF049114
## [3]  1010 CAAAATGTCATCTCATATGAAAT...GGAACTTAAAACACAGAAGGTTT AF049119
## [4]   990 ATGAGCAAGATTTTTATCAATCC...ACACCACTGACAAAATGAACATC AF049115
```

Now we can make an alignment use `msa()`.

```
virus_mrna_aln <- msa(inputSeqs = dna_seq_stringset,
                      method = "ClustalW")
```

```
## use default substitution matrix
```

The output looks like this

```
virus_mrna_aln
```

```
## CLUSTAL 2.1
##
## Call:
##    msa(inputSeqs = dna_seq_stringset, method = "ClustalW")
##
## MsaDNAMultipleAlignment with 4 rows and 1097 columns
##      aln                                                names
## [1] -----------------------C...------------------------ AF049114
## [2] ---------------------------...---------------------- AF049119
## [3] TCAGAGAGCTCCTTTGCAAAGGAGGA...AAAAACATTTAACAT--------- AF049118
## [4] -------------------------...CAACACCACTGACAAAATGAACATC AF049115
## Con -------------------------...-??-??---?-???---------- Consensus
```

This looks a LOT different than an amino acid alignment, which looked like this:

```
data(virusaln)
virusaln
```

```
## AAMultipleAlignment with 4 rows and 306 columns
##      aln                                                names
## [1] MSKDLVHPSLIRAGIVELEMAEETTD...NDKVARLIQEDINSYMARLEEAE-- P0C569
## [2] MSKGLIHPSAIRSGLVDLEMAEETVD...TDKVARLMQDDIHNYMTRIEEIDHN O56773
## [3] MSKIFVNPSAIRAGLADLEMAEETVD...SNKLSKIMQDDLNRYTSC------- P06747
## [4] MSKSLIHPSDLRAGLADIEMADETVD...QDKLCKLMQEDLNAYSVSSNN---- Q5VKP1
```

Why might the be different?  First, examine the output above and determine how long the DNA alignment is versus the amino acid alignment?  Why are they different, and why is one longer than the other?

The DNA alignment is 1097 columns, while the amino acid alignment is only 306 rows.  Note that 306*3 = 918.  1097 is pretty close to 1097.  What's the relevance of multiplying by 3?

## 32.4   Calculationg genetic distance

You can calculate a genetic distance for DNA or mRNA sequences using the `dist.dna()` function in the `ape` package. `dist.dna()` takes a MSA of DNA or

mRNA sequences as its input, and calculates the genetic distance between each pair of DNA sequences in the multiple alignment.

The `dist.dna()` function requires the input alignment to be in a special format known as `DNAbin` format, so we must use the `as.DNAbin()` function to convert our DNA alignment into this format before using the `dist.dna()` function.

```r
# Convert the alignment to "DNAbin" format
virus_mrna_aln_bin <- ape::as.DNAbin(virus_mrna_aln)
```

The output of `as.DNAbin()` gives us a short summery of the alignment

```r
virus_mrna_aln_bin
```

```
## 4 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1097
##
## Labels:
## AF049114
## AF049119
## AF049118
## AF049115
##
## Base composition:
##     a     c     g     t
## 0.324 0.202 0.240 0.234
## (Total: 4.39 kb)
```

Now to make and view the alignment:

```r
# Calculate the genetic distance matrix
virus_mrna_dist <- ape::dist.dna(virus_mrna_aln_bin)

# Print out the genetic distance matrix
virus_mrna_dist
```

```
##          AF049114  AF049119  AF049118
## AF049119 0.3400576
## AF049118 0.5235850 0.5637372
## AF049115 0.6854129 0.6852311 0.7656023
```

NOTE: my results for this alignment are the same as the original by Coghlan. I'm not sure why my amino acid alignment produces divergent results but the DNA is the same.

# Chapter 33

# Unrooted neighbor-joining phylogenetic trees

**NOTE**: the code for this chapter works as intended but there are some differences between my results and what is reported by the original author of the chapter. This is likely to do with different alignment software, though it could just be a typo.

**By**: Avril Coghlan. Multiple Alignment and Phylogenetic trees https://a-little-book-of-r-for-bioinformatics.readthedocs.io/en/latest/src/chapter5.html

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 33.1 Preliminaries

```
library(compbio4all)
library(seqinr)
```

You will need to install the `ape` package if you do not have it already using `install.packages("ape")`.

```
library(ape)
```

### 33.1.1 Key functions

- compbio4all::unrooted_NJ_tree (Coghlan 200x)

### 33.1.2 Key vocab

- clade

- bootstrap
- resample
- rooted vs. unrooted tree
- outgroup

## 33.2   Building an unrooted phylogenetic tree for protein sequences

Once we have a **distance matrix** that gives the **pairwise distances** between all our protein sequences, we can build a **phylogenetic tree** based on that distance matrix. One method for using this is the **neighbor-joining algorithm**.

If we have the distance matrix already made we can make the tree like this using `ape::nj()`. The distance matrix is saved in `compbio4all` as `virus_mrna_dist`. Load this with `data()`.

```r
# load the distance matrix
data(virus_mrna_dist)

# build the tree
tree_from_distmat <- nj(virus_mrna_dist)


plot.phylo(tree_from_distmat, type = "unrooted")
```
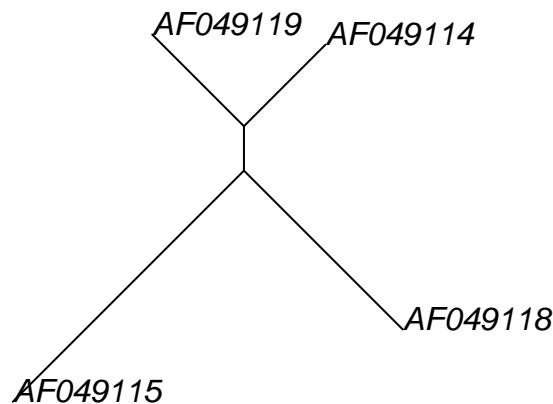


### 33.2.1   Build tree with `unrooted_NJ_tree()`

Coghlan (2011) wrote a function to simply the steps of making an NJ tree. The R function `unrooted_NJ_tree()` is a **wrapper** for functions from the `ape` package which builds a phylogenetic tree based on an alignment of sequences, using the NJ algorithm.

The `unrooted_NJ_tree()` function takes an alignment of sequences its input,

calculates **pairwise distances** between the sequences based on the alignment behind the scenes, and then builds a phylogenetic tree based on the pairwise distances. It returns the phylogenetic tree, and also makes a plot of that tree. It also gives us information about to what extent the data in the original MSA support the evolutionary relationships shown in the tree.

The alignment is saved in `compbio4all` as `virusaln` and can be loaded with the `data()` command.

```
data(virusaln_seqinr)
```

Take a look at the structure of the data

```
str(virusaln_seqinr)
```

```
## List of 4
##  $ nb : int 4
##  $ nam: chr [1:4] "P0C569" "O56773" "P06747" "Q5VKP1"
##  $ seq: chr [1:4] "MSKDLVHPSLIRAGIVELEMAEETTDLINRTIESNQAHLQGEPLYVDSLPEDMSRLRIEDKSRRTKTEEEERDE(
##  $ com: logi NA
##  - attr(*, "class")= chr "alignment"
```

```
virusalntree <- unrooted_NJ_tree(virusaln_seqinr_clean,
                                 type="protein")
```

Note that you need to specify that the type of sequences that you are using are protein sequences when you use `unrooted_NJ_tree()`, by setting `type=protein`.

We can see that `Q5VKP1` (Western Caucasian bat virus phosphoprotein) and `P06747` (rabies virus phosphoprotein) have been grouped together into a **clade** on the tree, and that `O56773` (Lagos bat virus phosphoprotein) and `P0C569` (Mokola virus phosphoprotein) are grouped together.

This is consistent with what we saw above in the genetic distance matrix, which showed that the genetic distance between Lagos bat virus phosphoprotein (O56773) and Mokola virus phosphoprotein (P0C569) is relatively small.

## 33.3 Boostrap values indicate support for clades

In the plot, the numbers in blue boxes are **bootstrap values** for the nodes in the tree. A bootstrap value for a particular node in the tree gives an idea of the confidence that we have in the clade (group) defined by that node in the tree. If a node has a high bootstrap value (near 100%) then we are very confident that the clade defined by the node is correct, while if it has a low bootstrap value (near 0%) then we are not so confident.

Note that the fact that a bootstrap value for a node is high does not necessarily guarantee that the clade defined by the node is correct, but just tells us that it

is quite likely that it is correct given the data and analysis we're using.

The bootstrap values are calculated by making many (for example, 100) random **resamples** of the alignment that the phylogenetic tree was based upon. Each resample of the alignment consists of a certain number x (e.g.. 200) of randomly sampled *columns* from the alignment. Each resample of the alignment (e.g.. 200 randomly sampled columns) forms a sort of fake alignment of its own, and a phylogenetic tree can be based upon the *resample*. We can make 100 random resamples of the alignment, and build 100 phylogenetic trees based on the 100 resamples. These 100 trees are known as the **bootstrap trees**. For each clade (grouping) that we see in our original phylogenetic tree, we can count in how many of the 100 bootstrap trees it appears. This is known as the **bootstrap value** for the clade in our original phylogenetic tree.

For example, if we calculate 100 random resamples of the virus phosphoprotein alignment, and build 100 phylogenetic trees based on these resamples, we can calculate the bootstrap values for each clade in the virus phosphoprotein phylogenetic tree.

**NOTE:** I am currently not able to reproduce these results:

In this case, the bootstrap value for the node defining the clade containing Q5VKP1 (Western Caucasian bat virus phosphoprotein) and P06747 (rabies virus phosphoprotein) is 25%, while the bootstrap value for node defining the clade containing of Lagos bat virus phosphoprotein (O56773) and Mokola virus phosphoprotein (P0C569) is 100%. The bootstrap values for each of these clades is the percent of 100 bootstrap trees that the clade appears in.

Therefore, we are very confident that Lagos bat virus and Mokola virus phosphoproteins should be grouped together in the tree. However, we are not so confident that the Western Caucasian bat virus and rabies virus phosphoproteins should be grouped together.

## 33.4   Branch lengths indicate divergence between sequences

The lengths of the branches in the plot of the tree are proportional to the amount of evolutionary change (estimated number of mutations) along the branches. In this case, the branches leading to Lagos bat virus phosphoprotein (O56773) and Mokola virus phosphoprotein (P0C569) from the node representing their common ancestor are slightly shorter than the branches leading to the Western Caucasian bat virus (Q5VKP1) and rabies virus (P06747) phosphoproteins from the node representing their common ancestor.

This suggests that there might have been more mutations in the Western Caucasian bat virus (Q5VKP1) and rabies virus (P06747) phosphoproteins since they shared a common ancestor, than in the Lagos bat virus phosphoprotein

(O56773) and Mokola virus phosphoprotein (P0C569) since they shared a common ancestor.

## 33.5  Unrooted trees lack an outgroup

The tree above of the virus phosphoproteins is an **unrooted** phylogenetic tree as it does not contain an **outgroup** sequence; that is, a sequence of a protein that is known to be more distantly related to the other proteins in the tree than they are to each other.

As a result, we cannot tell which direction evolutionary time ran in along the internal branches of the tree. For example, we cannot tell whether the node representing the common ancestor of (O56773, P0C569) was an ancestor of the node representing the common ancestor of (Q5VKP1, P06747), or the other way around.

In order to build a **rooted** phylogenetic tree, we need to have an outgroup sequence in our tree. In the case of the virus phosphoproteins, this is unfortunately not possible, as there is not any protein known that is more distantly related to the four proteins already in our tree than they are to each other.

However, in many other cases, an outgroup - a sequence known to be more distantly related to the other sequences in the tree than they are to each other - is known, and so it is possible to build a rooted phylogenetic tree.

We discussed above that it is a good idea to investigate whether discarding the poorly conserved regions of a multiple alignment has an effect on the phylogenetic analysis. In this case, we made a filtered copy of the multiple alignment and stored it in the variable virusln_seqinr_clean (see above). We can make a phylogenetic tree based this filtered alignment, and see if it agrees with the phylogenetic tree based on the original alignment:

```
data(virusaln_seqinr_clean)
cleanedvirusalntree <- unrooted_NJ_tree(virusaln_seqinr_clean,type="protein")
```

As in the phylogenetic tree based on the raw (unfiltered) multiple alignment, O56773 and P0C569 are still grouped together, and Q5VKP1 and P06747 are still grouped together. Thus, filtering the multiple alignment does not have an effect on the tree. The bootstrap value, however, have changed.

If we had found a difference in the trees made using the unfiltered and filtered multiple alignments, we would have to examine the multiple alignments closely, to see if the unfiltered multiple alignment contains a lot of very poorly aligned regions that might be adding noise to the phylogenetic analysis (if this is true, the tree based on the filtered alignment is likely to be more reliable).

# Chapter 34

# Testing the significance of an alignment

```
library(compbio4all)
library(Biostrings)
```

**By**: Avril Coghlan.
Pairwise Sequence Alignment https://a-little-book-of-r-for-bioinformatics.rea
dthedocs.io/en/latest/src/chapter4.html

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

## 34.1 Calculating the statistical significance of a pairwise global alignment

We have seen that when we align the "PAWHEAE" and "HEAGAWGHEE" protein sequences, they have some similarity, and the score for their optimal global alignment is -5.

But is this alignment statistically significant? In other words, is this alignment better than we would expect between any two random proteins?

The Needleman-Wunsch alignment algorithm will produce a global alignment even if we give it two unrelated random protein sequences, although the alignment score would be low.

Therefore, we should ask: is the score for our alignment better than expected between two random sequences of the same lengths and amino acid compositions?

It is reasonable to expect that if the alignment score is statistically significant, then it will be higher than the scores obtained from aligning pairs of random protein sequences that have the same lengths and amino acid compositions as our original two sequences.

Therefore, to assess if the score for our alignment between the "PAWHEAE" and "HEAGAWGHEE" protein sequence is statistically significant, a first step is to make some random sequences that have the same amino acid composition and length as one of our initial two sequences, for example, as the same amino acid composition and length as the sequence "PAWHEAE".

How can we obtain random sequences of the same amino acid composition and length as the sequence "PAWHEAE"? One way is to generate sequences using a multinomial model for protein sequences in which the probabilities of the different amino acids set to be equal to their frequencies in the sequence "PAWHEAE".

That is, we can generate sequences using a multinomial model for proteins, in which the probability of "P" is set to 0.1428571 (1/7); the probability of "A" is set to 0.2857143 (2/7); the probability of "W" is set to 0.1428571 (1/7); the probability of "H' is set to 0.1428571 (1/7); and the probabilty of 'E' is set to 0.2857143 (2/7), and the probabilities of the other 15 amino acids are set to 0.

To generate a sequence with this multinomial model, we choose the letter for each position in the sequence according to those probabilities. This is as if we have made a roulette wheel in which $1/7th$ of the circle is taken up by a pie labelled "P", $2/7ths$ by a pie labelled "A", $1/7th$ by a pie labelled "W", $1/7th$ by a pie labelled "H", and $2/7ths$ by a pie labeled "E":

To generate a sequence using the multinomial model, we keep spinning the arrow in the centre of the roulette wheel, and write down the letter that the arrow stops on after each spin. To generate a sequence that is 7 letters long, we can spin the arrow 7 times. To generate 1000 sequences that are each 7 letters long, we can spin the arrow 7000 times, where the letters chosen form 1000 7-letter amino acid sequences.

To generate a certain number (eg.1000) random amino acid sequences of a certain length using a multinomial model, you can use the function `make_seqs_multinom_mod()` below:

The function `make_seqs_multinom_mod()` generates X random sequences with a multinomial model, where the probabilities of the different letters are set equal to their frequencies in an input sequence, which is passed to the function as a string of characters (eg. "PAWHEAE").

The function returns X random sequences in the form of a vector which has X elements, the first element of the vector contains the first sequence, the second element contains the second sequence, and so on.

You will need to copy and paste this function into R before you can use it.

We can use this function to generate 1000 7-letter amino acid sequences using a multinomial model in which the probabilities of the letters are set equal to their frequencies in "PAWHEAE" (i.e. probabilities 1/7 for P, 2/7 for A, 1/7 for W, 1/7 for H and 2/7 for E), by typing:

```
randomseqs <- make_seqs_multinom_mod('PAWHEAE',1000)
randomseqs[1:10] # Print out the first 10 random sequences
```

```
##  [1] "EAEEHEE" "AHEAAAA" "AEAEPAW" "AAWHPHW" "HHAWPEP" "AHAEPEE" "AWWAAAH"
##  [8] "PEPWEAH" "EAHAAPE" "AAAAPAE"
```

The 1000 random sequences are stored in a vector `randomseqs` that has 1000 elements, each of which contains one of the random sequences.

We can then use the Needleman-Wunsch algorithm to align the sequence "HEAGAWGHEE" to one of the 1000 random sequences generated using the multinomial model with probabilities 1/7 for P, 2/7 for A, 1/7 for W, 1/7 for H and 2/7 for E.

For example, to align "HEAGAWGHEE" to the first of the 1000 random sequences ("EEHAAAE"), we type:

```
s4 <- "HEAGAWGHEE"
```

```
Biostrings::pairwiseAlignment(s4, randomseqs[1],
                              substitutionMatrix = "BLOSUM50",
                              gapOpening = -2,
                              gapExtension = -8,
                              scoreOnly = FALSE)
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: HEAGAWGHEE
## subject: -EAEE--HEE
## score: 1
```

If we use the `pairwiseAlignment()` function with the argument `scoreOnly=TRUE`, it will just give us the score for the alignment:

```
pairwiseAlignment(s4, randomseqs[1],
                  substitutionMatrix = "BLOSUM50",
                  gapOpening = -2,
                  gapExtension = -8,
                  scoreOnly = TRUE)
```

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

If we repeat this 1000 times, that is, for each of the 1000 random sequences in vector randomseqs, we can get a distribution of alignment scores expected for aligning "HEAGAWGHEE" to random sequences of the same length and (approximately the same) amino acid composition as "PAWHEAE".

We can then compare the actual score for aligning "PAWHEAE" to "HEA-GAWGHEE" (i.e. -5) to the distribution of scores for aligning "HEAGAWGHEE" to the random sequences.

The code above first uses the **double()** function to create a numeric vector **randomscores** for storing real numbers (i.e. not integers), with 1000 elements. This will be used to store the alignment scores for 1000 alignments between "HEAGAWGHEE" and the 1000 different random sequences generated using the multinomial model.

The for loop takes each of the 1000 different random sequences, aligns each one to "HEAGAWGHEE", and stores the 1000 alignment scores in the **randomscores** vector.

```r
randomscores <- double(1000) # Create a numeric vector with 1000 elements
for (i in 1:1000)
{
  score <- Biostrings::pairwiseAlignment(s4, randomseqs[i],
                                         substitutionMatrix = "BLOSUM50",
                                         gapOpening = -2,
                                         gapExtension = -8,
                                         scoreOnly = TRUE)
                                         randomscores[i] <- score
}
```

Once we have run the for loop, we can make a histogram plot of the 1000 scores in vector randomscores by typing:

```r
hist(randomscores, col="red") # Draw a red histogram
```

## Histogram of randomscores
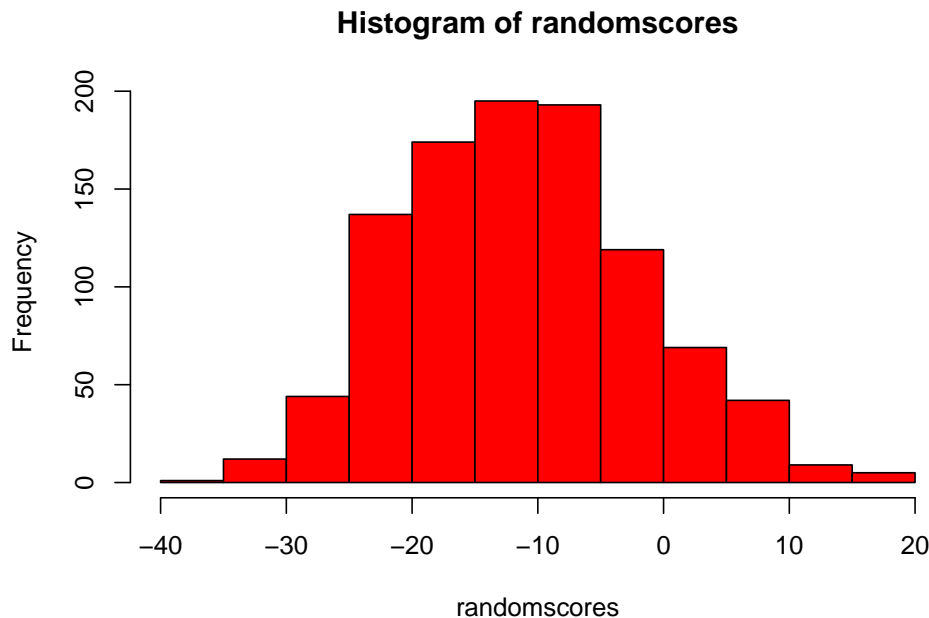


We can see from the histogram that quite a lot of the random sequences seem to have higher alignment scores than -5 when aligned to "HEAGAWGHEE" (where -5 is the alignment score for "PAWHEAE" and "HEAGAWGHEE").

We can use the vector randomscores of scores for 1000 alignments of random sequences to "HEAGAWGHEE" to calculate the probability of getting a score as large as the real alignment score for "PAWHEAE" and "HEAGAWGHEE" (i.e. -5) by chance.

```
sum(randomscores >= -5)
```

```
## [1] 284
```

We see that 266 of the 1000 alignments of random sequences to "HEA-GAWGHEE" had alignment scores that were equal to or greater than -5. Thus, we can estimate that the probability of getting a score as large as the real alignment score by chance is (266/1000 =) 0.266. In other words, we can calculate a P-value of 0.266. This probability or P-value is quite high (almost 30%, or 1 in 3), so we can conclude that it is quite probable that we could get an alignment score as high as -5 by chance alone. This indicates that the sequences "HEAGAWGHEE" and "PAWHEAE" are not more similar than any two random sequences, and so they are probably not related sequences.

Another way of saying this is that the P-value that we calculated is high (0.266), and as a result we conclude that the alignment score for the sequences "HEA-GAWGHEE" and "PAWHEAE" is not statistically significant. Generally, if the P-value that we calculate for an alignment of two sequences is >0.05, we conclude that the alignment score is not statistically significant, and that the

sequences are probably not related. On the other hand, if the P-value is less than or equal to 0.05, we conclude that the alignment score is "statistically significant", and the sequences are very probably related (homologous).

## 34.2 Summary

In this practical, you will have learned to use the following functions:

- `data()` for reading in data that comes with an *R* package
- `double()` for creating a numeric vector for storing real (non-integer) numbers
- `toupper()` for converting a string of characters from lowercase to uppercase

All of these functions belong to the standard installation of R.

You have also learned the following R functions that belong to bioinformatics packages:

- `nucleotideSubstitutionMatrix()` in the Biostrings package for making a nucleotide scoring matrix
- `pairwiseAlignment()` in the Biostrings package for making a global alignment between two sequences
- `c2s()` in the seqinr package for converting a sequence stored in a vector to a string of characters

## 34.3 Links and Further Reading

Some links are included here for further reading.

For background reading on sequence alignment, it is recommended to read Chapter 3 of Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

For more in-depth information and more examples on using the SeqinR package for sequence analysis, look at the SeqinR documentation, http://pbil.univ-lyon1.fr/software/seqinr/doc.php?lang=eng.

There is also a very nice chapter on "Analyzing Sequences", which includes examples of using SeqinR and Biostrings for sequence analysis, as well as details on how to implement algorithms such as Needleman-Wunsch and Smith-Waterman in R yourself, in the book Applied statistics for bioinformatics using R by Krijnen (available online at cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf).

For a more in-depth introduction to R, a good online tutorial is available on the "Kickstarting R" website, cran.r-project.org/doc/contrib/Lemon-kickstart.

There is another nice (slightly more in-depth) tutorial to R available on the "Introduction to R" website, cran.r-project.org/doc/manuals/R-intro.html.

For more information on and examples using the Biostrings package, see the Biostrings documentation at http://www.bioconductor.org/packages/release/bioc/html/Biostrings.html.

The examples of DNA sequences and protein sequences to align ('GAATTC' and 'GATTA', and sequences "PAWHEAE" and "HEAGAWGHEE"), as well as some ideas related to finding the statistical significance of a pairwise alignment, were inspired by the chapter on "Analyzing Sequences" in the book Applied statistics for bioinformatics using R by Krijnen (cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf).

## 34.4 Exercises

Answer the following questions using *R*. For each question, please record your answer, and what you typed into R to get this answer.

Model answers to the exercises are given in Answers to the exercises on Sequence Alignment.

1. Download FASTA-format files of the **Brugia malayi** Vab-3 protein (UniProt accession A8PZ80) and the *Loa loa* Vab-3 protein (UniProt accession E1FTG0) sequences from UniProt. Note: the vab-3 gene of *Brugia malayi* and the vab-3 gene of *Loa loa* are related genes that control eye development in these two species. *Brugia malayi* and *Loa loa* are both parasitic nematode worms, which both cause filariasis, which is classified by the WHO as a neglected tropical disease.

2. What is the alignment score for the optimal global alignment between the *Brugia malayi* Vab-3 protein and the *Loa loa* Vab-3 protein, when you use the BLOSUM50 scoring matrix, a gap opening penalty of -10 and a gap extension penalty of -0.5? Note: to specify a gap opening penalty of -10 and a gap extension penalty of -0.5, set the `gapOpening` argument to -9.5, and the `gapExtension` penalty to -0.5 in the `pairwiseAlignment()` function.

3. Use the print_pairwise_alignment() function to view the optimal global alignment between **Brugia malayi** Vab-3 protein and the *Loa loa* Vab-3 protein, using the BLOSUM50 scoring matrix, a gap opening penalty of -10 and a gap extension penalty of -0.5. Do you see any regions where the alignment is very good (lots of identities and few gaps)?

4. What global alignment score do you get for the two Vab-3 proteins, when you use the BLOSUM62 alignment matrix, a gap opening penalty of -10 and a gap extension penalty of -0.5? Which scoring matrix do you think is more appropriate for using for this pair of proteins: BLOSUM50 or BLOSUM62?

5. What is the statistical significance of the optimal global alignment for the

**Brugia malayi** and **Loa loa** Vab-3 proteins made using the BLOSUM50 scoring matrix, with a gap opening penalty of -10 and a gap extension penalty of -0.5? In other words, what is the probability of getting a score as large as the real alignment score for Vab-3 by chance?

6. What is the optimal global alignment score between the *Brugia malayi* Vab-6 protein and the *Mycobacterium leprae* chorismate lyase protein? Is the alignment score statistically significant (what is the P- value?)? Does this surprise you?

# Chapter 35

# A complete bioinformatics workflow in R

**By**: Nathan L. Brouwer

# Chapter 36

# "Worked example: Building a phylogeny in R"

## 36.1 Introduction

Phylogenies play an important role in computational biology and bioinformatics. Phylogenetics itself is an obligatly computational field that only began rapid growth when computational power allowed the many algorithms it relies on to be done rapidly. Phylogeneies of species, genes and proteins are used to address many biological issues, including

- Patterns of protein evolution
- Origin and evolution of phenotypic traits
- Origin and progression of epidemics
- Origin of evolution of diseases (eg, zooenoses)
- Prediction of protein function from its sequence
- ... and many more

The actual building of a phylogeny is a computationally intensive task; moreover, there are many bioinformatics and computational tasks the precede the construction of a phylogeny:

- genome sequencing and assembly
- computational gene prediction and annotation
- database searching and results screening
- pairwise sequence alignment
- data organization and cleaning
- multiple sequence alignment
- evaluation and validation of alignment accuracy

Once all of these steps have been carried out, the building of a phylogeny involves

- picking a model of sequence evolution or other description of evolution
- picking a statistical approach to tree construction
- evaluationg uncertainty in the final tree

In this chapter we will work through many of these steps. In most cases we will pick the easiest or fastest option; in later chapters we will unpack the various options. This chapter is written as an interactive R sessions. You can follow along by opening the .Rmd file of the chapter or typing the appropriate commands into your own script. I assume that all the necessary packages have been installed and they only need to be loaded into R using the *library()* command.

This lesson walks you through and entire workflow for a bioinformatics, including

1. obtaining FASTA sequences
2. cleaning sequences
3. creating alignments
4. creating distance a distance matrix
5. building a phylogenetic tree

We'll examine the Shroom family of genes, which produces Shroom proteins essential for tissue formation in many multicellular eukaryotes, including neural tube formation in vertebrates. We'll examine shroom in severall very different organism, including humans, mice and sea urchins. There is more than one type of shroom in vertebrates, and we'll also look at two different Shroom genes: shroom 1 and shroom 2.

This lesson draws on skills from previous sections of the book, but is written to act as a independent summary of these activities. There is therefore review of key aspects of R and bioinformatics throughout it.

## 36.2   Software Preliminaires

### 36.2.1   Vocab

- arguement
- function
- list
- named list
- vector
- named vector
- for() loop
- R console

### 36.2.2   R functions

- library()
- round()
- plot()

- mtext()
- nchar()
- rentrez::entrez_fetch()
- combio4all::entrez_fetch_list()
- combio4aal::print_msa() (Coghlan 2011)
- Biostrings::AAStringSet()
- msa::msa()
- msa::msaConvert()
- msa::msaPrettyPrint()
- seqinr::dist.alignment()
- ape::nj()

A few things need to be done to get started with our R session.

### 36.2.3  Download necessary packages

Many *R* sessions begin by downloading necessary software packages to augment *R's* functionality.

If you don't have them already, you'll need the following packages from CRAN:

1. `ape`
2. `seqinr`
3. `rentrez`
4. `devtools`

The CRAN packages can be loaded with `install.packages()`.

You'll also need these packages from Bioconductor:

1. `msa`
2. `Biostrings`

For install packages from Bioconductor, see the chapter at the beginning of this book on this process.

Finally, you'll need this package from GitHub

1. `compbio4all`

To install packages from GitHub you can use the code `devtools::install_github("brouwern/combio4all")`

### 36.2.4  Load packages into memory

We noew need to load up all our bioinformatics and phylogenetics software into R. This is done with the `library()` command.

To run this code just clock on the sideways green triangel all the way to the right of the code.

NOTE: You'll likely see some red code appear on your screen. No worries, totally normal!

```
# github packages
library(compbio4all)

# CRAN packages
library(rentrez)
library(seqinr)
library(ape)

# Bioconductor packages
library(msa)
library(Biostrings)
```

## 36.3 Downloading macro-molecular sequences

We're going to explore some sequences. First we need to download them. To do this we'll use a function, `entrez_fretch()`, which accesses the **Entrez** system of database (ncbi.nlm.nih.gov/search/). This function is from the `rentrez` package, which stands for "R-Entrez."

We need to tell `entrez_fetch()` several things

1. `db = ...` the type of entrez database.

2. `id = ...` the **accession** (ID) number of the sequence
3. `rettype = ...` file type what we want the function to return.

Formally, these things are called **arguements** by *R*.

We'll use these settings:

1. `db = "protein"` to access the Entrez database of protein sequences

2. `rettype = "fasta"`, which is a standard file format for nucleic acid and protein sequences

We'll set `id = ...` to sequences whose **accession numbers** are:

1. NP_065910: Human shroom 3
2. AAF13269: Mouse shroom 3a
3. CAA58534: Human shroom 2
4. XP_783573: Sea urchin shroom

There are two highly conserved regions of shroom 3 1. ASD 1: aa 884 to aa 1062 in hShroom3 1. ASD 2: aa 1671 to aa 1955 in hShroom3

Normally we'd have to download these sequences by hand through pointing and clicking on GeneBank records on the NCBI website. In *R* we can do it automatically; this might take a second.

All the code needed is this:

```r
# Human shroom 3 (H. sapiens)
hShroom3 <- entrez_fetch(db = "protein",
                         id = "NP_065910",
                         rettype = "fasta")
```

The output is in FASTA format; we'll use the `cat()` to do a little formating for us:

```r
cat(hShroom3)
```

```
## >NP_065910.3 protein Shroom3 [Homo sapiens]
## MMRTTEDFHKPSATLNSNTATKGRYIYLEAFLEGGAPWGFTLKGGLEHGEPLIISKVEEGGKADTLSSKL
## QAGDEVVHINEVTLSSSRKEAVSLVKGSYKTLRLVVRRDVCTDPGHADTGASNFVSPEHLTSGPQHRKAA
## WSGGVKLRLKHRRSEPAGRPHSWHTTKSGEKQPDASMMQISQGMIGPPWHQSYHSSSSTSDLSNYDHAYL
## RRSPDQCSSQGSMESLEPSGAYPPCHLSPAKSTGSIDQLSHFHNKRDSAYSSFSTSSSILEYPHPGISGR
## ERSGSMDNTSARGGLLEGMRQADIRYVKTVYDTRRGVSAEYEVNSSALLLQGREARASANGQGYDKWSNI
## PRGKGVPPPSWSQQCPSSLETATDNLPPKVGAPLPPARSDSYAAFRHRERPSSWSSLDQKRLCRPQANSL
## GSLKSPFIEEQLHTVLEKSPENSPPVKPKHNYTQKAQPGQPLLPTSIYPVPSLEPHFAQVPQPSVSSNGM
## LYPALAKESGYIAPQGACNKMATIDENGNQNGSGRPGFAFCQPLEHDLLSPVEKKPEATAKYVPSKVHFC
## SVPENEEDASLKRHLTPPQGNSPHSNERKSTHSNKPSSHPHSLKCPQAQAWQAGEDKRSSRLSEPWEGDF
## QEDHNANLWRRLEREGLGQSLSGNFGKTKSAFSSLQNIPESLRRHSSLELGRGTQEGYPGGRPTCAVNTK
## AEDPGRKAAPDLGSHLDRQVSYPRPEGRTGASASFNSTDPSPEEPPAPSHPHTSSLGRRGPGPGSASALQ
## GFQYGKPHCSVLEKVSKFEQREQGSQRPSVGGSGFGHNYRPHRTVSTSSTSGNDFEETKAHIRFSESAEP
## LGNGEQHFKNGELKLEEASRQPCGQQLSGGASDSGRGPQRPDARLLRSQSTFQLSSEPEREPEWRDRPGS
## PESPLLDAPFSRAYRNSIKDAQSRVLGATSFRRRDLELGAPVASRSWRPRPSSAHVGLRSPEASASASPH
## TPRERHSVTPAEGDLARPVPPAARRGARRRLTPEQKKRSYSEPEKMNEVGIVEEAEPAPLGPQRNGMRFP
## ESSVADRRRLFERDGKACSTLSLSGPELKQFQQSALADYIQRKTGKRPTSAAGCSLQEPGPLRERAQSAY
## LQPGPAALEGSGLASASSLSSLREPSLQPRREATLLPATVAETQQAPRDRSSSFAGGRRLGERRRGDLLS
## GANGGTRGTQRGDETPREPSSWGARAGKSMSAEDLLERSDVLAGPVHVRSRSSPATADKRQDVLLGQDSG
## FGLVKDPCYLAGPGSRSLSCSERGQEEMLPLFHHLTPRWGGSGCKAIGDSSVPSECPGTLDHQRQASRTP
## CPRPPLAGTQGLVTDTRAAPLTPIGTPLPSAIPSGYCSQDGQTGRQPLPPYTPAMMHRSNGHTLTQPPGP
## RGCEGDGPEHGVEEGTRKRVSLPQWPPPSRAKWAHAAREDSLPEESSAPDFANLKHYQKQQSLPSLCSTS
## DPDTPLGAPSTPGRISLRISESVLRDSPPPHEDYEDEVFVRDPHPKATSSPTFEPLPPPPPPPPSQETPV
## YSMDDFPPPPPHTVCEAQLDSEDPEGPRPSFNKLSKVTIARERHMPGAAHVVGSQTLASRLQTSIKGSEA
## ESTPPSFMSVHAQLAGSLGGQPAPIQTQSLSHDPVSGTQGLEKKVSPDPQKSSEDIRTEALAKEIVHQDK
## SLADILDPDSRLKTTMDLMEGLFPRDVNLLKENSVKRKAIQRTVSSSGCEGKRNEDKEAVSMLVNCPAYY
## SVSAPKAELLNKIKEMPAEVNEEEEQADVNEKKAELIGSLTHKLETLQEAKGSLLTDIKLNNALGEEVEA
## LISELCKPNEFDKYRMFIGDLDKVVNLLLSLSGRLARVENVLSGLGEDASNEERSSLYEKRKILAGQHED
## ARELKENLDRRERVVLGILANYLSEEQLQDYQHFVKMKSTLLIEQRKLDDKIKLGQEQVKCLLESLPSDF
## IPKAGALALPPNLTSEPIPAGGCTFSGIFPTLTSPL
```

Note the initial `>`, then the header line of `NP_065910.3 protein Shroom3 [Homo sapiens]`. After that is the amino acid sequence. The underlying data also includes the **newline character \n** to designate where each line of amino acids stops.

We can get the rest of the data by just chaing the `id = ...` argument:

```
# Mouse shroom 3a (M. musculus)
mShroom3a <- entrez_fetch(db = "protein",
                          id = "AAF13269",
                          rettype = "fasta")


# Human shroom 2 (H. sapiens)
hShroom2 <- entrez_fetch(db = "protein",
                         id = "CAA58534",
                         rettype = "fasta")



# Sea-urchin shroom
sShroom <- entrez_fetch(db = "protein",
                        id = "XP_783573",
                        rettype = "fasta")
```

I'm going to check about how long each of these sequences is - each should have an at least slightly different length. If any are identical, I might have repeated an accession name or re-used an object name. The function `nchar()` counts of the number of characters in an *R* object.

```
nchar(hShroom3)
```

```
## [1] 2070
```

```
nchar(mShroom3a)
```

```
## [1] 2083
```

```
nchar(sShroom)
```

```
## [1] 1758
```

```
nchar(hShroom2)
```

```
## [1] 1673
```

## 36.4   Prepping macromolecular sequences

> "90% of data analysis is data cleaning" (-Just about every data analyst and data scientist on twitter)

We have our sequences, but the current format isn't directly usable for us yet because there are several thigns that aren't sequence information

1. metadata (the header)
2. page formatting information (the newline character)

We can remove this non-sequence information using a function I wrote called `fasta_cleaner()`, which is in the `compbio4all` package. The function uses **regular expressions** to remove the info we don't need.

**ASIDE**: If we run the name of the command with out any quotation marks we can see the code:

```
fasta_cleaner
```

```
## function(fasta_object, parse = TRUE){
##
##   fasta_object <- sub("^(>)(.*?)(\\n)(.*)(\\n\\n)","\\4",fasta_object)
##   fasta_object <- gsub("\n", "", fasta_object)
##
##   if(parse == TRUE){
##     fasta_object <- stringr::str_split(fasta_object,
##                                        pattern = "",
##                                        simplify = FALSE)
##   }
##
##   return(fasta_object[[1]])
## }
## <bytecode: 0x7f822ac4bea0>
## <environment: namespace:compbio4all>
```

**End ASIDE**

Now use the function to clean our sequences; we won't worry about what `pare = ...` is for.

```
hShroom3  <- fasta_cleaner(hShroom3,  parse = F)
mShroom3a <- fasta_cleaner(mShroom3a, parse = F)
hShroom2  <- fasta_cleaner(hShroom2,  parse = F)
sShroom   <- fasta_cleaner(sShroom,   parse = F)
```

Now let's take a peek at what our sequences look like:

```
hShroom3
```

```
## [1] "MMRTTEDFHKPSATLNSNTATKGRYIYLEAFLEGGAPWGFTLKGGLEHGEPLIISKVEEGGKADTLSSKLQAGDEVVHINEVTLSSSRK
```

## 36.5  Aligning sequences

We can do a **global alignment** of one sequence against another using the `pairwiseAlignment()` function from the **Bioconductor** package `Biostrings` (note that capital "B" in `Biostrings`; most *R* package names are all lower case, but not this one).

Let's align human versus mouse shroom:

```
align.h3.vs.m3a <- Biostrings::pairwiseAlignment(
                   hShroom3,
                   mShroom3a)
```

We can peek at the alignment

```
align.h3.vs.m3a
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: MMRTTEDFHKPSATLN-SNTATKGRYIYLEAFLE...KAGALALPPNLTSEPIPAGGCTFSGIFPTLTSPL
## subject: MK-TPENLEEPSATPNPSRTPTE-RFVYLEALLE...KAGAISLPPALTGHATPGGTSVFGGVFPTLTSPL
## score: 2189.934
```

The **score** tells us how closely they are aligned; higher scores mean the sequences
are more similar. Its hard to interpret the number on its own so we can get the
**percent sequence identity (PID)** using the `pid()` function.

```
Biostrings::pid(align.h3.vs.m3a)
```

```
## [1] 70.56511
```

So, *shroom3* from humans and *shroom3* from mice are ~71% similar (at least
using this particular method of alignment, and there are many ways to do this!)

What about human shroom 3 and sea-urchin shroom?

```
align.h3.vs.h2 <- Biostrings::pairwiseAlignment(
                   hShroom3,
                   hShroom2)
```

First check out the score using `score()`, which accesses it directly without all
the other information.

```
score(align.h3.vs.h2)
```

```
## [1] -5673.853
```

Now the percent sequence alignment with `pid()`:

```
Biostrings::pid(align.h3.vs.h2)
```

```
## [1] 33.83277
```

So Human shroom 3 and Mouse shroom 3 are 71% identical, but Human shroom
3 and human shroom 2 are only 34% similar? How does it work out evolutionary
that a human and mouse gene are more similar than a human and a human gene?
What are the evolutionary relationships among these genes within the shroom
gene family?

## 36.6  The shroom family of genes

I've copied a table from a published paper which has accession numbers for 15 different Shroom genes.

```r
shroom_table <- c("CAA78718" , "X. laevis Apx" ,         "xShroom1",
           "NP_597713" , "H. sapiens APXL2" ,      "hShroom1",
           "CAA58534" , "H. sapiens APXL",         "hShroom2",
           "ABD19518" , "M. musculus Apxl" ,       "mShroom2",
           "AAF13269" , "M. musculus ShroomL" ,    "mShroom3a",
           "AAF13270" , "M. musculus ShroomS" ,    "mShroom3b",
           "NP_065910", "H. sapiens Shroom" ,      "hShroom3",
           "ABD59319" , "X. laevis Shroom-like",   "xShroom3",
           "NP_065768", "H. sapiens KIAA1202" ,    "hShroom4a",
           "AAK95579" , "H. sapiens SHAP-A" ,      "hShroom4b",
           #"DQ435686" , "M. musculus KIAA1202" ,   "mShroom4",
           "ABA81834" , "D. melanogaster Shroom", "dmShroom",
           "EAA12598" , "A. gambiae Shroom",       "agShroom",
           "XP_392427" , "A. mellifera Shroom" ,   "amShroom",
           "XP_783573" , "S. purpuratus Shroom" , "spShroom") #sea urchin
```

I'll do a bit of formatting; you can ignore these details if you want

```r
# convert to matrix
shroom_table_matrix <- matrix(shroom_table,
                              byrow = T,
                              nrow = 14)
# convert to dataframe
shroom_table <- data.frame(shroom_table_matrix,
                  stringsAsFactors = F)

# name columns
names(shroom_table) <- c("accession", "name.orig","name.new")

# Create simplified species names
shroom_table$spp <- "Homo"
shroom_table$spp[grep("laevis",shroom_table$name.orig)] <- "Xenopus"
shroom_table$spp[grep("musculus",shroom_table$name.orig)] <- "Mus"
shroom_table$spp[grep("melanogaster",shroom_table$name.orig)] <- "Drosophila"
shroom_table$spp[grep("gambiae",shroom_table$name.orig)] <- "mosquito"
shroom_table$spp[grep("mellifera",shroom_table$name.orig)] <- "bee"
shroom_table$spp[grep("purpuratus",shroom_table$name.orig)] <- "sea urchin"
```

Take a look:

```r
shroom_table
```

```
##      accession                name.orig  name.new         spp
```

```
## 1   CAA78718        X. laevis Apx   xShroom1    Xenopus
## 2  NP_597713      H. sapiens APXL2  hShroom1       Homo
## 3   CAA58534       H. sapiens APXL  hShroom2       Homo
## 4   ABD19518       M. musculus Apxl mShroom2        Mus
## 5   AAF13269   M. musculus ShroomL mShroom3a        Mus
## 6   AAF13270   M. musculus ShroomS mShroom3b        Mus
## 7  NP_065910      H. sapiens Shroom  hShroom3       Homo
## 8   ABD59319 X. laevis Shroom-like  xShroom3    Xenopus
## 9  NP_065768   H. sapiens KIAA1202 hShroom4a       Homo
## 10  AAK95579     H. sapiens SHAP-A hShroom4b       Homo
## 11  ABA81834 D. melanogaster Shroom  dmShroom Drosophila
## 12  EAA12598      A. gambiae Shroom  agShroom   mosquito
## 13 XP_392427    A. mellifera Shroom  amShroom        bee
## 14 XP_783573   S. purpuratus Shroom  spShroom sea urchin
```

## 36.7   Downloading multiple sequences

Instead of getting one sequence at a time we can download several by accessing the "accession" column from the table

```
shroom_table$accession
```

```
##  [1] "CAA78718"  "NP_597713" "CAA58534"  "ABD19518"  "AAF13269"  "AAF13270"
##  [7] "NP_065910" "ABD59319"  "NP_065768" "AAK95579"  "ABA81834"  "EAA12598"
## [13] "XP_392427" "XP_783573"
```

We can give this whole set of accessions to `entrez_fetch()`:

```
shrooms <- entrez_fetch(db = "protein",
                        id = shroom_table$accession,
                        rettype = "fasta")
```

We can look at what we got here with `cat()` (I won't display this because it is very long!)

```
cat(shrooms)
```

The current format of these data is a single, long set of data. This is a standard way to store, share and transmit FASTA files, but in *R* we'll need a slightly different format.

We'll download all of the sequences again, this time using a function from `compbio4all` called `entrez_fetch_list()` which is a **wrapper** function I wrote to put the output of `entrez_fetch()` into an *R* data format called a **list**.

```
shrooms_list <- entrez_fetch_list(db = "protein",
                        id = shroom_table$accession,
                        rettype = "fasta")
```

Now we have a list which as 14 **elements**, one for each sequence in our table.

```
length(shrooms_list)
```

```
## [1] 14
```

We now need to clean up each one fo these sequences. We can do that using a simple `for()` loop:

```
for(i in 1:length(shrooms_list)){
  shrooms_list[[i]] <- fasta_cleaner(shrooms_list[[i]], parse = F)
}
```

Second to last step: we need to take each one of our sequences from our list and put it into a **vector**, in particular a **named vector**

```
# make a vector to store output
shrooms_vector <- rep(NA, length(shrooms_list))

# run the loop
for(i in 1:length(shrooms_vector)){
  shrooms_vector[i] <- shrooms_list[[i]]
}

# name the vector
names(shrooms_vector) <- names(shrooms_list)
```

Now the final step: we need to convert our named vector to a **string set** using `Biostrings::AAStringSet()`. Note the `_ss` tag at the end of the object we're assigning the output to, which designates this as a string set.

```
shrooms_vector_ss <- Biostrings::AAStringSet(shrooms_vector)
```

## 36.8 Multiple sequence alignment

We must **align** all of the sequences we downloaded and use that **alignment** to build a **phylogenetic tree**. This will tell us how the different genes, both within and between species, are likely to be related.

### 36.8.1 Building an Multiple Sequence Alignment (MSA)

We'll use the software `msa,` which implements the **ClustalW** multiple sequence alignment algorithm. Normally we'd have to download the ClustalW program and either point-and-click our way through it or use the **command line**\*, but these folks wrote up the algorithm in R so we can do this with a line of R code. This will take a second or two.

```
shrooms_align <- msa(shrooms_vector_ss,
                     method = "ClustalW")
```

```
## use default substitution matrix
```

## 36.8.2 Viewing an MSA

Once we build an MSA we need to visualize it.

### 36.8.2.1 Viewing an MSA in R

We can look at the output from `msa()`, but its not very helpful

```
shrooms_align
```

```
## CLUSTAL 2.1
##
## Call:
##    msa(shrooms_vector_ss, method = "ClustalW")
##
## MsaAAMultipleAlignment with 14 rows and 2252 columns
##        aln                                                   names
##   [1] ------------------------...------------------------ NP_065768
##   [2] ------------------------...------------------------ AAK95579
##   [3] ------------------------...SVFGGVFPTLTSPL----------- AAF13269
##   [4] ------------------------...SVFGGVFPTLTSPL----------- AAF13270
##   [5] ------------------------...CTFSGIFPTLTSPL----------- NP_065910
##   [6] ------------------------...NKS--LPPPLTSSL----------- ABD59319
##   [7] ------------------------...------------------------ CAA58534
##   [8] ------------------------...------------------------ ABD19518
##   [9] ------------------------...LT---------------------- NP_597713
##  [10] ------------------------...------------------------ CAA78718
##  [11] ------------------------...------------------------ EAA12598
##  [12] ------------------------...------------------------ ABA81834
##  [13] MTELQPSPPGYRVQDEAPGPPSCPP...------------------------ XP_392427
##  [14] ------------------------...AATSSSSNGIGGPEQLNSNATSSYC XP_783573
##  Con ------------------------...------------------------ Consensus
```

A function called `print_msa()` (Coghlan 2011) which I've put into`combio4all` can give us more informative output by printing out the actual alignment into the R console.

To use `print_msa()` We need to make a few minor tweaks though first. These are behind the scenes changes so don't worry about the details right now. We'll change the name to `shrooms_align_seqinr` to indcate that one of our changes is putting this into a format defined by the bioinformatics package `seqinr`.

```
class(shrooms_align) <- "AAMultipleAlignment"
shrooms_align_seqinr <- msaConvert(shrooms_align, type = "seqinr::alignment")
```

I won't display the output from `shrooms_align_seqinr` because its very long; we have 14 shroom genes, and shroom happens to be a rather long gene.

```
print_msa(alignment = shrooms_align_seqinr,
          chunksize = 60)
```

### 36.8.2.2 Displaying an MSA as an R plot

I'm going to just show about 100 amino acids near the end of the alignment, where there is the most overlap across all of the sequences. This is set with the `start = ...` and `end = ...` arguments. Note that we're using the `shrooms_align` object.

```
ggmsa::ggmsa(shrooms_align,     # shrooms_align, NOT shrooms_align_seqinr
     start = 2000,
     end = 2100)
```
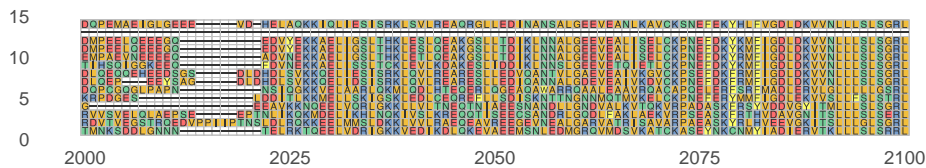


### 36.8.2.3 Saving an MSA as PDF

We can take a look at the alignment in PDF format if we want. I this case I'm going to just show about 100 amino acids near the end of the alignment, where there is the most overlap across all of the sequences. This is set with the `y = c(...)` argument.

```
msaPrettyPrint(shrooms_align,              # alignment
               file = "shroom_msa.pdf",    # file name
               y=c(2000, 2100),            # range
               askForOverwrite=FALSE)
```

You can see where R is saving thigns by running `getwd()`

```
getwd()
```

```
## [1] "/Users/nlb24/google_backup_sync_nlb24/lbrb"
```

On a Mac you can usually find the file by searching in Finder for the file name, which I set to be "shroom_msa.pdf" using the `file = ...` arguement above.

## 36.9 Genetic distance.

Next need to first get an estimate of how similar each sequences is. The more amino acids that are identical to each other, the more similar.

Instead of similarity, we usually work in terms of *difference* or **genetic distance** (a.k.a. **evolutionary distance**). This is done with the `dist.alignment()` function.

```
shrooms_dist <- seqinr::dist.alignment(shrooms_align_seqinr,
                                       matrix = "identity")
```

We've made a matrix using `dist.alignment()`; let's round it off so its easier to look at using the `round()` function.

```
shrooms_dist_rounded <- round(shrooms_dist,
                              digits = 3)
```

Now let's look at it

```
shrooms_dist_rounded
```

```
##           NP_065768 AAK95579 AAF13269 AAF13270 NP_065910 ABD59319 CAA58534
## AAK95579      0.000
## AAF13269      0.884    0.917
## AAF13270      0.897    0.917    0.000
## NP_065910     0.878    0.912    0.533    0.536
## ABD59319      0.893    0.921    0.783    0.783     0.782
## CAA58534      0.872    0.908    0.838    0.849     0.840    0.864
## ABD19518      0.866    0.912    0.834    0.846     0.838    0.855    0.548
## NP_597713     0.916    0.939    0.903    0.903     0.902    0.904    0.896
## CAA78718      0.925    0.955    0.896    0.895     0.893    0.893    0.898
## EAA12598      0.914    0.947    0.899    0.899     0.902    0.897    0.891
## ABA81834      0.938    0.943    0.935    0.934     0.936    0.940    0.935
## XP_392427     0.936    0.963    0.935    0.934     0.938    0.941    0.938
## XP_783573     0.940    0.958    0.942    0.939     0.942    0.935    0.942
##           ABD19518 NP_597713 CAA78718 EAA12598 ABA81834 XP_392427
## AAK95579
## AAF13269
## AAF13270
## NP_065910
## ABD59319
## CAA58534
## ABD19518
## NP_597713     0.900
## CAA78718      0.891    0.919
## EAA12598      0.896    0.920    0.922
## ABA81834      0.935    0.932    0.946    0.882
## XP_392427     0.934    0.927    0.947    0.878     0.923
## XP_783573     0.946    0.947    0.941    0.925     0.954    0.943
```

## 36.10   Phylognetic trees (finally!)

We got our sequence, built multiple sequence alignment, and calcualted the genetic distance between sequences. Now we are - finally - ready to build a

phylogenetic tree.

First, we let R figure out the structure of the tree. There are **MANY** ways to build phylogenetic trees. We'll use a common one used for exploring sequences called **neighbor joining** algorithm via the function `nj()`. Neighbor joining uses genetic distances to cluster sequences into **clades**.

```
tree <- nj(shrooms_dist)
```

### 36.10.1   Plotting phylogenetic trees

Now we'll make a quick plot of our tree using `plot()` (and add a little label using a function called `mtext()`).

```
# plot tree
plot.phylo (tree, main="Phylogenetic Tree",
            type = "unrooted",
            use.edge.length = F)

# add label
mtext(text = "Shroom family gene tree - unrooted, no branch lengths")
```

## Phylogenetic Tree

Shroom family gene tree – unrooted, no branch lengths
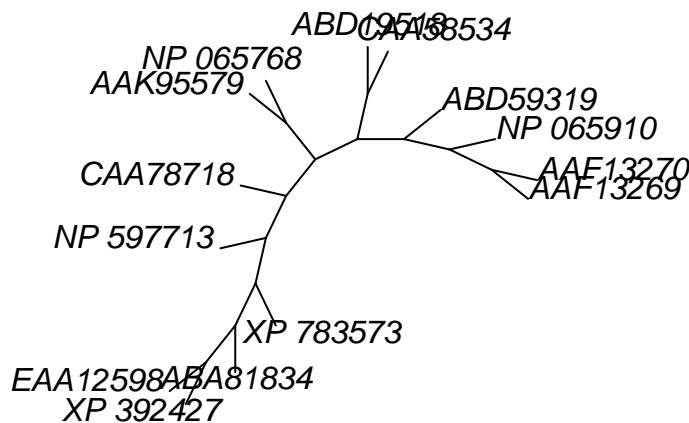


This is an **unrooted tree*. For the sake of plotting we've also ignored the evolutionary distance between the sequences.

To make a rooted tree we remove `type = "unrooted`.

```
# plot tree
plot.phylo (tree, main="Phylogenetic Tree",
            use.edge.length = F)
```

```
# add label
mtext(text = "Shroom family gene tree - rooted, no branch lenths")
```

## Phylogenetic Tree

Shroom family gene tree – rooted, no branch lenths
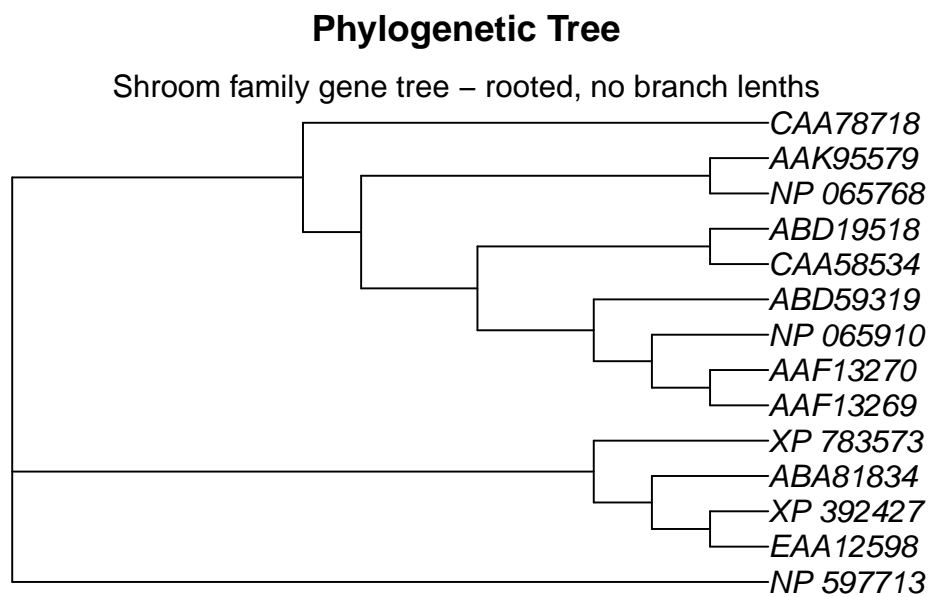


We can include information about branch length by setting `use.edge.length = ...` to `T`.

```
# plot tree
plot.phylo (tree, main="Phylogenetic Tree",
            use.edge.length = T)

# add label
mtext(text = "Shroom family gene tree - rooted, with branch lenths")
```

# Phylogenetic Tree

### Shroom family gene tree – rooted, with branch lenths



Some of the branches are now very short, but most are very long, indicating that these genes have been evolvin independently for many millions of years.

Let's make a fancier plot. Don't worry about all the steps; I've added some more code to add some annotations on the right-hand side to help us see what's going on.

```
plot(tree, main="Phylogenetic Tree")
mtext(text = "Shroom family gene tree")

x <- 0.551
x2 <- 0.6

# label Shrm 3
segments(x0 = x, y0 = 1,
         x1 = x, y1 = 4,
         lwd=2)
text(x = x*1.01, y = 2.5, "Shrm 3",adj = 0)

segments(x0 = x, y0 = 5,
         x1 = x, y1 = 6,
         lwd=2)
text(x = x*1.01, y = 5.5, "Shrm 2",adj = 0)

segments(x0 = x, y0 = 7,
         x1 = x, y1 = 9,
         lwd=2)
text(x = x*1.01, y = 8, "Shrm 1",adj = 0)
```

```
segments(x0 = x, y0 = 10,
         x1 = x, y1 = 13,
         lwd=2)
text(x = x*1.01, y = 12, "Shrm ?",adj = 0)


segments(x0 = x, y0 = 14,
         x1 = x, y1 = 15,
         lwd=2)
text(x = x*1.01, y = 14.5, "Shrm 4",adj = 0)


segments(x0 = x2, y0 = 1,
         x1 = x2, y1 = 6,
         lwd=2)

segments(x0 = x2, y0 = 7,
         x1 = x2, y1 = 9,
         lwd=2)

segments(x0 = x2, y0 = 10,
         x1 = x2, y1 = 15,
         lwd=2)
```

## Phylogenetic Tree

Shroom family gene tree
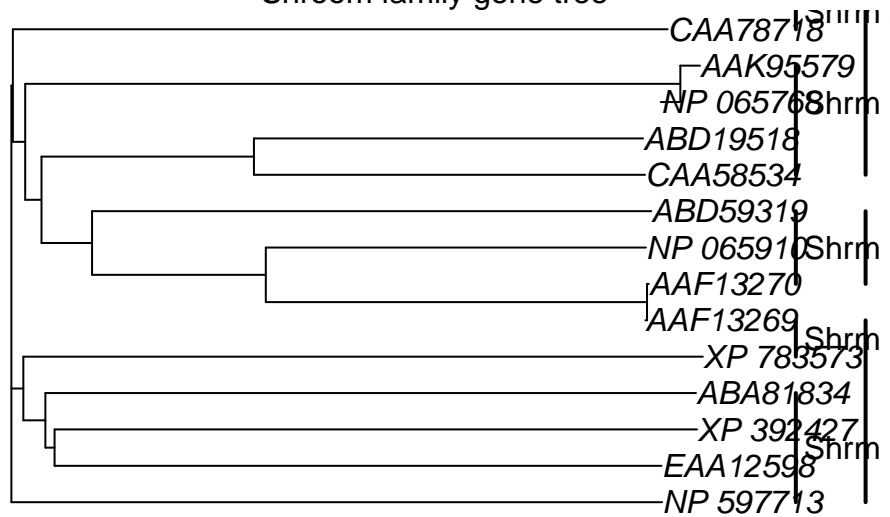
# Chapter 37

# Local variation in GC content

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

This is a modification of "DNA Sequence Statistics (1)" from Avril Coghlan's *A little book of R for bioinformatics.*. Almost all of text and code was originally written by Dr. Coghlan and distributed under the Creative Commons 3.0 license.

Preliminaries

```
library(rentrez)
library(seqinr)
library(compbio4all)
```

### 37.0.1 Note on the biology in this section

Some of the biology in this tutorial appears to be out of date. For example, using variation in GC content to ID horizontal gene transfer is currently considered to be biased. The examples are still good for practicing $R$ skills.

## 37.1 Vocabulary

- **GC content**
- **horizontal transfer**
- **local variation** in GC content ...

## 37.2   Reading sequence data with rentrez::entrez__fetch

In a previous section you learned how to use to search for and download the sequence data for a given NCBI accession from the NCBI Sequence Database, either via the NCBI website using `entrez_fetch()` from the rentrez package.

For example, you could have downloaded the sequence data for a the DEN-1 Dengue virus sequence (NCBI accession NC_001477), and stored it on a file on your computer (eg. `dengue_fasta.fasta`).

Download direct:

```
dengueseq_fasta <- entrez_fetch(db = "nucleotide",
                                id = "NC_001477",
                                rettype = "fasta")
```

From `combio4all` package

```
data(dengueseq_fasta)
```

As noted before, the file gets downloaded in FASTA format, which isn't directly useable in *R*

```
## [1] ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACAG
```

We can convert our FASTA object into a vector using the function fasta_cleaner()

```
header. <- ">NC_001477.1 Dengue virus 1, complete genome"
dengueseq_vector <- fasta_cleaner(dengueseq_fasta)
```

Once you have retrieved a sequence from the NCBI Sequence Database and stored it in a vector variable such as dengueseq_vector in the example above, it is possible to extract **subsequence** of the sequence by typing the name of the vector (eg. dengueseq_vector) followed by the square brackets containing the indices for those nucleotides. For example, to obtain nucleotides 452-535 of the DEN-1 Dengue virus genome, we can type:

```
dengueseq_vector[452:535]
```

```
##  [1] "C" "G" "A" "G" "G" "G" "G" "G" "A" "G" "A" "G" "C" "C" "G" "C" "A" "C" "A"
## [20] "T" "G" "A" "T" "A" "G" "T" "T" "A" "G" "C" "A" "A" "G" "C" "A" "G" "G" "A"
## [39] "A" "A" "G" "A" "G" "G" "A" "A" "A" "A" "T" "C" "A" "C" "T" "T" "T" "T" "G"
## [58] "T" "T" "T" "A" "A" "G" "A" "C" "C" "T" "C" "T" "G" "C" "A" "G" "G" "T" "G"
## [77] "T" "C" "A" "A" "C" "A" "T" "G"
```

## 37.3   Local variation in GC content

In a previous section, you learned that to find out the **GC content** of a genome sequence (percentage of nucleotides in a genome sequence that are Gs or Cs),

you can use the GC() function in the seqinr package. For example, to find the GC content of the DEN-1 Dengue virus sequence that we have stored in the vector dengueseq_vector, we can type:

```
seqinr::GC(dengueseq_vector)
```

```
## [1] 0.4666977
```

The output of the GC() is the fraction of nucleotides in a sequence that are Gs or Cs, so to convert it to a percentage we need to multiply by 100.

```
seqinr::GC(dengueseq_vector)*100
```

```
## [1] 46.66977
```

Thus, the GC content of the DEN-1 Dengue virus genome is about 0.467 or 46.7%.

Although the GC content of the whole DEN-1 Dengue virus genome sequence is about 46.7%, there is probably **local variation** in GC content within the genome. That is, some regions of the genome sequence may have GC contents quite a bit higher than 46.7%, while some regions of the genome sequence may have GC contents that are quite a big lower than 46.7%. Local fluctuations in GC content within the genome sequence can provide different interesting information, for example, they may reveal cases of **horizontal transfer** or reveal biases in mutation.

If a chunk of DNA has moved by horizontal transfer from the genome of a species with low GC content to a species with high GC content, the chunk of horizontally transferred DNA could be detected as a region of unusually low GC content in the high-GC recipient genome.

On the other hand, a region unusually low GC content in an otherwise high-GC content genome could also arise due to biases in mutation in that region of the genome, for example, if mutations from Gs/Cs to Ts/As are more common for some reason in that region of the genome than in the rest of the genome.

## 37.4 A sliding window analysis of GC content

In order to study local variation in GC content within a genome sequence, we could calculate the GC content for small chunks of the genome sequence. The DEN-1 Dengue virus genome sequence is 10735 nucleotides long. To study variation in GC content within the genome sequence, we could calculate the GC content of chunks of the DEN-1 Dengue virus genome, for example, for each 2000-nucleotide chunk of the genome sequence:

```
seqinr::GC(dengueseq_vector[1:2000])        # Calculate the GC content of nucleotides 1-2000 of the
```

```
## [1] 0.465
```

```r
seqinr::GC(dengueseq_vector[2001:4000])    # Calculate the GC content of nucleotides 20
```

```
## [1] 0.4525
```

From the output of the above calculations, we see that the region of the DEN-1 Dengue virus genome from nucleotides 1-2000 has a GC content of 46.5%, while the region of the Dengue genome from nucleotides 2001-4000 has a GC content of about 45.3%. Thus, there seems to be some local variation in GC content within the Dengue genome sequence.

Instead of typing in the commands above to tell R to calculate the GC content for each 2000-nucleotide chunk of the DEN-1 Dengue genome, we can use a for loop to carry out the same calculations, but by typing far fewer commands. That is, we can use a for loop to take each 2000-nucleotide chunk of the DEN-1 Dengue virus genome, and to calculate the GC content of each 2000-nucleotide chunk. Below we will explain the following for loop that has been written for this purpose:

```r
starts <- seq(1, length(dengueseq_vector)-2000, by = 2000)
starts
```

```
## [1]    1 2001 4001 6001 8001
```

```r
 n <- length(starts)    # Find the length of the vector "starts"
 for (i in 1:n) {
        chunk <- dengueseq_vector[starts[i]:(starts[i]+1999)]
        chunkGC <- GC(chunk)
        print (chunkGC)
    }
```

```
## [1] 0.465
## [1] 0.4525
## [1] 0.4705
## [1] 0.479
## [1] 0.4545
```

The command `starts <- seq(1, length(dengueseq_vector)-2000, by = 2000)` stores the result of the `seq()` command in the vector starts, which contains the values 1, 2001, 4001, 6001, and 8001.

We set the variable n to be equal to the number of elements in the vector starts, so it will be 5 here, since the vector starts contains the five elements 1, 2001, 4001, 6001 and 8001. The line "for (i in 1:n)" means that the counter i will take values of 1-5 in subsequent cycles of the for loop. The for loop above is spread over several lines. However, R will not execute the commands within the for loop until you have typed the final "}" at the end of the for loop and pressed "Return".

Each of the three commands within the for loop are carried out in each cycle of the loop. In the first cycle of the loop, i is 1, the vector variable chunk is

used to store the region from nucleotides 1-2000 of the Dengue virus sequence, the GC content of that region is calculated and stored in the variable chunkGC, and the value of chunkGC is printed out.

In the second cycle of the loop, i is 2, the vector variable chunk is used to store the region from nucleotides 2001-4000 of the Dengue virus sequence, the GC content of that region is calculated and stored in the variable `chunkGC`, and the value of `chunkGC` is printed out. The loop continues until the value of i is 5. In the fifth cycle through the loop, the value of i is 5, and so the GC content of the region from nucleotides 8001-10000 is printed out.

Note that we stop the loop when we are looking at the region from nucleotides 8001-10000, instead of continuing to another cycle of the loop where the region under examination would be from nucleotides 10001-12000. The reason for this is because the length of the Dengue virus genome sequence is just 10735 nucleotides, so there is not a full 2000-nucleotide region from nucleotide 10001 to the end of the sequence at nucleotide 10735.

The above analysis of local variation in GC content is what is known as a sliding window analysis of GC content. By calculating the GC content in each 2000-nucleotide chunk of the Dengue virus genome, you are effectively sliding a 2000-nucleotide window along the DNA sequence from start to end, and calculating the GC content in each non-overlapping window (chunk of DNA).

Note that this sliding window analysis of GC content is a slightly simplified version of the method usually carried out by bioinformaticians. In this simplified version, we have calculated the GC content in non-overlapping windows along a DNA sequence. However, it is more usual to calculate GC content in overlapping windows along a sequence, although that makes the code slightly more complicated.

## 37.5 A sliding window plot of GC content

It is common to use the data generated from a sliding window analysis to create a sliding window plot of GC content. To create a sliding window plot of GC content, you plot the local GC content in each window of the genome, versus the nucleotide position of the start of each window. We can create a sliding window plot of GC content by typing:

```
starts <- seq(1, length(dengueseq_vector)-2000, by = 2000)

n <- length(starts)      # Find the length of the vector "starts"

chunkGCs <- numeric(n)   # Make a vector of the same
                         #length as vector "starts",
                         # but just containing zeroes
for (i in 1:n) {
```

```
         chunk <- dengueseq_vector[starts[i]:(starts[i]+1999)]
         chunkGC <- GC(chunk)
         print(chunkGC)
         chunkGCs[i] <- chunkGC
     }
```

```
## [1] 0.465
## [1] 0.4525
## [1] 0.4705
## [1] 0.479
## [1] 0.4545
```

```
plot(starts,
     chunkGCs,
     type="b",
     xlab="Nucleotide start position",
     ylab="GC content")
```
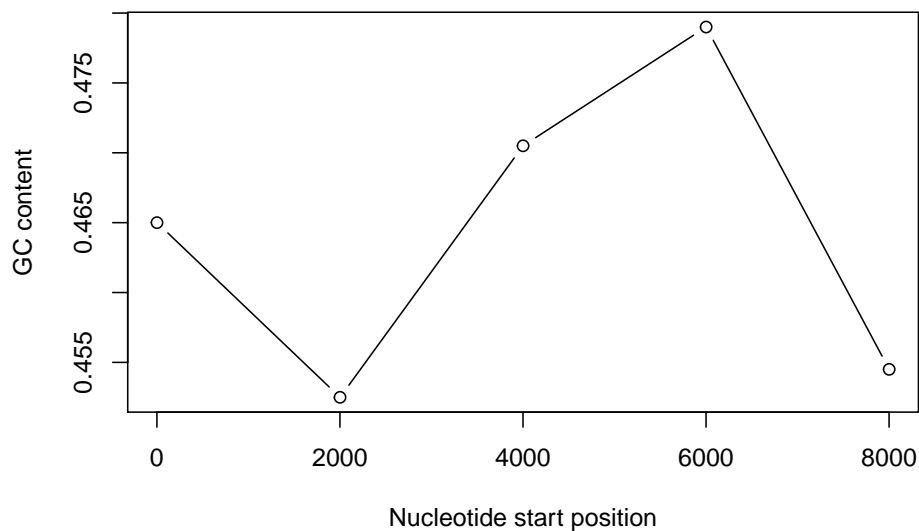


In the code above, the line `chunkGCs <- numeric(n)` makes a new vector `chunkGCs` which has the same number of elements as the vector starts (5 elements here). This vector chunkGCs is then used within the for loop for storing the GC content of each chunk of DNA.

After the loop, the vector starts can be plotted against the vector chunkGCs using the `plot()` function, to get a plot of GC content against nucleotide position in the genome sequence. This is a sliding window plot of GC content.

You may want to use the code above to create sliding window plots of GC content of different species' genomes, using different window sizes. Therefore, it makes sense to use a function to do the sliding window plot, that can take the

windo wsize that the user wants to use and the sequence that the user wants to study as arguments (inputs).

The compbio4all has a function `plot_sliding_window()` that does this.

This function will make a sliding window plot of GC content for a particular input sequence inputseq specified by the user, using a particular window size window size specified by the user. Once you have typed in this function once, you can use it again and again to make sliding window plots of GC contents for different input DNA sequences, with different window sizes. For example, you could create two different sliding window plots of the DEN-1 Dengue virus genome sequence, using window sizes of 3000 and 300 nucleotides, respectively:

```
plot_sliding_window(3000, dengueseq_vector)
```

```
## [1] 0.4646667
## [1] 0.4606667
## [1] 0.4653333
```
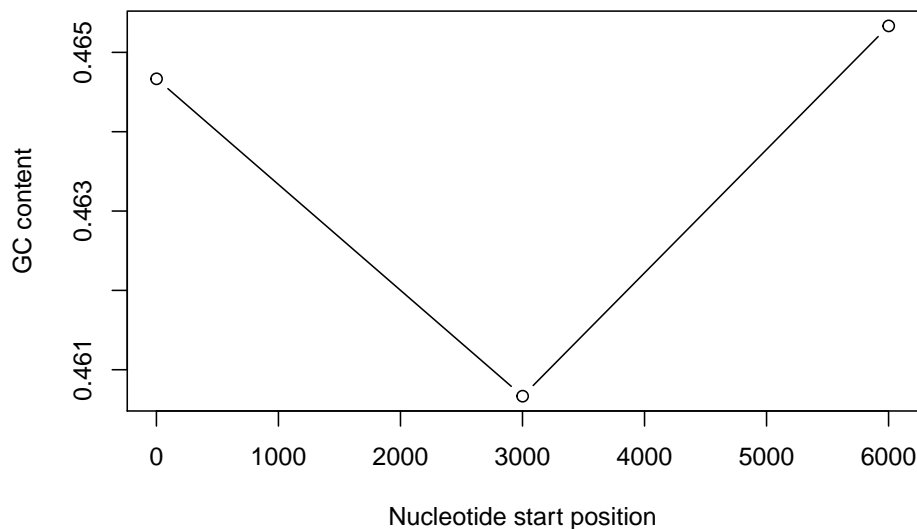


```
plot_sliding_window(300, dengueseq_vector)
```

```
## [1] 0.4366667
## [1] 0.4766667
## [1] 0.5
## [1] 0.5066667
## [1] 0.4466667
## [1] 0.4333333
## [1] 0.4466667
## [1] 0.4866667
## [1] 0.4633333
## [1] 0.45
```

```
## [1] 0.44
## [1] 0.4633333
## [1] 0.4433333
## [1] 0.4
## [1] 0.4766667
## [1] 0.49
## [1] 0.4733333
## [1] 0.4666667
## [1] 0.4666667
## [1] 0.4866667
## [1] 0.4966667
## [1] 0.48
## [1] 0.49
## [1] 0.49
## [1] 0.4633333
## [1] 0.46
## [1] 0.4633333
## [1] 0.42
## [1] 0.4466667
## [1] 0.4433333
## [1] 0.4466667
## [1] 0.4733333
## [1] 0.4733333
## [1] 0.4966667
## [1] 0.4833333
```
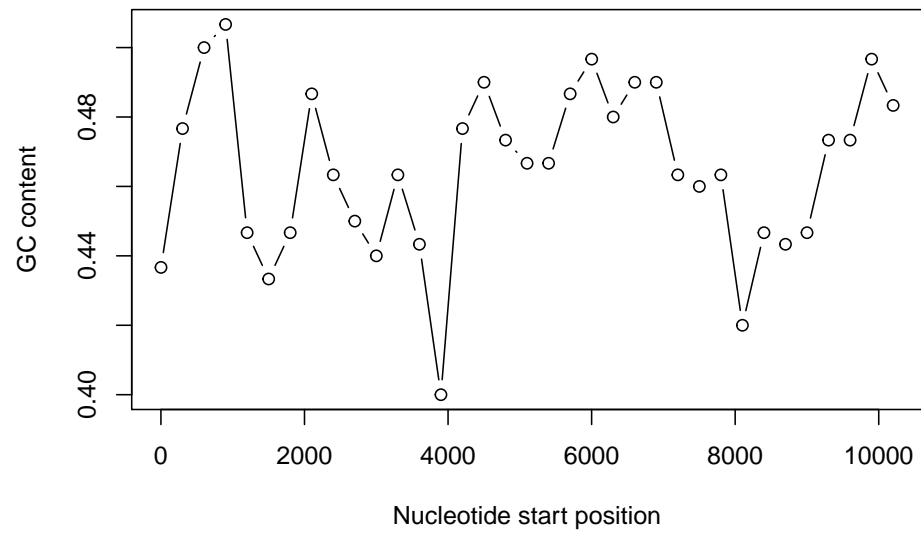
## 37.5.1 Over-represented and under-represented DNA words

In the previous chapter, you learned that the `count()` function in the `seqinr` R package can calculate the frequency of all DNA words of a certain length in a DNA sequence. For example, if you want to know the frequency of all DNA words that are 2 nucleotides long in the Dengue virus genome sequence, you can type:

```
dengueseq_vector_lc <- tolower(dengueseq_vector)
seqinr::count(dengueseq_vector_lc, 2)
```

```
##
##   aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
## 1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
```

It is interesting to identify DNA words that are two nucleotides long (**dinucleotides**, i.e.. "AT", "AC", etc.) that are over-represented or under-represented in a DNA sequence. If a particular DNA word is over-represented in a sequence, it means that it occurs many more times in the sequence than you would have expected by chance. Similarly, if a particular DNA word is under-represented in a sequence, it means it occurs far fewer times in the sequence than you would have expected.

A statistic called *p* (*Rho*) is used to measure how over- or under-represented a particular DNA word is. For a 2-nucleotide (dinucleotide) DNA word p is calculated as:

p(xy) = fxy/(fx*fy),

where fxy and fx are the frequencies of the DNA words xy and x in the DNA sequence under study. For example, the value of for the DNA word "TA" can be calculated as: p(TA) = fTA/(fT* fA), where fTA, fT and fA are the frequencies of the DNA words "TA", "T" and "A" in the DNA sequence.

The idea behind the p statistic is that, if a DNA sequence had a frequency fx of a 1-nucleotide DNA word x, and a frequency fy of a 1-nucleotide DNA word y, then we expect the frequency of the 2-nucleotide DNA word xy to be fx* fy. That is, the frequencies of the 2-nucleotide DNA words in a sequence are expected to be equal the products of the specific frequencies of the two nucleotides that compose them. If this were true, then p would be equal to 1. If we find that p is much greater than 1 for a particular 2-nucleotide word in a sequence, it indicates that that 2-nucleotide word is much more common in that sequence than expected (i.e.. it is over-represented).

For example, say that your input sequence has only 5% Ts (i.e.. fT = 0.05). In a random DNA sequence with 5% Ts, you would expect to see the word "TT" very infrequently. In fact, we would only expect 0.05 * 0.05=0.0025 (0.25%) of 2-nucleotide words to be TTs (i.e.. we expect fTT = fT* fT). This is because Ts are rare, so they are expected to be adjacent to each other very infrequently

if the few Ts are randomly scattered throughout the DNA. Therefore, if you see lots of TT 2-nucleotide words in your real input sequence (eg. fTT = 0.3, so p = 0.3/0.0025 = 120), you would suspect that natural selection has acted to increase the number of occurrences of the TT word in the sequence (presumably because it has some beneficial biological function).

To find over-represented and under-represented DNA words that are 2 nucleotides long in the DEN-1 Dengue virus sequence, we can calculate the p statistic for each 2-nucleotide word in the sequence. For example, given the number of occurrences of the individual nucleotides A, C, G and T in the Dengue sequence, and the number of occurrences of the DNA word GC in the sequence (500, from above), we can calculate the value of p for the 2-nucleotide DNA word "GC", using the formula p(GC) = fGC/(fG * fC), where fGC, fG and fC are the frequencies of the DNA words "GC", "G" and "C" in the DNA sequence:

```
count(dengueseq_vector_lc, wordsize = 1) # Get the number of occurrences of 1-nucleotic
```

```
##
##    a    c    g    t
## 3426 2240 2770 2299
```

```
2770/(3426+2240+2770+2299) # Get fG
```

```
## [1] 0.2580345
```

```
2240/(3426+2240+2770+2299) # Get fC
```

```
## [1] 0.2086633
```

```
count(dengueseq_vector_lc, 2) # Get the number of occurrences of 2-nucleotide DNA wor
```

```
##
##   aa   ac   ag   at   ca   cc   cg   ct   ga   gc   gg   gt   ta   tc   tg   tt
## 1108  720  890  708  901  523  261  555  976  500  787  507  440  497  832  529
```

```
500/(1108+720+890+708+901+523+261+555+976+500+787+507+440+497+832+529) # Get fGC
```

```
## [1] 0.04658096
```

```
0.04658096/(0.2580345*0.2086633) # Get rho(GC)
```

```
## [1] 0.8651364
```

We calculate a value of p(GC) of approximately 0.865. This means that the DNA word "GC" is about 0.865 times as common in the DEN-1 Dengue virus sequence than expected. That is, it seems to be slightly under-represented.

Note that if the ratio of the observed to expected frequency of a particular DNA word is very low or very high, then we would suspect that there is a statistical under-representation or over-representation of that DNA word. However, to be sure that this over- or under-representation is statistically significant, we would

need to do a statistical test. We will not deal with the topic of how to carry out the statistical test here.

## 37.6 Summary

In this chapter, you will have learned to use the following functions:

- `seq()` for creating a sequence of numbers
- `print()` for printing out the value of a variable
- `plot()` for making a plot (eg. a scatterplot)
- `numeric()` for making a numeric vector of a particular length

All of these functions belong to the standard installation of R. You also learned how to use for loops to carry out the same operation again and again, each time on different inputs.

## 37.7 Further Reading

For background reading on DNA sequence statistics, it is recommended to read Chapter 1 of Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

For more in-depth information and more examples on using the seqinr package for sequence analysis, look at the seqinr documentation, http://pbil.univ-lyon 1.fr/software/seqinr/doc.php?lang=eng.

There is also a very nice chapter on "Analyzing Sequences", which includes examples of using seqinr for sequence analysis, in the book Applied statistics for bioinformatics using R by Krijnen (available online at cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf).

## 37.8 Acknowledgements

In "A little book…" Coghlan write "Many of the ideas for the examples and exercises for this chapter were inspired by the Matlab case studies on *Haemophilus influenzae* (www.computational-genomics.net/case_studies/haemophilus_demo.html) and Bacteriophage lambda (http://www.computational-genomics.net/cas e_studies/lambdaphage_demo.html) from the website that accompanies the book Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/)."

## 37.9   Exercises

Answer the following questions, using the R package. For each question, please record your answer, and what you typed into R to get this answer.

Model answers to the exercises are given in Answers to the exercises on DNA Sequence Statistics.

1. Draw a sliding window plot of GC content in the DEN-1 Dengue virus genome, using a window size of 200 nucleotides. Do you see any regions of unusual DNA content in the genome (eg. a high peak or low trough)? Make a sketch of each plot that you draw. At what position (in base-pairs) in the genome is there the largest change in local GC content (approximate position is fine here)? Compare the sliding window plots of GC content created using window sizes of 200 and 2000 nucleotides. How does window size affect your ability to detect differences within the Dengue virus genome?

2. Draw a sliding window plot of GC content in the genome sequence for the bacterium Mycobacterium leprae strain TN (accession NC_002677) using a window size of 20000 nucleotides. Do you see any regions of unusual DNA content in the genome (eg. a high peak or low trough)?

3. Make a sketch of each plot that you drew for the previous quetion. Write down the approximate nucleotide position of the highest peak or lowest trough that you see. Why do you think a window size of 20000 nucleotides was chosen? What do you see if you use a much smaller window size (eg. 200 nucleotides) or a much larger window size (eg. 200,000 nucleotides)?

4. Advanced: Write a function to calculate the AT content of a DNA sequence (i.e.. the fraction of the nucleotides in the sequence that are As or Ts). What is the AT content of the Mycobacterium leprae TN genome? Hint: use the function count() to make a table containing the number of As, Gs, Ts and Cs in the sequence. Remember that function count() produces a table object, and you can access the elements of a table object using double square brackets. Do you notice a relationship between the AT content of the *Mycobacterium leprae* TN genome, and its GC content?

5. Advanced: Write a function to draw a sliding window plot of AT content. Use it to make a sliding window plot of AT content along the Mycobacterium leprae TN genome, using a window size of 20000 nucleotides. Do you notice any relationship between the sliding window plot of GC content along the Mycobacterium leprae genome, and the sliding window plot of AT content? Make a sketch of the plot that you draw.

6. Is the 3-nucleotide word GAC GC over-represented or under-represented in the Mycobacterium leprae TN genome sequence? What is the frequency of this word in the sequence? What is the expected frequency of this word in the sequence? What is the p (Rho) value for this word? How would you figure out whether there is already an R function to calculate p (Rho)? Is there one that you could use?

# Chapter 38

# Working with vectors and matrices

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0) with assistant from Havannah Tung.

In previous practicals, you learned how to create different types of variables in R such as scalars, vectors and lists. Sometimes it is useful to create a variable before you actually need to store any data in the variable. To create a vector without actually storing any data in it, you can use the numeric() command to create a vector for storing numbers, or the character() command to create a vector for storing characters (eg. "A", "hello", etc.) For example, you may want to create a vector variable for storing the square of a number, and then store numbers in its elements afterwards:

```r
# Create a vector "myvector" for storing numbers
myvector <- numeric()

# Fill in the values in the vector "myvector"
for (i in 1:10){
  myvector[i] <- i*i
  }

# Print out the vector "myvector"
myvector
```

```
## [1]   1   4   9  16  25  36  49  64  81 100
```

Note that if you try to store numbers in the elements of a vector that you have not yet created, you will get an error message, for example:

```r
for (i in 1:10) {
  avector[i] <- i*i
  }
```

Another very useful type of variable is a **matrix**. You can create a matrix in *R* using the *matrix()* command. If you look at the help page for the *matrix()* command, you will see that its arguments (inputs) are the data to store in the matrix, the number of rows to store it in, the number of columns to store it in, and whether to fill the matrix with data column-by-column or row-by-row. For example, say you have the heights and weights of eight patients in a hospital in two different vectors:

```r
heights <- c(180, 170, 175, 160, 183, 177, 179, 182)
weights <- c(90, 88, 100, 68, 95, 120, 88, 93)
```

To store this data in a matrix that has one column per person, and one row for heights and one row for weights, we type:

```r
mymatrix <- matrix(c(heights,weights), 2, 8, byrow=TRUE)
mymatrix # Print out the matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]  180  170  175  160  183  177  179  182
## [2,]   90   88  100   68   95  120   88   93
```

We needed to use the argument `byrow=TRUE` to tell the *matrix()* command to fill the matrix row-by-row (i.e.. to put the values from the vector heights into the first row of the matrix, and the values from the vector weights into the second row of the matrix).

You can assign names to the rows and columns of a matrix using the rownames() and colnames() commands, respectively. For example, to assign names to the rows and columns of matrix mymatrix, you could type:

```r
rownames(mymatrix) <- c("height", "weight")
colnames(mymatrix) <- c("patient1", "patient2", "patient3",
                        "patient4", "patient5", "patient6",
                        "patient7", "patient8")

mymatrix # Print out the matrix now
```

```
##        patient1 patient2 patient3 patient4 patient5 patient6 patient7 patient8
## height      180      170      175      160      183      177      179      182
## weight       90       88      100       68       95      120       88       93
```

Once you have created a matrix, you can access the values in the elements of the matrix by using square brackets containing the indices of the row and column of the element. For example, if you want to access the value in the second row and fourth column of matrix mymatrix, you can type:

```
mymatrix[2,4]
```

```
## [1] 68
```

If you want to access all the values in a particular row of the matrix, you can just type the index for the row, and leave out the index for the column. For example, if you want to get the values in the second row of the matrix mymatrix, type:

```
mymatrix[2,]
```

```
## patient1 patient2 patient3 patient4 patient5 patient6 patient7 patient8
##       90       88      100       68       95      120       88       93
```

Likewise, if you want to get the values in a particular column of a matrix, leave out the index for the row, and just type the column index. For example, if you want to get the values in the fourth row of the mymatrix, type:

```
mymatrix[,4]
```

```
## height weight
##    160     68
```

# Chapter 39

# A note on changes to database entries

Sometimes databases entries not linked or database entries get removed. When revising this chapter I had to update three of the accession numbers because the UniProt accessions in the original version weren't working for me.

For example the original UniProt entry included in the chapter E3M2K8 does not currently link directly to the NCBI entry for this same protein. Instead, I had to use the NCBI accession XP_003109757.1. I'm not sure why this is.

Two accession numbers in the original version of this chapter appear to have been removed from UniProt because they were preliminary and perhaps did not meet the quality standards for UniProt. While the NCBI Gene and Protein are meant to be a record of *all* sequence, NCBI RefSeq and UniProt are carefully curated to contain information that is consistent, accurate, and represents biological reality.

Searching the UniProt data base for two accessions in the original version of this chapter, E1FUV2 (*Loa loa*) and A8NSK3(*Brugia malayi*), both yielded the result: >"This entry is obsolete... this entry was deleted. The protein sequence for this entry is available in UniParc. For previous versions of this entry, please look at its history."

Looking at the history for the *Loa loa* version of the protein we can see it was listed as "unreviewed" prior to it being pulled and has the note "The sequence shown here is derived from an ... whole genome shotgun (WGS) entry which i preliminary data." (https://www.uniprot.org/uniprot/E1FUV2.txt?version=9). Once they got around to reviewing in, the curators at UniProt must have decided that it didn't meet the standards for inclusion in the database. The information about the sequence in other databases, however, was not deleted, just the entry in UniProt.

Most proteins are not experimentally studied in a lab, let alone their structure determined or their expression levels assessed. They are therefore predicted to be real based on their similarity to proteins that have been studied in model organisms.

To track down these sequences, I used BLAST. BLASTing the well-studied model organism *C. elegans* version of the protein against the *Loa loa* genome resulted in a hypothetical protein LOAG_18175 a 93% query coverage with C. elegans and 44% identity. The accession number for this predicted protein in NCBI is XP_020305433.1. I similarly tracked down the accession for the *Brugia malayi* version of the protein.

# Chapter 40

# Computational gene finding

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0).

**NOTE: this chapter has not been significantly revised**

## 40.1 Preliminaries

### 40.1.1 Packages

```
library(Biostrings)
library(seqinr)
library(compbio4all)
```

## 40.2 The genetic code

A **protein-coding gene** starts with an "ATG" and is followed by codons (DNA triplets) that code for amino acids. Protein-coding geneds end with a "TGA", "TAA", or "TAG". That is, the **start codon** of a gene is always "ATG", while the stop codon of a gene can be "TGA", "TAA" or "TAG". The start codon ATG also codes for the amino acid **methionine (Met)**. In some proteins this initial Met is part of the final protein, but in others is cleaved of as part of **post-transnational modification** of the polypeptide chain.

In *R*, you can view the **standard genetic code**, the correspondence between codons and the amino acids that they are translated into, by using the `tablecode()` function in the **seqinr** package:

```
seqinr::tablecode()
```

**Genetic code 1 : standard**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| TTT | Phe | TCT | Ser | TAT | Tyr | TGT | Cys |
| TTC | Phe | TCC | Ser | TAC | Tyr | TGC | Cys |
| TTA | Leu | TCA | Ser | TAA | Stp | TGA | Stp |
| TTG | Leu | TCG | Ser | TAG | Stp | TGG | Trp |
| CTT | Leu | CCT | Pro | CAT | His | CGT | Arg |
| CTC | Leu | CCC | Pro | CAC | His | CGC | Arg |
| CTA | Leu | CCA | Pro | CAA | Gln | CGA | Arg |
| CTG | Leu | CCG | Pro | CAG | Gln | CGG | Arg |
| ATT | Ile | ACT | Thr | AAT | Asn | AGT | Ser |
| ATC | Ile | ACC | Thr | AAC | Asn | AGC | Ser |
| ATA | Ile | ACA | Thr | AAA | Lys | AGA | Arg |
| ATG | Met | ACG | Thr | AAG | Lys | AGG | Arg |
| GTT | Val | GCT | Ala | GAT | Asp | GGT | Gly |
| GTC | Val | GCC | Ala | GAC | Asp | GGC | Gly |
| GTA | Val | GCA | Ala | GAA | Glu | GGA | Gly |
| GTG | Val | GCG | Ala | GAG | Glu | GGG | Gly |

You can see from this table that "ATG" is translated to Met (the amino acid methionine), and that "TAA", "TGA" and "TAG" correspond to "Stp" (stop codons), which are not translated to any amino acid, but signal the end of translation.

## 40.3   Finding start and stop codons in a DNA sequence

When a stretch of a genome is sequence for the first time researchers do not know if there are any genes present. Fhe first requirement for a gene is a start codon and a stop codon. To look for all the potential start and stop codons in a DNA sequence, we need to find all the "ATG"s, "TGA"s, "TAA"s, and "TAG"s in the sequence.

To do this, we can use the `matchPattern()` function from the `Biostrings` package, which identifies all occurrences of a particular **motif** (eg. "ATG") in a sequence. As input, `matchPattern()` requires that the sequences be in the form of a **string** of single characters with no spaces.

For example, we can look for all "ATG"s in the sequence "AAAATGCAGTAAC-CCATGCCC" like this. First, define the sequence we want to scan.

```r
s1 <- "aaaatgcagtaacccatgccc"
```

Then use `matchPattern()` from `Biostrings` to located all of the `atg` codons.

```r
matchPattern("atg", s1) # Find all ATGs in the sequence s1
```

```
## Views on a 21-letter BString subject
```

```
## subject: aaaatgcagtaacccatgccc
## views:
##       start end width
## [1]     4   6     3 [atg]
## [2]    16  18     3 [atg]
```

The output from `matchPattern()` tells us that there are two "ATG"s in the sequence, at nucleotides 4-6, and at nucleotides 16-18. Beause this its a short sequence, we can see these by visually inspecting the sequence "AAAAT-GCAGTAACCCATGCCC". For long sequences of 100s to 1000s to tens of thousands of bases this is not a tenable approach.

Similarly, if you use `matchPattern()` to find the positions of "TAA"s, "TGA"s, and "TAG"s in the sequence "AAAATGCAGTAACCCATGCCC", you will find that it has one "TAA" at nucleotides 10-12, but no "TAG"s or "TGA"s.

The function `find_start_stop_codons()` written by Coghlan (2011) and included in `combio4all` can be used to find all potential start and stop codons in a DNA sequence:

```
compbio4all::find_start_stop_codons(s1)
```

```
## $positions
## [1]  4 10 16
##
## $types
## [1] "atg" "taa" "atg"
```

The result of the function is returned as a list variable that contains two elements: the first element of the list is a vector containing the positions of potential start and stop codons in the input sequence, and the second element of the list is a vector containing the type of those start/stop codons ("atg", "taa", "tag", or "tga").

The output for sequence s1 tells us that sequence s1 has an "ATG" starting at nucleotide 4, a "TAA" starting at nucleotide 10, and another "ATG" starting at nucleotide 16.

We can use the function `find_start_stop_codons()` to find all potential start and stop codons in longer sequences. For example, say we want to find all potential start and stop codons in the first 500 nucleotides of the genome sequence of the DEN-1 Dengue virus (NCBI accession NC_001477).

In a previous chapter, you learned that you can retrieve a sequence for an NCBI accession using the `entrez_fetch()` function. Thus, to retrieve the genome sequence of the DEN-1 Dengue virus (NCBI accession NC_001477), we can type:

```
dengueseq_fasta <- rentrez::entrez_fetch(db = "nucleotide",
                          id = "NC_001477",
```

```
                                rettype = "fasta")
```

The data are also included in the `combio4all` package

```
data(dengueseq_fasta)
```

```
is(dengueseq_fasta)
```

```
## [1] "character"          "vector"             "data.frameRowLabels"
## [4] "SuperClassMethod"   "characterORconnection" "character_OR_NULL"
## [7] "atomic"             "vector_OR_Vector"   "vector_OR_factor"
```

```
str(dengueseq_fasta)
```

```
##  chr ">NC_001477.1 Dengue virus 1, complete genome\nAGTTGTTAGTCTACGTGGACCGACAAGAACAG
```

The raw FASTA file needs to be cleaned up a bit first.

```
dengueseq_fasta <- gsub("\n", "", dengueseq_fasta)
header. <- ">NC_001477.1 Dengue virus 1, complete genome"
dengueseq_fasta <- gsub(header.,"", dengueseq_fasta)
```

```
str(dengueseq_fasta)
```

```
##  chr "AGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCGGAAGCTTGCTTAACGTAGTTCTAACAGTTTTTTAT
```

And the format finallized

```
dengueseq_fasta_vector <- stringr::str_split(dengueseq_fasta,"")
```

```
is(dengueseq_fasta_vector)
```

```
## [1] "list"            "vector"            "list_OR_List"     "vector_OR_Vector"
## [5] "vector_OR_factor"
```

```
length(dengueseq_fasta_vector)
```

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

```
dengueseq_fasta_vector <- unlist(dengueseq_fasta_vector)
```

```
is(dengueseq_fasta_vector)
```

```
## [1] "character"          "vector"             "data.frameRowLabels"
## [4] "SuperClassMethod"   "characterORconnection" "character_OR_NULL"
## [7] "atomic"             "vector_OR_Vector"   "vector_OR_factor"
```

```
length(dengueseq_fasta_vector)
```

```
## [1] 10735
```

```
dengueseq_fasta_vector[1]
```

```
## [1] "A"
```

The object `dengueseq_fasta_vector` is a vector, and each letter in the DEN-1 Dengue virus DNA sequence is stored in one element of this vector.

```
str(dengueseq_fasta_vector)
```

```
##  chr [1:10735] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T" "C" "T" "A" "C" "G" ...
```

```
dengueseq_fasta_vector[1:10]
```

```
##  [1] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T"
```

To cut out the first 500 nucleotides of the DEN-1 Dengue virus sequence, we can just take the first 500 elements of this vector using bracket notation:

```
dengueseqstart <- dengueseq_fasta_vector[1:500]
```

Find the length of the `dengueseqstart` start vector

```
length(dengueseqstart)
```

```
## [1] 500
```

Next we want to find potential start and stop codons in the first 500 nucleotides of the Dengue virus sequence. We can do this using the `find_start_stop_codons()` function described above. However, the find_start_stop_codons() function requires that the input sequence be in the format of a string of characters, rather than a vector. Therefore, we first need to convert the vector dengueseqstart into a string of characters. We can do that using the `c2s()` function in the seqinr package:

```
dengueseqstart                          # Print out the vector dengueseqstart
```

```
##     [1] "A" "G" "T" "T" "G" "T" "T" "A" "G" "T" "C" "T" "A" "C" "G" "T" "G" "G"
##    [19] "A" "C" "C" "G" "A" "C" "A" "A" "G" "A" "A" "C" "A" "G" "T" "T" "T" "C"
##    [37] "G" "A" "A" "T" "C" "G" "G" "A" "A" "G" "C" "T" "T" "G" "C" "T" "T" "A"
##    [55] "A" "C" "G" "T" "A" "G" "T" "T" "C" "T" "A" "A" "C" "A" "G" "T" "T" "T"
##    [73] "T" "T" "T" "A" "T" "T" "A" "G" "A" "G" "A" "G" "C" "A" "G" "A" "T" "C"
##    [91] "T" "C" "T" "G" "A" "T" "G" "A" "A" "C" "A" "A" "C" "C" "A" "A" "C" "G"
##   [109] "G" "A" "A" "A" "A" "A" "G" "A" "C" "G" "G" "G" "T" "C" "G" "A" "C" "C"
##   [127] "G" "T" "C" "T" "T" "T" "C" "A" "A" "T" "A" "T" "G" "C" "T" "G" "A" "A"
##   [145] "A" "C" "G" "C" "G" "C" "G" "A" "G" "A" "A" "A" "C" "C" "G" "C" "G" "T"
##   [163] "G" "T" "C" "A" "A" "C" "T" "G" "T" "T" "T" "C" "A" "C" "A" "G" "T" "T"
##   [181] "G" "G" "C" "G" "A" "A" "G" "A" "G" "A" "T" "T" "C" "T" "C" "A" "A" "A"
##   [199] "A" "G" "G" "A" "T" "T" "G" "C" "T" "T" "T" "C" "A" "G" "G" "C" "C" "A"
##   [217] "A" "G" "G" "A" "C" "C" "C" "A" "T" "G" "A" "A" "A" "T" "T" "G" "G" "T"
##   [235] "G" "A" "T" "G" "G" "C" "T" "T" "T" "T" "A" "T" "A" "G" "C" "A" "T" "T"
##   [253] "C" "C" "T" "A" "A" "G" "A" "T" "T" "T" "C" "T" "A" "G" "C" "C" "A" "T"
##   [271] "A" "C" "C" "T" "C" "C" "A" "A" "C" "A" "G" "C" "A" "G" "G" "A" "A" "T"
##   [289] "T" "T" "T" "G" "G" "C" "T" "A" "G" "A" "T" "G" "G" "G" "G" "C" "T" "C"
```

```
## [307] "A" "T" "T" "C" "A" "A" "G" "A" "A" "G" "A" "A" "T" "G" "G" "A" "G" "C"
## [325] "G" "A" "T" "C" "A" "A" "A" "G" "T" "G" "T" "T" "A" "C" "G" "G" "G" "G"
## [343] "T" "T" "T" "C" "A" "A" "G" "A" "A" "A" "G" "A" "A" "A" "T" "C" "T" "C"
## [361] "A" "A" "A" "C" "A" "T" "G" "T" "T" "G" "A" "A" "C" "A" "T" "A" "A" "T"
## [379] "G" "A" "A" "C" "A" "G" "G" "A" "G" "G" "A" "A" "A" "A" "G" "A" "T" "C"
## [397] "T" "G" "T" "G" "A" "C" "C" "A" "T" "G" "C" "T" "C" "C" "T" "C" "A" "T"
## [415] "G" "C" "T" "G" "C" "T" "G" "C" "C" "C" "A" "C" "A" "G" "C" "C" "C" "T"
## [433] "G" "G" "C" "G" "T" "T" "C" "C" "A" "T" "C" "T" "G" "A" "C" "C" "A" "C"
## [451] "C" "C" "G" "A" "G" "G" "G" "G" "G" "A" "G" "A" "G" "C" "C" "G" "C" "A"
## [469] "C" "A" "T" "G" "A" "T" "A" "G" "T" "T" "A" "G" "C" "A" "A" "G" "C" "A"
## [487] "G" "G" "A" "A" "A" "G" "A" "G" "G" "A" "A" "A" "A" "T"
```

Convert the vector "dengueseqstart" to a string of characters

```
dengueseqstartstring <- c2s(dengueseqstart) #
```

Print out the object string of characters `dengueseqstartstring`

```
dengueseqstartstring
```

```
## [1] "AGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCGGAAGCTTGCTTAACGTAGTTCTAACAGTTTTTTATT
```

```
dengueseqstartstring <- tolower(dengueseqstartstring)
```

We can then find potential start and stop codons in the first 500 nucleotides of the DEN-1 Dengue virus sequence by typing:

```
find_start_stop_codons(dengueseqstartstring)
```

```
## $positions
##  [1]    7  53  58  64  78  93  95  96 137 141 224 225 234 236 246 255 264 295 298
## [20] 318 365 369 375 377 378 399 404 413 444 470 471 474 478
##
## $types
##  [1] "tag" "taa" "tag" "taa" "tag" "tga" "atg" "tga" "atg" "tga" "atg" "tga"
## [13] "tga" "atg" "tag" "taa" "tag" "tag" "atg" "atg" "atg" "tga" "taa" "atg"
## [25] "tga" "tga" "atg" "atg" "tga" "atg" "tga" "tag" "tag"
```

We see that the lambda sequence has many different potential start and stop codons, for example, a potential stop codon (TAG) at nucleotide 7, a potential stop codon (TAA) at nucleotide 53, a potential stop codon (TAG) at nucleotide 58, and so on.

## 40.4   Reading frames

Potential start and stop codons in a single-standed DNA sequence can be in three different possible **reading frames**. A potential start/stop codon is said to be in the **+1 reading frame** if there is an integer number of triplets $x$ between the first nucleotide of the sequence and the start of the start/stop codon.

Thus, a potential start/stop codon that begins at nucleotides 1 (0 triplets), 4 (1 triplet), 7 (2 triplets)... will be in the +1 reading frame.

If there is an integer number of triplets x, plus one nucleotide (ie. x.3 triplets), between the first nucleotide of the sequence and the start of the start/stop codon, then the start/stop codon is said to be in the **+2 reading frame**. A potential start/stop codon that begins at nucleotides 2 (0.3 triplets), 5 (1.3 triplets), 8 (2.3 triplets) ... is in the +2 reading frame.

Similarly, if there is an integer number of triplets x, plus two nucleotides (ie. x.6 triplets), between the first nucleotides of the sequence and the start of the start/stop codon, the start/stop codon is in the **+3 reading frame**. So a potential start/stop codon that begins at nucleotides 3 (0.6 triplets), 6 (1.6 triplets), 9 (2.6 triplets)... is in the +3 reading frame.

For a potential start and stop codon to be part of the same gene, they must be in the same reading frame.

From the output of `find_start_stop_codons()` for the first 500 nucleotides of the genome of DEN-1 Dengue virus (see above), you can see that there is a potential start codon (ATG) that starts at nucleotide 137, and a potential stop codon (TGA) that starts at nucleotide 141. That is, the potential start codon is from nucleotides 137-139 and the potential stop codon is from nucleotides 141-143. Could the region from nucleotides 137 to 143 possibly be a gene?

We can cut out the region from nucleotides 137 to 143 of the sequence `dengueseqstartstring` to have a look, by using the `substring()` function. If you look at the help page for the `substring()` function, you will see that its arguments (inputs) are the name of the variable containing the string of characters (ie., the DNA sequence), and the coordinates of the substring that you want:

```
substring(dengueseqstartstring,137,143)
```

```
## [1] "atgctga"
```

If we look at the sequence from nucleotides 137-143, "ATGCTGA", we see that it starts with a potential start codon (ATG) and ends with a potential stop codon (TGA).

However, the **ribosome** reads the sequence by scanning the codons (triplets) one-by-one from left to right, and when we break up the sequence into codons (triplets) we see that it does not contain an integer (whole) number of triplets: "ATG CTG A".

This means that even if the ribosome will not recognise the region from 137-143 as a potential gene, as the ATG at nucleotide 137 is not separated from the TGA at nucleotide 141 by an integer number of codons. That is, this ATG and TGA are not in the same reading frame, and so cannot be the start and stop codon of the same gene.

The potential start codon at nucleotide 137 of the `lambdaseqstartstring` sequence is in the +2 reading frame, as there is an integer number of triplets, plus one nucleotide, between the start of the sequence and the start of the start codon (ie. triplets 1-3, 4-6, 7-9, 10-12, 13-15, 16-18, 19-21, 22-24, 25-27, 28-30, …, 133-135, and a single nucleotide 136).

However, the potential stop codon at nucleotide 141 is the +3 reading frame, as there are two nucleotides plus an integer number of triplets between the start of the sequence and the start of the stop codon (ie. triplets 1-3, 4-6, 7-9, 10-12, 13-15, 16-18, 19-21, 22-24, 25-27, 28-30, 31-33, 34-36, 37-39, 40-42, 43-45, …, 133-135, 136-138, and two nucleotides 139, 140).

As the potential start codon at nucleotide 137 and the potential stop codon at nucleotide 141 are in different reading frames, they are not separated by an integer number of codons, and therefore cannot be part of the same gene.

## 40.5  Finding open reading frames on the forward strand of a DNA sequence

To find potential genes, we need to look for a potential start codon, followed by an integer number of codons, followed by a potential stop codon. This is eqivalent to looking for a potential start codon followed by a potential stop codon that is in the same reading frame. Such a stretch of DNA is known as an **open reading frame (ORF), and is a good candidate for a potential gene.

The function `plot_ORFs_in_seq()` (Coghlan 2011) plots the potential start and stop codons in the three different reading frames of a DNA sequence. For example, to plot the potential start and stop codons in the first 500 nucleotides of the DEN-1 Dengue virus genome, we type:

```
par(mfrow = c(1,1))
plot_start_stop_codons(sq = dengueseqstartstring)
```

```
## This may take a second...
```

**Predicted start (red) and stop (blue) codons**
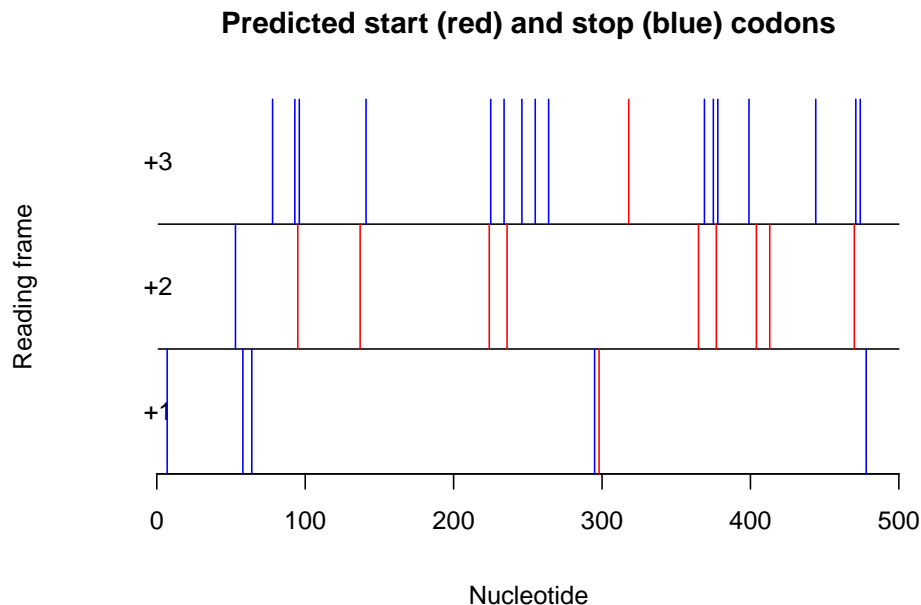


In the picture produced by `plot_ORFs_in_seq()`, the x-axis represents the input sequence (`dengueseqstartstring` here). The potential start codons are represented by vertical red lines, and potential stop codons are represented by vertical blue lines.

Three different layers in the picture show potential start/stop codons in the +1 reading frame (bottom layer), +2 reading frame (middle layer), and +3 reading frame (top layer).

We can see that the start codon at nucleotide 137 is represented by a vertical red line in the layer corresponding to the +2 reading frame (middle layer). There are no potential stop codons in the +2 reading frame to the right of that start codon. Thus, the start codon at nucleotide 137 does not seem to be part of an open reading frame.

We can see however that in the +3 reading frame (top layer) there is a predicted start codon (red line) at position 318 and that this is followed by a predicted stop codon (blue line) at position 371. Thus, the region from nucleotides 318 to 371 could be a potential gene in the +3 reading frame. In other words, the region from nucleotides 318 to 371 is an open reading frame, or ORF.

For example, we can use `find_start_stop_codons()` to find all ORFs in the sequence s1:

```
s1 <- "aaaatgcagtaacccatgccc"
```

```
find_start_stop_codons(s1)
```

```
## $positions
```

```
## [1]   4 10 16
##
## $types
## [1] "atg" "taa" "atg"
```

The function `find_ORFs_in_seq()` returns a list variable, where the first element of the list is a vector of the start positions of ORFs, the second element of the list is a vector of the end positions of those ORFs, and the third element is a vector containing the lengths of the ORFs.

The output for the `find_ORFs_in_seq()` function for s1 tells us that there is one ORF in the sequence s1, and that the predicted start codon starts at nucleotide 4 in the sequence, and that the predicted stop codon ends at nucleotide 12 in the sequence.

We can use the function `find_ORFs_in_seq()` to find the ORFs in the first 500 nucleotides of the DEN-1 Dengue virus genome sequence by typing:

```
find_ORFs_in_seq(dengueseqstartstring)
```

```
## $orfstarts
## [1] 298 318
##
## $orfstops
## [1] 480 371
##
## $orflengths
## [1] 183  54
```

The result from `find_ORFs_in_seq()` indicates that there are two ORFs in the first 500 nucleotides of the DEN-1 Dengue virus genome, at nucleotides 298-480 (start codon at 298-300, stop codon at 478-480), and 318-371 (start codon at 318-320, stop codon at 369-371).

The `compbio4all` function `plot_ORFs_in_seq()` plots the positions of ORFs in a sequence. You can then use this function to plot the positions of the ORFs in dengueseqstartstring by typing:

```
plot_ORFs_in_seq(dengueseqstartstring)
```

**Predicted ORFs**
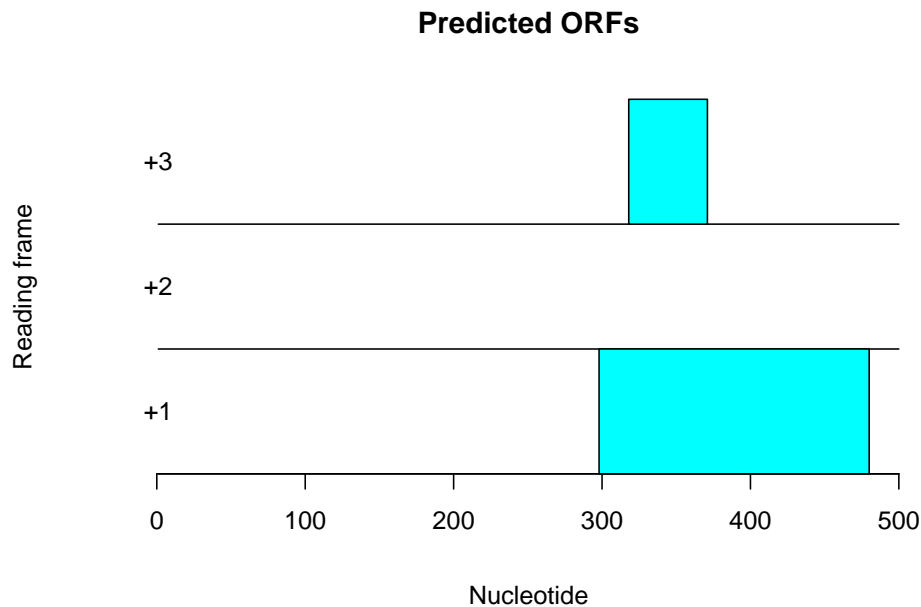


The picture produced by `plot_ORFs_in_seq()` represents the two ORFs in the first 500 nucleotides of the lambda genome as blue rectangles.

One of the ORFs is in the +3 reading frame, and one is in the +1 reading frame. There are no ORFs in the +2 reading frame, as there are no potential stop codons to the right (3') of the potential start codons in the +2 reading frame, as we can see from the picture produced by `find_start_stop_codons()` above.

## 40.6 Predicting the protein sequence for an ORF

If you find an ORF in a DNA sequence, it is interesting to find the DNA sequence of the ORF. For example, the function `find_start_stop_codons()` indicates that there is an ORF from nucleotides 4-12 of the sequence s1 (aaaatgcagtaac-ccatgccc). To look at the DNA sequence for just the ORF, we can use the `substring()` function to cut out that piece of DNA. For example, to cut out the substring of sequence s1 that corresponds to the ORF from nucleotides 4-12, we type:

```
s1 <- "aaaatgcagtaacccatgccc"
```

```
myorf <- substring(s1, 4, 12)
```

Print out the sequence of "myorf"

```
myorf
```

```
## [1] "atgcagtaa"
```

As you can see, the ORF starts with a predicted start codon (ATG), is followed by an integer number of codons (just one codon, CAG, in this case), and ends with a predicted stop codon (TAA).

If you have the DNA sequence of an ORF, you can predict the protein sequence for the ORF by using the `translate()` function from the `seqinr` package. Note that as there is a function called `translate()` in both the `Biostrings` and `seqinr` packages, we need to type `seqinr::translate()` to specify that we want to use the `seqinr` version of the 'translate() function.

The `translate()` function requires that the input sequence be in the form of a *vector* of characters. If your sequence is in the form of a string of characters, you can convert it to a vector of characters using the `s2c()` function from `seqinr` For example, to predict the protein sequence of the ORF myorf, you would type:

```
myorfvector <- s2c(myorf) # Convert the sequence of characters to a vector
```

Print out the value of "myorfvector"

```
myorfvector
```

```
## [1] "a" "t" "g" "c" "a" "g" "t" "a" "a"
```

Now tanslate:

```
seqinr::translate(myorfvector)
```

```
## [1] "M" "Q" "*"
```

From the output of the `seqinr::translate()` function, we see that the predicted start codon (ATG) is translated as a Methionine (M), and that this is followed by a Glutamine (Q). The predicted stop codon is represented as "*" as it is not translated into any amino acid.

## 40.7   Finding open reading frames on the reverse strand of a DNA sequence

Genes in a genome sequence can occur either on the **forward (plus) strand** of the DNA, or on the **reverse (minus) strand**. To find ORFs on the reverse strand of a sequence, we must first infer the reverse strand sequence, and then use our `find_start_stop_codons()` function to find ORFs on the reverse strand.

The reverse strand sequence easily can be inferred from the forward strand sequence, as it is always the **reverse complement** sequence of the forward strand sequence. We can use the `comp()` function from the seqinr package to determine the complement of a sequence, and `rev()` to reverse that sequence in order to give us the reverse complement sequence.

The `comp()` and `rev()` functions require that the input sequence is in the form of a vector of characters. The `s2c()` function can be used to convert a sequence in the form of a string of characters to a vector, while the `c2s()` function is useful for convering a vector back to a string of characters.

For example, if our forward strand sequence is "AAAATGCTTAAAC-CATTGCCC", and we want to find the reverse strand sequence, we type:

```
forward <- "AAAATGCTTAAACCATTGCCC"
```

Convert the string of characters to a vector

```
forwardvector <- s2c(forward)
```

Print out the vector containing the forward strand sequence

```
forwardvector
```

```
##  [1] "A" "A" "A" "A" "T" "G" "C" "T" "T" "A" "A" "A" "C" "C" "A" "T" "T" "G" "C"
## [20] "C" "C"
```

```
comp(forwardvector)
```

```
##  [1] "t" "t" "t" "t" "a" "c" "g" "a" "a" "t" "t" "t" "g" "g" "t" "a" "a" "c" "g"
## [20] "g" "g"
```

Find the reverse strand sequence, by finding the reverse complement

```
reversevector <- rev(comp(forwardvector))
```

Print out the vector containing the reverse strand sequence

```
reversevector
```

```
##  [1] "g" "g" "g" "c" "a" "a" "t" "g" "g" "t" "t" "t" "a" "a" "g" "c" "a" "t" "t"
## [20] "t" "t"
```

Convert the vector to a string of characters

```
reverse <- c2s(reversevector)
```

Print out the string of characters containing the reverse strand sequence

```
reverse
```

```
## [1] "gggcaatggtttaagcatttt"
```

In the command `reversevector <- rev(comp(forwardvector))` above, we are first using the `comp()` function to find the **complement** of the forward strand sequence. We are then using the `rev()` function to take the output sequence given by `comp()` and reverse the order of the letters in that sequence. An equivalent way of doing the same thing would be to type:

Find the complement of the forward strand sequence

```
complement <- comp(forwardvector)
```

Reverse the order of the letters in sequence "complement", to find the reverse strand sequence (the reverse complement sequence)

```
reversevector <- rev(complement)              #
```

Once we have inferred the reverse strand sequence, we can then use the `find_start_stop_codons()` function to find ORFs in the reverse strand sequence:

```
find_start_stop_codons(reverse)
```

```
## $positions
## [1]  6 12
##
## $types
## [1] "atg" "taa"
```

This indicates that there is one ORF of length 9 bp in the reverse strand of sequence "AAAATGCTTAAACCATTGCCC", that has a predicted start codon that starts at nucleotide 6 in the reverse strand sequence and a predicted stop codon that ends at nucleotide 14 in the reverse strand sequence.

## 40.8   Lengths of open reading frames

As you can see from the picture displaying the genetic code made using `tablecode()` (above), three of the 64 different codons are stop codons. This means that in a random DNA sequence the probability that any codon is a potential stop codon is 3/64, or about 1/21 (about 5%).

Therefore, you might expect that sometimes potential start and stop codons can occur in a DNA sequence just due to chance alone, not because they are actually part of any real gene that is transcribed and translated into a protein.

As a result, many of the ORFs in a DNA sequence will not correspond to real genes, but just be stretches of DNA between potential start and stop codons that happened by chance to be found in the sequence.

In other words, an open reading frame (ORF) is just a gene prediction, or a potential gene. It may correspond to a real gene (may be a true positive gene prediction), but it may not (may be a false positive gene prediction).

How can we tell whether the potential start and stop codons of an ORF are probably real start and stop codons, that is, whether an ORF probably corresponds to a real gene that is transcribed and translated into a protein?

In fact, we cannot tell using bioinformatics methods alone (we actually need to do some lab experiments to know), but we can make a fairly confident prediction.

We can make our prediction based on the length of the ORF.

By definition, an ORF is a stretch of DNA that starts with a potential start codon, and ends with a potential stop codon in the same reading frame, and so has no internal stop codons in that reading frame. Because about 1/21 of codons (~5%) in a random DNA sequence are expected to be potential stop codons just by chance alone, if we see a very long ORF of hundreds of codons, it would be surprising that there would be no internal stop codons in such a long stretch of DNA if the ORF were not a real gene.

In other words, long ORFs that are hundreds of codons long are unlikely to occur due to chance alone, and therefore we can be fairly confident that such long ORFs probably correspond to real genes.

# Chapter 41

# Comparative Genomics

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0), with assistance from Aman Virmani.

**NOTE: this chapter has not yet been significaly revised**.

## 41.1   Preliminaries

### 41.1.1   Packages

```
library(compbio4all)

# BiocManager::install("biomaRt")
library(biomaRt)
```

```
##
## Attaching package: 'biomaRt'

## The following object is masked from 'package:seqinr':
##
##     getSequence
```

## 41.2   Introduction

**Comparative genomics** is the field of bioinformatics that involves comparing the genomes of two different species, or of two different strains of the same species.

One of the first questions to ask when comparing the genomes of two species is: do the two species have the same number of genes (i.e.. the same gene content)? Since all life on earth shared a common ancestor at some point, any two species, for example, human and a fruit fly, must have descended from a common ancestor species.

Since the time of the common ancestor of two species (e.g. of human and mouse), some of the genes that were present in the common ancestor species may have been lost from either of the two descendant lineages. Furthermore, the two descendant lineages may have gained genes that were not present in the common ancestor species.

## 41.3   Using the biomaRt R Library to Query the Ensembl Database

To carry out comparative genomic analyses of two animal species whose genomes have been fully sequenced (e.g. human and mouse), it is useful to analyze the data in the Ensembl database (www.ensembl.org).

The main Ensembl database which you can browse on the main Ensembl webpage contains genes from fully sequenced vertebrates, as well as *Saccharomyces cerevisiae* (yeast) and a small number of additional model organism animals (e.g. the nematode worm *Caenorhabditis elegans* and the fruit fly *Drosophila melanogaster*).

There are also Ensembl databases for other groups of organisms, for example Ensembl Protists for Protists, Ensembl Metazoa for Metazoans, Ensembl Bacteria for Bacteria, Ensembl Plants for Plants, and Ensembl Fungi for Fungi.

It is possible to carry out analyses of the Ensembl database using R, with the `biomaRt` R package. The `biomaRt` package can connect to the Ensembl database, and perform queries on the data.

The `biomaRt` *R* package is part of the Bioconductor set of *R* packages, and so can be installed as explained here.

Once you have installed the `biomaRt` package, you can get a list of databases that can be queried using this package by typing:

```
listEnsemblGenomes()        # List all Ensembl genomes that can be queried

##                biomart                            version
## 1       protists_mart      Ensembl Protists Genes 51
## 2 protists_variations Ensembl Protists Variations 51
## 3          fungi_mart          Ensembl Fungi Genes 51
## 4    fungi_variations     Ensembl Fungi Variations 51
## 5        metazoa_mart        Ensembl Metazoa Genes 51
## 6  metazoa_variations  Ensembl Metazoa Variations 51
```

```
## 7         plants_mart       Ensembl Plants Genes 51
## 8   plants_variations   Ensembl Plants Variations 51
```

The names of the databases are listed, and the version of the database.

The `biomaRt` *R* package can actually be used to query many different databases including WormBase, UniProt, Ensembl, etc.

Here, we will discuss using the `biomaRt` package to query the Ensembl database, but it is worth remembering that it also be used to perform queries on other databases such as UniProt.

If you want to perform a query on the Ensembl database using `biomaRt`, you first need to specify that this is the database that you want to query. You can do this using the `useEnsemblGenomes()` function from the `biomaRt` package:

```r
# Specify that we want to query the Ensembl database
ensemblprotists <- useEnsemblGenomes("protists_mart")
```

This tells `biomaRt` that you want to query the Ensembl Protists database. The Ensembl Protists database contains data sets of genomic information for different protist species whose genomes have been fully sequenced.

To see which data sets you can query in the database that you have selected (using `useDataset()`), you can type:

```r
listDatasets(ensemblprotists)
```

```
##                     dataset
## 1        alaibachii_eg_gene
## 2          bnatans_eg_gene
## 3       ddiscoideum_eg_gene
## 4       ehistolytica_eg_gene
## 5          ehuxleyi_eg_gene
## 6         glamblia_eg_gene
## 7           gtheta_eg_gene
## 8    harabidopsidis_eg_gene
## 9           lmajor_eg_gene
## 10 paphanidermatum_eg_gene
## 11   parrhenomanes_eg_gene
## 12         pberghei_eg_gene
## 13        pchabaudi_eg_gene
## 14      pfalciparum_eg_gene
## 15       pinfestans_eg_gene
## 16      pirregulare_eg_gene
## 17        piwayamai_eg_gene
## 18       pkernoviae_eg_gene
## 19        pknowlesi_eg_gene
## 20       plateralis_eg_gene
## 21   pmultistriata_eg_gene
```

```
## 22       pparasitica_eg_gene
## 23        pramorum_eg_gene
## 24          psojae_eg_gene
## 25     ptetraurelia_eg_gene
## 26     ptricornutum_eg_gene
## 27         pultimum_eg_gene
## 28          pvexans_eg_gene
## 29           pvivax_eg_gene
## 30          tbrucei_eg_gene
## 31          tgondii_eg_gene
## 32      tpseudonana_eg_gene
## 33     tthermophila_eg_gene
##                                                                        description
## 1                                            Albugo laibachii genes (ENA 1)
## 2                                          Bigelowiella natans genes (Bigna1)
## 3                                    Dictyostelium discoideum genes (dicty_2.7)
## 4                                    Entamoeba histolytica genes (JCVI-ESG2-1.0)
## 5   Emiliania huxleyi genes (Emiliana huxleyi CCMP1516 main genome assembly v1.0)
## 6                                               Giardia lamblia genes (GL2)
## 7                                     Guillardia theta CCMP2712 genes (Guith1)
## 8                             Hyaloperonospora arabidopsidis genes (HyaAraEmoy2_2.0)
## 9                                            Leishmania major genes (ASM272v2)
## 10                               Pythium aphanidermatum genes (pag1_scaffolds_v1)
## 11                                Pythium arrhenomanes genes (par_scaffolds_v1)
## 12                                      Plasmodium berghei genes (PBANKA01)
## 13                                      Plasmodium chabaudi genes (PCHAS01)
## 14                                 Plasmodium falciparum 3D7 genes (ASM276v2)
## 15                                  Phytophthora infestans genes (ASM14294v1)
## 16                                   Pythium irregulare genes (pir_scaffolds_v1)
## 17                                    Pythium iwayamai genes (piw_scaffolds_v1)
## 18                                 Phytophthora kernoviae genes (PhyKer238/432v1)
## 19                                      Plasmodium knowlesi genes (ASM635v1)
## 20                                   Phytophthora lateralis genes (MPF4_v1.0)
## 21                               Pseudo-nitzschia multistriata genes (ASM90066040v1)
## 22                                Phytophthora parasitica genes (Phyt_para_P1569_V1)
## 23                                    Phytophthora ramorum genes (ASM14973v1)
## 24                                     Phytophthora sojae genes (P.sojae V3.0)
## 25                                   Paramecium tetraurelia genes (ASM16542v1)
## 26                                 Phaeodactylum tricornutum genes (ASM15095v2)
## 27                                            Pythium ultimum genes (pug)
## 28                                   Pythium vexans genes (pve_scaffolds_v1)
## 29                                       Plasmodium vivax genes (ASM241v2)
## 30                               Trypanosoma brucei genes (TryBru_Apr2005_chr11)
## 31                                      Toxoplasma gondii ME49 genes (TGA4)
## 32                                 Thalassiosira pseudonana genes (ASM14940v2)
## 33                              Tetrahymena thermophila genes (JCVI-TTA1-2.2)
```

```
##                                                          version
## 1                                                         ENA 1
## 2                                                         Bigna1
## 3                                                      dicty_2.7
## 4                                                   JCVI-ESG2-1.0
## 5   Emiliana huxleyi CCMP1516 main genome assembly v1.0
## 6                                                            GL2
## 7                                                         Guith1
## 8                                                 HyaAraEmoy2_2.0
## 9                                                        ASM272v2
## 10                                              pag1_scaffolds_v1
## 11                                               par_scaffolds_v1
## 12                                                        PBANKA01
## 13                                                        PCHAS01
## 14                                                       ASM276v2
## 15                                                     ASM14294v1
## 16                                              pir_scaffolds_v1
## 17                                              piw_scaffolds_v1
## 18                                                 PhyKer238/432v1
## 19                                                       ASM635v1
## 20                                                       MPF4_v1.0
## 21                                                  ASM90066040v1
## 22                                               Phyt_para_P1569_V1
## 23                                                     ASM14973v1
## 24                                                     P.sojae V3.0
## 25                                                     ASM16542v1
## 26                                                     ASM15095v2
## 27                                                            pug
## 28                                              pve_scaffolds_v1
## 29                                                       ASM241v2
## 30                                             TryBru_Apr2005_chr11
## 31                                                           TGA4
## 32                                                     ASM14940v2
## 33                                                   JCVI-TTA1-2.2
```

You will see a long list of the organisms for which the Ensembl Protists database has genome data, including *Plasmodium falciparium* (which causes malaria), and *Leishmania major*, which causes leishmaniasis, which is classified by the WHO as a neglected tropical disease.

To perform a query on the Ensembl database using the `biomaRt` R package, you first need to specify which Ensembl data set your query relates to. You can do this using the `useDataset()` function from the `biomaRt` package. For example, to specify that you want to perform a query on the Ensembl *Leishmania major* data set, you would type:

```
ensemblleishmania <- useDataset("lmajor_eg_gene",
                                mart=ensemblprotists)
```

Note that the name of the *Leishmania major* Ensembl data set is "lmajor_eg_gene"; this is the data set listed for *Leishmania major* genomic information when we typed `listDatasets(ensemblprotists)` above.

Once you have specified the particular Ensembl data set that you want to perform a query on, you can perform the query using the `getBM()` function from the `biomaRt` package.

Usually, you will want to perform a query to a particular set of features from the *Leishmania major* Ensembl data set. What types of features can you search for? You can find this out by using the `listAttributes()` function from the `biomaRt` package:

```
leishmaniaattributes <- listAttributes(ensemblleishmania)
```

The listAttributes()' function returns a list object, the first element of which is a vector of all possible features that you can select, and the second element of which is a vector containing explanations of all those features:

```
attributenames <- leishmaniaattributes[[1]]
attributedescriptions <- leishmaniaattributes[[2]]
length(attributenames)  # Find the length of vector "attributenames"
```

```
## [1] 630
```

```
attributenames[1:10]    # Print out the first 10 entries
```

```
##  [1] "ensembl_gene_id"       "ensembl_transcript_id" "ensembl_peptide_id"
##  [4] "ensembl_exon_id"       "description"           "chromosome_name"
##  [7] "start_position"        "end_position"          "strand"
## [10] "band"
                        # in vector "attributenames"
attributedescriptions[1:10]   # Print out the first 10 entries
```

```
##  [1] "Gene stable ID"        "Transcript stable ID"
##  [3] "Protein stable ID"     "Exon stable ID"
##  [5] "Gene description"      "Chromosome/scaffold name"
##  [7] "Gene start (bp)"       "Gene end (bp)"
##  [9] "Strand"                "Karyotype band"
                              # in vector "attributedescriptions"
```

This gives us a very long list of 630 features in the *Leishmania major* Ensembl data set that we can search for by querying the database, such as genes, transcripts (mRNAs), peptides (proteins), chromosomes, GO (Gene Ontology) terms, and so on.

When you are performing a query on the Ensembl *Leishmania major* data set using `getBM()`, you have to specify which of these features you want to retrieve. For example, you can see from the output of `listAttributes()` (see above) that one possible type of feature we can search for are *Leishmania major* genes. To retrieve a list of all *Leishmania major* genes from the *Leishmania major* Ensembl data set, we just need to type:

```
leishmaniagenes <- getBM(attributes = c("ensembl_gene_id"),
                         mart=ensemblleishmania)
```

This returns a list variable leishmaniagenes, the first element of which is a vector containing the names of all *Leishmania major* genes. Thus, to find the number of genes, and print out the names of the first ten genes stored in the vector, we can type:

```
# Get the vector of the names of all L. major genes
leishmaniagenenames <- leishmaniagenes[[1]]

length(leishmaniagenenames)
```

```
## [1] 10030
```

```
leishmaniagenenames[1:10]
```

```
##  [1] "ENSRNAG00049765313" "ENSRNAG00049765315" "ENSRNAG00049765317"
##  [4] "ENSRNAG00049765319" "ENSRNAG00049765321" "ENSRNAG00049765323"
##  [7] "ENSRNAG00049765325" "ENSRNAG00049765327" "ENSRNAG00049765329"
## [10] "ENSRNAG00049765333"
```

This tells us that there are 10300 different *Leishmania major* genes in the L. major Ensembl data set. Note that this includes various types of genes including protein-coding genes (both "known" and "novel" genes, where the "novel" genes are gene predictions that don't have sequence similarity to any sequences in sequence databases), RNA genes, and pseudogenes.

What if we are only interested in protein-coding genes? If you look at the output of `listAttributes(ensemblleishmania)`, you will see that one of the features is "gene_biotype", which is tells us what sort of gene each gene is (e.g. protein-coding, pseudogene, etc.):

```
leishmaniagenes2 <- biomaRt::getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
                         mart=ensemblleishmania)
```

In this case, the `getBM()` function will return a list variable leishmaniagenes2, the first element of which is a vector containing the names of all *Leishmania major* genes, and the second of which is a vector containing the types of those genes:

```
leishmaniagenenames2 <- leishmaniagenes2[[1]] # Get the vector of the names of all L. major genes
leishmaniagenebiotypes2 <- leishmaniagenes2[[2]] # Get the vector of the biotypes of all genes
```

We can make a table of all the different types of genes using the table() function:

```
table(leishmaniagenebiotypes2)
```

```
## leishmaniagenebiotypes2
##          ncRNA protein_coding      pseudogene            rRNA           snRNA
##           1130           8315              94              92             233
##           tRNA
##            166
```

This tells us that there are 8315 protein-coding genes, 94 pseudogenes, and various types of RNA genes (tRNA genes, rRNA genes, snRNA genes, etc.). Thus, there are 8315 human protein-coding genes.

## 41.4  Comparing the number of genes in two species

Ensembl is a very useful resource for comparing the gene content of different species. For example, one simple question that we can ask by analyzing the Ensembl data is: how many protein-coding genes are there in *Leishmania major*, and how many in *Plasmodium falciparum*?

We know how many protein-coding genes are in *Leishmania major* (8315; see above), but what about *Plasmodium falciparum*? To answer this question, we first need to tell the `biomaRt` package that we want to make a query on the Ensembl *Plasmodium falciparum* data set.

We can do this using the `useDataset()` function to select the *Plasmodium falciparum* Ensembl data set.

```
ensemblpfalciparum <- useDataset("pfalciparum_eg_gene",
                                 mart=ensemblprotists)
```

Note that the name of the *Plasmodium falciparum* Ensembl data set is "pfalciparum_eg_gene"; this is the data set listed for *Plasmodium falciparum* genomic information when we typed listDatasets(ensemblprotists) above.

We can then use `getBM()` as above to retrieve the names of all *Plasmodium falciparum* protein-coding genes. This time we have to set the "mart" option in the `getBM()` function to "ensemblpfalciparum", to specify that we want to query the *Plasmodium falciparum* Ensembl data set rather than the *Leishmania major* Ensembl data set:

```
pfalciparumgenes <- getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
                          mart=ensemblpfalciparum)

# Get the names of the P. falciparum genes
pfalciparumgenenames <- pfalciparumgenes[[1]]
```

```
# Get the number of P. falciparum genes
length(pfalciparumgenenames)
```

```
## [1] 5767
```

```
# Get the types of the P. falciparum genes
pfalciparumgenebiotypes <- pfalciparumgenes[[2]]

table(pfalciparumgenebiotypes)
```

```
## pfalciparumgenebiotypes
##            ncRNA nontranslating_CDS      protein_coding         pseudogene
##              102                  4                5358                153
##             rRNA              snRNA                sRNA               tRNA
##               44                 10                  17                 79
```

This tells us that there are 5767 *Plasmodium falciparum* protein-coding genes in Ensembl. That is, *Plasmodium falciparum* seems to have less protein-coding genes than *Leishmania major* (8315 protein-coding genes; see above).

It is interesting to ask: why does *Plasmodium falciparum* have less protein-coding genes than *Leishmania major*? There are several possible explanations: (i) that there have been gene duplications in the *Leishmania major* lineage since Leishmania and Plasmodium shared a common ancestor, which gave rise to new *Leishmania major* genes; (ii) that completely new genes (that are not related to any other *Leishmania major* gene) have arisen in the *Leishmania major* lineage since Leishmania and Plasmodium shared a common ancestor; or (iii) that there have been genes lost from the *Plasmodium falciparum* genome since Leishmania and Plasmodium shared a common ancestor.

To investigate which of these explanations is most likely to be correct, we need to figure out how the *Leishmania major* protein-coding genes are related to *Plasmodium falciparum* protein-coding genes.

## 41.5   Identifying homologous genes between two species

The Ensembl database groups homologous (related) genes together into gene families. If a gene from *Leishmania major* and a gene from *Plasmodium falciparum* are related, they should be placed together into the same Ensembl gene family in the Ensembl Protists database. In fact, if a *Leishmania major* gene has any homologs (related genes) in other protists, it should be placed into some Ensembl gene family in the Ensembl Protists database.

For all *Leishmania major* and *Plasmodium falciparum* genes that are placed together in a gene family, Ensembl classifies the relationship between each pair of *Leishmania major* and *Plasmodium falciparum* genes as orthologs (related genes

that shared a common ancestor in the ancestor of Leishmania and Plasmodium, and arose due to the *Leishmania - Plasmodium* speciation event) or paralogs (related genes that arose due to a duplication event within a species, for example, due to a duplication event in *Leishmania major*, or a duplication event in the Leishmania - *Plasmodium* ancestor).

If you type `listAttributes(ensemblleishmania)` again, you will see that one possible feature that you can search for is "pfalciparum_eg_homolog_ensembl_gene", which is the *Plasmodium falciparum* ortholog of a *Leishmania major* gene.

Another possible feature that you can search for is "pfalciparum_eg_homology_orthology_type", which describes the type of orthology relationship between a particular *Leishmania major* gene and its *Plasmodium falciparum* ortholog. For example, if a particular *Leishmania major* gene has two *Plasmodium falciparum* orthologs, the relationship between the *Leishmania major* gene and each of the *Plasmodium falciparum* orthologs will be "ortholog_one2many" (one-to-many orthology).

This can arise in the case where there was a duplication in the *Plasmodium falciparum* lineage after Plasmodium and Leishmania diverged, which means that two different *Plasmodium falciparum* genes (which are paralogs of each other) are both orthologs of the same *Leishmania major* gene.

Therefore, we can retrieve the Ensembl identifiers of the *Plasmodium falciparum* orthologs of all *Leishmania major* genes by typing:

```
leishmaniagenes <- getBM(attributes = c("ensembl_gene_id",
                                        "pfalciparum_eg_homolog_ensembl_gene",
                              "pfalciparum_eg_homolog_orthology_type"),
                                 mart=ensemblleishmania)
```

This will return an *R* list variable leishmaniagenes, the first element of which is a vector of Ensembl identifiers for all *Leishmania major* coding genes, and the second element of which is a vector of Ensembl identifiers for their *Plasmodium falciparum* orthologs, and the third element of which is a vector with information on the orthology types.

We can print out the names of the first 10 *Leishmania major* genes and their *Plasmodium falciparum* orthologs, and their orthology types, by typing:

```
# Get the names of all *Leishmania major* genes
leishmaniagenenames <- leishmaniagenes[[1]]

# Get the P. falciparum orthologs of all L. major genes
leishmaniaPforthologues <- leishmaniagenes[[2]]

# Get the orthology relationship type
leishmaniaPforthologuetypes <- leishmaniagenes[[3]]
```

```
leishmaniagenenames[1:10]
```

```
##  [1] "LMJF_01_0780" "LMJF_01_0350" "LMJF_01_0280" "LMJF_01_0290" "LMJF_01_0335"
##  [6] "LMJF_01_0600" "LMJF_01_0220" "LMJF_01_0220" "LMJF_01_0220" "LMJF_01_0220"
```

```
leishmaniaPforthologues[1:10]
```

```
##  [1] ""                 ""                 ""                 ""
##  [5] ""                 ""                 "PF3D7_0528700" "PF3D7_1116300"
##  [9] "PF3D7_0804800" "PF3D7_1423200"
```

```
leishmaniaPforthologuetypes[1:10]
```

```
##  [1] ""                   ""                   ""
##  [4] ""                   ""                   ""
##  [7] "ortholog_many2many" "ortholog_many2many" "ortholog_many2many"
## [10] "ortholog_many2many"
```

Not all *Leishmania major* genes have *Plasmodium falciparum* orthologs; this is why when we print out the first 10 elements of the vector leishmaniaPfortho-logues, some of the elements are empty.

To find out how many *Leishmania major* genes have orthologs in *Plasmodium falciparum*, we can first find the indices of the elements of the vector leishma-niaPforthologues that are empty:

```
myindex <- leishmaniaPforthologues == ""
```

We can then find out the names of the Leishmania gene genes corresponding to those indices:

```
leishmaniagenenames2 <- leishmaniagenenames[myindex]
length(leishmaniagenenames2)
```

```
## [1] 8191
```

This tells us that 8191 *Leishmania major* genes do not have *Plasmodium falci-parum* orthologs.

How many of the 8191 *Leishmania major* genes that do not have *Plasmodium falaciparum* orthologs are protein-coding genes? To answer this question, we can merge together the information in the R list variable leishmaniagenes2 (which contains information on the name of each *Leishmania major* gene and its type; see above), and the R list variable leishmaniagenes (which contains information on the name of each L. major gene and its *Plasmodium falciparum* orthologs).

Remember that leishmaniagenes2 was created by typing:

```
leishmaniagenes2 <- getBM(attributes = c("ensembl_gene_id", "gene_biotype"),
                          mart=ensemblleishmania)
```

To combine leishmaniagenes and leishmaniagenes2, we can use the merge() function in R, which can merge together two list variables that contain some named elements in common (in this case, both list variables contain a vector that has the names of *Leishmania major* genes):

```
leishmaniagenes3 <- merge(leishmaniagenes2, leishmaniagenes)
```

The first element of the merged list variable leishmaniagenes3 contains a vector of the *Leishmania major* gene names, the second has a vector of the types of those genes (e.g. protein-coding, pseudogene etc.), and the third element has a vector of the *Plasmodium falciparum* orthologs' names. We can therefore find out how many protein-coding *Leishmania major* genes lack *Plasmodium falciparum* orthologs by typing:

```
leishmaniagenenames <- leishmaniagenes3[[1]]
leishmaniagenebiotypes <- leishmaniagenes3[[2]]
leishmaniaPforthologues <- leishmaniagenes3[[3]]
myindex <- leishmaniaPforthologues=="" & leishmaniagenebiotypes=="protein_coding"
leishmaniagenenames2 <- leishmaniagenenames[myindex]
length(leishmaniagenenames2)
```

```
## [1] 6476
```

This tells us that there are 6476 *Leishmania major* protein-coding genes that lack *Plasmodium falciparum* orthologs.

# Chapter 42

# Summary

In this practical, you will have learned to use the following R functions:

- **useDataset()** to select a data set in a database to query (in the biomaRt package)
- **listDatasets()** to get a list of all data sets in a database (in the biomaRt package)
- **listAttributes()** to get a list of all features of a data set (in the biomaRt package)
- **getBM()** to make a query on a database (in the biomaRt package) merge() to merge R list objects that contain some named elements in common

# Chapter 43

# #Links and Further Reading

Some links are included here for further reading.

For background reading on comparative genomics, it is recommended to read Chapter 8 of Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

For more information and examples on using the biomaRt R package, see the biomaRt package website.

For a more in-depth introduction to R, a good online tutorial is available on the "Kickstarting R" website, cran.r-project.org/doc/contrib/Lemon-kickstart.

There is another nice (slightly more in-depth) tutorial to R available on the "Introduction to R" website, cran.r-project.org/doc/manuals/R-intro.html.

## 43.1 Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the Matlab case study on Chlamydia (http://www.computational-genomics .net/case_studies/chlamydia_demo.html) from the website that accompanies the book Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

## 43.2   Exercises

Answer the following questions, using the R package. For each question, please record your answer, and what you typed into R to get this answer.

Model answers to the exercises are given in Answers to the exercises on Comparative Genomics.

1. How many *Mycobacterium ulcerans* genes are there in the current version of the Ensembl Bacteria database? Note: the bacterium *Mycobacterium ulcerans* causes Buruli ulcer, which is classified by the WHO as a neglected tropical disease.
2. How many of the *Mycobacterium ulcerans* Ensembl genes are protein-coding genes?
3. How many *Mycobacterium ulcerans* protein-coding genes have *Mycobacterium leprae* orthologs? Note: *Mycobacterium leprae* is the bacterium that causes leprosy, which is classified by the WHO as a neglected tropical disease.
4. How many of the *Mycobacterium ulcerans* protein-coding genes have one-to-one orthologs in *Mycobacterium leprae*?
5. How many *Mycobacterium ulcerans* genes have Pfam domains?
6. What are the top 5 most common Pfam domains in *Mycobacterium ulcerans* genes?
7. How many copies of each of the top 5 domains found in Q6 are there in the *Mycobacterium ulcerans* protein set?
8. How many of copies are there in the *Mycobacterium leprae* protein set, of each of the top 5 *Mycobacterium ulcerans* Pfam protein domains?
9. Are the numbers of copies of some domains different in the two species?
10. Of the differences found in Q9, are any of the differences statistically significant?

# Chapter 44

# Hidden Markov Models

```
library(compbio4all)
```

**By**: Avril Coghlan.

**Adapted, edited and expanded**: Nathan Brouwer under the Creative Commons 3.0 Attribution License (CC BY 3.0) with assistance from Havannah Tung.

**NOTE: this chapter has not yet been significaly revised**.

## 44.1 A multinomial model of DNA sequence evolution

The simplest model of DNA sequence evolution assumes that the sequence has been produced by a random process that randomly chose any of the four nucleotides at each position in the sequence, where the probability of choosing any one of the four nucleotides depends on a predetermined probability distribution. That is, the four nucleotides are chosen with pA, pC, pG, and pT respectively. This is known as the multinomial sequence model.

A multinomial model for DNA sequence evolution has four parameters: the probabilities of the four nucleotides pA, pC, pG, and pT. For example, say we may create a multinomial model where pA=0.2, pC=0.3, pG=0.3, and pT=0.2. This means that the probability of choosing a A at any particular sequence position is set to be 0.2, the probability of choosing a C is 0.3, of choosing a G is 0.3, and of choosing a T is 0.2. Note that pA + pC + pG + pT = 1, as the sum of the probabilities of the four different types of nucleotides must be equal to 1, as there are only four possible types of nucleotide.

The multinomial sequence model is like having a roulette wheel that is divided into four different slices labeled "A", "T", "G" and "C", where the pA, pT, pG

and pC are the fractions of the wheel taken up by the slices with these four labels. If you spin the arrow attached to the center of the roulette wheel, the probability that it will stop in the slice with a particular label (eg. the slice labeled "A") only depends on the fraction of the wheel taken up by that slice (pA here; SEE THE PICTURE BELOW).

## 44.2    Generating a DNA sequence using a multi-nomial model

We can use R to generate a DNA sequence using a particular multinomial model. First we need to set the values of the four parameters of the multinomial model, the probabilities pA, pC, pG, and pT of choosing the nucleotides A, C, G and T, respectively, at a particular position in the DNA sequence. For example, say we decide to set pA=0.2, pC=0.3, pG=0.3, and pT=0.2. We can use the function sample() in R to generate a DNA sequence of a certain length, by selecting a nucleotide at each position according to this probability distribution:

Define the alphabet of nucleotides

```r
nucleotides    <- c("A", "C", "G", "T")
```

Set the values of the probabilities

```r
probabilities1 <- c(0.2, 0.3, 0.3, 0.2)
```

Set the length of the sequence

```r
seqlength      <- 30
```

Generate a sequence

```r
sample(nucleotides,
       seqlength,
       rep=TRUE,
       prob=probabilities1)
```

```
##  [1] "G" "G" "G" "A" "G" "G" "C" "T" "G" "G" "A" "T" "G" "G" "G" "C" "T" "T" "C"
## [20] "G" "A" "A" "G" "G" "A" "T" "C" "C" "G" "G"
```

If you look at the help page for the function, you will find that its inputs are the vector to sample from (nucleotides here), the size of the sample (seqlength here), and a vector of probabilities for obtaining the elements of the vector being sampled (probabilities1 here). If we use the sample() function to generate a sequence again, it will create a different sequence using the same multinomial model:

```r
# Generate another sequence
sample(nucleotides,
       seqlength,
```

```
      rep=TRUE,
      prob=probabilities1)
```

```
##  [1] "A" "T" "C" "G" "G" "G" "A" "C" "G" "A" "C" "T" "A" "T" "G" "T" "C" "T" "T"
## [20] "G" "A" "C" "C" "G" "G" "C" "T" "G" "C" "A"
```

In the same way, we can generate a sequence using a different multinomial model, where pA=0.1, pC=0.41, pG=0.39, and pT=0.1:

```
# Set the values of the probabilities for the new model
probabilities2 <- c(0.1, 0.41, 0.39, 0.1)

# Generate a sequence
sample(nucleotides, seqlength, rep=TRUE, prob=probabilities2)
```

```
##  [1] "C" "C" "T" "A" "C" "C" "G" "C" "A" "A" "G" "G" "A" "G" "G" "A" "C" "C" "C"
## [20] "G" "C" "A" "C" "C" "G" "G" "G" "C" "T" "G"
```

As you would expect, the sequences generated using this second multinomial model have a higher fraction of Cs and Gs compared to the sequences generated using the first multinomial model above. This is because pC and GT are higher for this second model than for the first model (pC=0.41 and GT=0.39 in the second model, versus pC=0.3 and GT=0.3 in the first model). That is, in the second multinomial model we are using a roulette wheel that has large slices labeled "C" and "G", while in the first multinomial model we were using a roulette wheel with relatively smaller slices labeled "C" and "G" (SEE THE PICTURE BELOW).

## 44.3   A Markov model of DNA sequence evolution

A multinomial model of DNA sequence evolution is a good model of the evolution of many DNA sequences. However, for some DNA sequences, a multinomial model is not an accurate representation of how the sequences have evolved. One reason is that a multinomial model assumes that each part of the sequence (eg. the first 100 nucleotides of the sequence, the second 100 nucleotides, the third 100 nucleotides, etc.) have the same frequency of each type of nucleotide (the same pA, pC, pG, and pT), and this may not be true for a particular DNA sequence if there are considerable differences in nucleotide frequencies in different parts of the sequence.

Another assumption of a multinomial model of DNA sequence evolution is that the probability of choosing a particular nucleotide (eg. "A") at a particular position in the sequence only depends on the predetermined frequency of that nucleotide (pA here), and does not depend at all on the nucleotides found at adjacent positions in the sequence. This assumption holds true for many DNA

sequences. However, for some DNA sequences, it is not true, because the probability of finding a particular nucleotide at a particular position in the sequence does depend on what nucleotides are found at adjacent positions in the sequence. In this case, a different type of DNA sequence model called a Markov sequence model is a more accurate representation of the evolution of the sequence.

A Markov sequence model assumes that the sequence has been produced by a process that chose any of the four nucleotides in the sequence, where the probability of choosing any one of the four nucleotides at a particular position depends on the nucleotide chosen for the previous position. That is, if "A" was chosen at the previous position, then the probability of choosing any one of the four nucleotides at the current position depends on a predetermined probability distribution. That is, given that "A" was chosen at the previous position, the four nucleotides are chosen at the current position with probabilities of pA, pC, pG, and pT of choosing "A", "C", "G", or "T", respectively (eg. pA=0.2, pC=0.3, pG=0.3, and pT=0.2). In contrast, if "C" was chosen at the previous position, then the probability of choosing any one of the four nucleotides at the current position depends on a different predetermined probability distribution, that is, the probabilities of choosing "A", "C", "G", or "T" at the current position are now different (eg. pA=0.1, pC=0.41, pG=0.39, and pT=0.1).

A Markov sequence model is like having four different roulette wheels, labeled "afterA", "afterT", "afterG", and "afterC", for the cases when "A", "T", "G", or "C" were chosen at the previous position in a sequence, respectively. Each of the four roulette wheels has four slices labeled "A", "T", "G", and "C", but in each roulette wheel a different fraction of the wheel is taken up by the four slices. That is, each roulette wheel has a different pA, pT, pG and pC. If we are generating a new DNA sequence using a Markov sequence model, to decide what nucleotide to choose at a particular position in the sequence, you spin the arrow at the center of a roulette wheel, and see in which slice the arrow stops. There are four roulette wheels, and the particular roulette wheel we use at a particular position in the sequence depends on the nucleotide chosen for the previous position in the sequence. For example, if "T" was chosen at the previous position, we use the "afterT" roulette wheel to choose the nucleotide for the current position. The probability of choosing a particular nucleotide at the current position (eg. "A") then depends on the fraction of the "afterT" roulette wheel taken up by the the slice labeled with that nucleotide (pA here; SEE THE PICTURE BELOW).

## 44.4   The transition matrix for a Markov model

A multinomial model of DNA sequence evolution just has four parameters: the probabilities pA, pC, pG, and pT. In contrast, a Markov model has many more parameters: four sets of probabilities pA, pC, pG, and pT, that differ according to whether the previous nucleotide was "A", "G", "T" or "C". The symbols pAA, pAC, pAG, and pAT are usually used to represent the four probabilities for the

case where the previous nucleotide was "A", the symbols pCA, pCC, pCG, and pCT for the case when the previous nucleotide was "C", and so on.

It is common to store the probability parameters for a Markov model of a DNA sequence in a square matrix, which is known as a Markov transition matrix. The rows of the transition matrix represent the nucleotide found at the previous position in the sequence, while the columns represent the nucleotides that could be found at the current position in the sequence. In R, you can create a matrix using the *matrix()* command, and the rownames() and colnames() functions can be used to label the rows and columns of the matrix. For example, to create a transition matrix, we type:

Define the alphabet of nucleotides

```
nucleotides          <- c("A", "C", "G", "T")
```

Set the values of the probabilities, where the previous nucleotide was "A"

```
afterAprobs <- c(0.2, 0.3, 0.3, 0.2)
```

Set the values of the probabilities, where the previous nucleotide was "C"

```
afterCprobs <- c(0.1, 0.41, 0.39, 0.1)
```

Set the values of the probabilities, where the previous nucleotide was "G"

```
afterGprobs <- c(0.25, 0.25, 0.25, 0.25)
```

Set the values of the probabilities, where the previous nucleotide was "T"

```
afterTprobs <- c(0.5, 0.17, 0.17, 0.17)
```

Create a 4 x 4 matrix

```
mytransitionmatrix <- matrix(c(afterAprobs,
                               afterCprobs,
                               afterGprobs,
                               afterTprobs),
                             4, 4, byrow = TRUE)

rownames(mytransitionmatrix) <- nucleotides
colnames(mytransitionmatrix) <- nucleotides
```

Print out the transition matrix

```
mytransitionmatrix
```

```
##      A    C    G    T
## A 0.20 0.30 0.30 0.20
## C 0.10 0.41 0.39 0.10
## G 0.25 0.25 0.25 0.25
## T 0.50 0.17 0.17 0.17
```

Rows 1, 2, 3 and 4 of the transition matrix give the probabilities pA, pC, pG, and pT for the cases where the previous nucleotide was "A", "C", "G", or "T", respectively. That is, the element in a particular row and column of the transition matrix (eg. the row for "A", column for "C") holds the probability (pAC) of choosing a particular nucleotide ("C") at the current position in the sequence, given that was a particular nucleotide ("A") at the previous position in the sequence.

## 44.5   Generating   a   DNA   sequence   using   a   Markov model

Just as you can generate a DNA sequence using a particular multinomial model, you can generate a DNA sequence using a particular Markov model. When you are generating a DNA sequence using a Markov model, the nucleotide chosen at each position at the sequence depends on the nucleotide chosen at the previous position. As there is no previous nucleotide at the first position in the new sequence, we need to define the probabilities of choosing "A", "C", "G" or "T" for the first position. The symbols piA, piC, piG, and piT are used to represent the probabilities of choosing "A", "C", "G", or "T" at the first position.

We will use an *R* function `generate_markov_seq()` (Coghlan 2011) to generate a DNA sequence using a particular Markov model.

The function `generate_markov_seq()` takes as its arguments (inputs) the transition matrix for the particular Markov model; a vector containing the values of piA, piC, piG, and piT; and the length of the DNA sequence to be generated.

The probabilities of choosing each of the four nucleotides at the first position in the sequence are piA, piC, piG, and piT. The probabilities of choosing each of the four nucleotides at the second position in the sequence depend on the particular nucleotide that was chosen at the first position in the sequence. The probabilities of choosing each of the four nucleotides at the third position depend on the nucleotide chosen at the second position, and so on.

We can use the `generate_markov_seq()` function to generate a sequence using a particular Markov model. For example, to create a sequence of 30 nucleotides using the Markov model described in the transition matrix mytransitionmatrix, using uniform starting probabilities (ie. piA = 0.25, piC = 0.25, piG = 0.25, and piT = 0.25) , we type:

```
myinitialprobs <- c(0.25, 0.25, 0.25, 0.25)

generate_markov_seq(mytransitionmatrix, myinitialprobs, 30)
```

```
##  [1] "C" "G" "A" "G" "G" "T" "C" "C" "C" "C" "A" "G" "T" "G" "A" "T" "T" "A" "T"
## [20] "C" "C" "G" "A" "C" "C" "T" "G" "G" "G" "G"
```

As you can see, there are many "A"s after "T"s in the sequence. This is because

pTA has a high value (0.5) in the Markov transition matrix mytransitionmatrix. Similarly, there are few "A"s or "T"s after "C"s, which is because pCA and pCT have low values (0.1) in this transition matrix.

## 44.6 A Hidden Markov Model of DNA sequence evolution

In a Markov model, the nucleotide at a particular position in a sequence depends on the nucleotide found at the previous position. In contrast, in a Hidden Markov model (HMM), the nucleotide found at a particular position in a sequence depends on the state at the previous nucleotide position in the sequence. The state at a sequence position is a property of that position of the sequence, for example, a particular HMM may model the positions along a sequence as belonging to either one of two states, "GC-rich" or "AT-rich". A more complex HMM may model the positions along a sequence as belonging to many different possible states, such as "**promoter**", "**exon**", "**intron**", and "**intergenic DNA**".

A HMM is like having several different roulette wheels, one roulette wheel for each state in the HMM, for example, a "GC-rich" and an "AT-rich" roulette wheel. Each of the roulette wheels has four slices labeled "A", "T", "G", and "C", and in each roulette wheel a different fraction of the wheel is taken up by the four slices. That is, the "GC-rich" and "AT-rich" roulette wheels have different pA, pT, pG and pC values. If we are generating a new DNA sequence using a HMM, to decide what nucleotide to choose at a particular sequence position, we spin the arrow of a particular roulette wheel, and see in which slice it stops.

How do we decide which roulette wheel to use? Well, if there are two roulette wheels, we tend to use the same roulette wheel that we used to choose the previous nucleotide in the sequence, but there is also a certain small probability of switching to the other roulette wheel. For example, if we used the "GC-rich" roulette wheel to choose the previous nucleotide in the sequence, there may be a 90% chance that we will use the "GC-rich" roulette wheel again to choose the nucleotide at the current position, but a 10% chance that we will switch to using the "AT-rich" roulette wheel to choose the nucleotide at the current position. Likewise, if we used the "AT-rich" roulette wheel to choose the nucleotide at the previous position, there may be 70% chance that we will use the "AT-rich" wheel again at this position, but a 30% chance that we will switch to using the "GC-rich" roulette wheel to choose the nucleotide at this position.

## 44.7   The transition matrix and emission matrix for a HMM

A HMM has two important matrices that hold its parameters. The first is the HMM transition matrix, which contains the probabilities of switching from one state to another. For example, in a HMM with two states, an AT-rich state and a GC-rich state, the transition matrix will hold the probabilities of switching from the AT-rich state to the GC-rich state, and of switching from the GC-rich state to the AT-rich state. For example, if the previous nucleotide was in the AT-rich state there may be a probability of 0.3 that the current nucleotide will be in the GC-rich state, and if the previous nucleotide was in the GC-rich state there may be a probability of 0.1 that the current nucleotide will be in the AT-rich state:

Define the names of the states

```
states              <- c("AT-rich", "GC-rich")
```

Set the probabilities of switching states, where the previous state was "AT-rich"

```
ATrichprobs       <- c(0.7, 0.3)
```

Set the probabilities of switching states, where the previous state was "GC-rich"

```
GCrichprobs       <- c(0.1, 0.9)
```

Create a 2 x 2 matrix

```
thetransitionmatrix <- matrix(c(ATrichprobs,
                                GCrichprobs),
                              2, 2,
                              byrow = TRUE)

rownames(thetransitionmatrix) <- states
colnames(thetransitionmatrix) <- states
```

Print out the transition matrix

```
thetransitionmatrix
```

```
##         AT-rich GC-rich
## AT-rich     0.7     0.3
## GC-rich     0.1     0.9
```

There is a row in the transition matrix for each of the possible states at the previous position in the nucleotide sequence. For example, in this transition matrix, the first row corresponds to the case where the previous position was in the "AT-rich" state, and the second row corresponds to the case where the previous position was in the "GC-rich" state. The columns give the probabilities of switching to different states at the current position. For example, the value

in the second row and first column of the transition matrix above is 0.1, which is the probability of switching to the AT-rich state, if the previous position of the sequence was in the GC-rich state.

The second important matrix is the HMM emission matrix, which holds the probabilities of choosing the four nucleotides "A", "C", "G", and "T", in each of the states. In a HMM with an AT-rich state and a GC-rich state, the emission matrix will hold the probabilities of choosing each of the four nucleotides "A", "C", "G" and "T" in the AT-rich state (for example, pA=0.39, pC=0.1, pG=0.1, and pT=0.41 for the AT-rich state), and the probabilities of choosing "A", "C", "G", and "T" in the GC-rich state (for example, pA=0.1, pC=0.41, pG=0.39, and pT=0.1 for the GC-rich state).

```r
# Define the alphabet of nucleotides
nucleotides           <- c("A", "C", "G", "T")

# Set the values of the probabilities, for the AT-rich state
ATrichstateprobs     <- c(0.39, 0.1, 0.1, 0.41)

# Set the values of the probabilities, for the GC-rich state
GCrichstateprobs     <- c(0.1, 0.41, 0.39, 0.1)

# Create a 2 x 4 matrix
theemissionmatrix <- matrix(c(ATrichstateprobs,
                              GCrichstateprobs),
                            2, 4,
                            byrow = TRUE)

rownames(theemissionmatrix) <- states
colnames(theemissionmatrix) <- nucleotides

# Print out the emission matrix
theemissionmatrix
```

```
##            A    C    G    T
## AT-rich 0.39 0.10 0.10 0.41
## GC-rich 0.10 0.41 0.39 0.10
```

There is a row in the emission matrix for each possible state, and the columns give the probabilities of choosing each of the four possible nucleotides when in a particular state. For example, the value in the second row and third column of the emission matrix above is 0.39, which is the probability of choosing a "G" when in the "GC-rich state" (ie. when using the "GC-rich" roulette wheel).

## 44.8   Generating a DNA sequence using a HMM

The following function `generate_hmm_seq()` can be used to generate a DNA sequence using a particular HMM. As its arguments (inputs), it requires the parameters of the HMM: the HMM transmission matrix and HMM emission matrix.

When you are generating a DNA sequence using a HMM, the nucleotide is chosen at each position depending on the state at the previous position in the sequence. As there is no previous nucleotide at the first position in the sequence, the function `generate_hmm_seq()` also requires the probabilities of the choosing each of the states at the first position (eg. piAT-rich and piGC-rich being the probability of the choosing the "AT-rich" or "GC-rich" states at the first position for a HMM with these two states).

We can use the `generate_hmm_seq()` function to generate a sequence using a particular HMM. For example, to create a sequence of 30 nucleotides using the HMM with "AT-rich" and "GC-rich" states described in the transition matrix thetransitionmatrix, the emission matrix theemissionmatrix, and uniform starting probabilities (ie. piAT-rich = 0.5, piGC-rich = 0.5), we type:

```r
theinitialprobs <- c(0.5, 0.5)

generate_hmm_seq(thetransitionmatrix,
                 theemissionmatrix,
                 theinitialprobs, 30)
```

```
## [1] "Position 1 , State GC-rich , Nucleotide =  A"
## [1] "Position 2 , State GC-rich , Nucleotide =  G"
## [1] "Position 3 , State GC-rich , Nucleotide =  T"
## [1] "Position 4 , State GC-rich , Nucleotide =  G"
## [1] "Position 5 , State GC-rich , Nucleotide =  G"
## [1] "Position 6 , State GC-rich , Nucleotide =  C"
## [1] "Position 7 , State GC-rich , Nucleotide =  C"
## [1] "Position 8 , State GC-rich , Nucleotide =  G"
## [1] "Position 9 , State GC-rich , Nucleotide =  C"
## [1] "Position 10 , State GC-rich , Nucleotide =  C"
## [1] "Position 11 , State GC-rich , Nucleotide =  C"
## [1] "Position 12 , State GC-rich , Nucleotide =  G"
## [1] "Position 13 , State GC-rich , Nucleotide =  C"
## [1] "Position 14 , State GC-rich , Nucleotide =  C"
## [1] "Position 15 , State AT-rich , Nucleotide =  T"
## [1] "Position 16 , State AT-rich , Nucleotide =  A"
## [1] "Position 17 , State AT-rich , Nucleotide =  T"
## [1] "Position 18 , State GC-rich , Nucleotide =  G"
## [1] "Position 19 , State GC-rich , Nucleotide =  C"
## [1] "Position 20 , State GC-rich , Nucleotide =  C"
```

```
## [1] "Position 21 , State GC-rich , Nucleotide =  C"
## [1] "Position 22 , State GC-rich , Nucleotide =  G"
## [1] "Position 23 , State GC-rich , Nucleotide =  C"
## [1] "Position 24 , State GC-rich , Nucleotide =  C"
## [1] "Position 25 , State GC-rich , Nucleotide =  G"
## [1] "Position 26 , State GC-rich , Nucleotide =  G"
## [1] "Position 27 , State GC-rich , Nucleotide =  C"
## [1] "Position 28 , State AT-rich , Nucleotide =  A"
## [1] "Position 29 , State AT-rich , Nucleotide =  A"
## [1] "Position 30 , State AT-rich , Nucleotide =  A"
```

As you can see, the nucleotides generated by the GC-rich state are mostly but not all "G"s and "C"s (because of the high values of pG and pC for the GC-rich state in the HMM emission matrix), while the nucleotides generated by the AT-rich state are mostly but not all "A"s and "T"s (because of the high values of pT and pA for the AT-rics state in the HMM emission matrix).

Furthermore, there tends to be runs of nucleotides that are either all in the GC-rich state or all in the AT-rich state, as the transition matrix specifies that the probabilities of switching from the AT-rich to GC-rich state (probability 0.3), or GC-rich to AT-rich state (probability 0.1) are relatively low.

## 44.9   Inferring the states of a HMM that generated a DNA sequence

If we have a HMM with two states, "GC-rich" and "AT-rich", and we know the transmission and emission matrices of the HMM, can we take some new DNA sequence, and figure out which state (GC-rich or AT-rich) is the most likely to have generated each nucleotide position in that DNA sequence? This is a common problem in bioinformatics. It is called the problem of finding the most probable state path, as it essentially consists of assigning the most likely state to each position in the DNA sequence. The problem of finding the most probable state path is also sometimes called segmentation. For example, give a DNA sequence of 1000 nucleotides, you may wish to use your HMM to segment the sequence into blocks that were probably generated by the "GC-rich" state or by the "AT-rich" state.

The problem of finding the most probable state path given a HMM and a sequence (ie. the problem of segmenting a sequence using a HMM), can be solved by an algorithm called the Viterbi algorithm. As its output, the Viterbi algorithm gives for each nucleotide position in a DNA sequence, the state of your HMM that most probably generated the nucleotide in that position. For example, if you segmented a particular DNA sequence of 1000 nucleotides using a HMM with "AT-rich" and "GC-rich" states, the Viterbi algorithm may tell you that nucleotides 1-343 were most probably generated by the AT-rich state, nucleotides 344-900 were most probably generated by the GC-rich state, and

901-1000 were most probably generated by the AT-rich state.

The function `viterbi()` (Coghlan 2011) is a function for the Viterbi algorithm. The `viterbi()` function requires a second function `make_viterbi_mat()`. Given a HMM, and a particular DNA sequence, you can use the `viterbi()` function to find the state of that HMM that was most likely to have generated the nucleotide at each position in the DNA sequence:

```r
myseq <- c("A", "A", "G", "C", "G", "T", "G", "G", "G", "G", "C", "C", "C", "C", "G",
```

```r
compbio4all::viterbi(myseq, thetransitionmatrix, theemissionmatrix)
```

```
## [1] "Positions 1 - 2 Most probable state =  AT-rich"
## [1] "Positions 3 - 21 Most probable state =  GC-rich"
## [1] "Positions 22 - 22 Most probable state =  AT-rich"
## [1] "Positions 23 - 30 Most probable state =  GC-rich"
```

## 44.10   A Hidden Markov Model of protein sequence evolution

We have so far talked about using HMMs to model DNA sequence evolution. However, it is of course possible to use HMMs to model protein sequence evolution. When using a HMM to model DNA sequence evolution, we may have states such as "AT-rich" and "GC-rich". Similarly, when using a HMM to model protein sequence evolution, we may have states such as "hydrophobic" and "hydrophilic". In a protein HMM with "hydrophilic" and "hydrophilic" states, the "hydrophilic" HMM will have probabilities pA, pR, pC... of choosing each of the 20 amino acids alanine (A), arginine (R), cysteine (C), etc. when in that state. Similarly, the "hydrophilic" state will have different probabilities pA, pR, pC... of choosing each of the 20 amino acids. The probability of choosing a hydrophobic amino acid such as alanine will be higher in the "hydrophobic" state than in the "hydrophilic" state (ie. pA of the "hydrophobic" state will be higher than the pA of of the "hydrophilic" state, where A represents alanine here). A HMM of protein sequence evolution also defines a certain probability of switching from the "hydrophilic" state to the "hydrophobic" state, and a certain probability of switching from the "hydrophobic" state to the "hydrophilic" state.

## 44.11   Summary

In this practical, you will have learned to use the following R functions:

1. `numeric()` for making a vector for storing numbers
2. `character()` for making a vector for storing characters
3. `matrix()` for making a matrix variable
4. `rownames()` for assigning names to the rows of a matrix variable
5. `colnames()` for assigning names to the columns of a matrix variable

6. `sample()` for making a random sample of numbers from a vector of numbers

All of these functions belong to the standard installation of *R*.

## 44.12  Links and Further Reading

Some links are included here for further reading, which will be especially useful if you need to use the R package for your project or assignments.

For background reading on multinomial models, Markov models, and HMMs, it is recommended to read Chapters 1 and 4 of Introduction to Computational Genomics: a case studies approach by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

There is also a very nice chapter on "Markov Models" in the book Applied statistics for bioinformatics using R by Krijnen (available online at cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf).

## 44.13  Exercises

Answer the following questions, using the R package. For each question, please record your answer, and what you typed into R to get this answer.

1. In a previous practical, you saw that the Bacteriophage lambda genome sequence (NCBI accession NC_001416) has long stretches of either very GC-rich (mostly in the first half of the genome) or very AT-rich sequence (mostly in the second half of the genome). Use a HMM with two different states ("AT-rich" and "GC-rich") to infer which state of the HMM is most likely to have generated each nucleotide position in the Bacteriophage lambda genome sequence. For the AT-rich state, set pA= 0.27, pC= 0.2084, pG= 0.198, and pT= 0.3236. For the GC-rich state, set pA= 0.2462, pC= 0.2476, pG= 0.2985, and pT= 0.2077. Set the probability of switching from the AT-rich state to the GC-rich state to be 0.0002, and the probability of switching from the GC-rich state to the AT-rich state to be 0.0002. What is the most probable state path?

2. Given a HMM with four different states ("A-rich", "C-rich", "G-rich" and "T-rich"), infer which state of the HMM is most likely to have generated each nucleotide position in the Bacteriophage lambda genome sequence. For the A-rich state, set pA= 0.3236, pC= 0.2084, pG= 0.198, and pT= 0.27. For the C-rich state, set pA= 0.2462, pC= 0.2985, pG= 0.2476, and pT= 0.2077. For the G-rich state, set pA= 0.2462, pC= 0.2476, pG= 0.2985, and pT= 0.2077. For the T-rich state, set pA= 0.27, pC= 0.2084, pG= 0.198, and pT= 0.3236. Set the probability of switching from the A-rich state to any of the three other states to be 6.666667e-05. Likewise, set the probability of switching from the C-rich/G-rich/T-rich state to any

of the three other states to be 6.666667e-05. What is the most probable state path? Do you find differences between these results and the results from simply using a two-state HMM (as in Q1)?

3. Make a two-state HMM to model protein sequence evolution, with "hydrophilic" and "hydrophobic" states. For the hydrophilic state, set pA= 0.02, pR= 0.068, pN= 0.068, pD= 0.068, pC= 0.02, pQ= 0.068, pE= 0.068, pG= 0.068, pH= 0.068, pI= 0.012, pL= 0.012, pK= 0.068, pM= 0.02, pF= 0.02, pP= 0.068, pS= 0.068, pT= 0.068, pW= 0.068, pY= 0.068, and pV= 0.012. For the hydrophobic state, set pA= 0.114, pR= 0.007, pN= 0.007, pD= 0.007, pC= 0.114, pQ= 0.007, pE= 0.007, pG= 0.025, pH= 0.007, pI= 0.114, pL= 0.114, pK= 0.007, pM= 0.114, pF= 0.114, pP= 0.025, pS= 0.026, pT= 0.026, pW= 0.025, pY= 0.026, and pV= 0.114. Set the probability of switching from the hydrophilic state to the hydrophobic state to be 0.01. Set the probability of switching from the hydrophobic state to the hydrophilic state to be 0.01. Now infer which state of the HMM is most likely to have generated each amino acid position in the the human odorant receptor 5BF1 protein (UniProt accession Q8NHC7). What is the most probable state path? The odorant receptor is a 7-transmembrane protein, meaning that it crosses the cell membrane seven times. As a consequence the protein has seven hydrophobic regions that cross the fatty cell membrane, and seven hydrophilic segments that touch the watery cytoplasm and extracellular environments. What do you think are the coordinates in the protein of the seven transmembrane regions?

# Chapter 45

# Logarithms in R

**By** Nathan Brouwer

Logging splits up multiplication into addition. So, log(m*n) is the same as log(m) + log(n)

You can check this

```r
m<-10
n<-11
log(m*n)
```

```
## [1] 4.70048
```

```r
log(m)+log(n)
```

```
## [1] 4.70048
```

```r
log(m*n) == log(m)+log(n)
```

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

Exponentiation undos logs

```r
exp(log(m*n))
```

```
## [1] 110
```

```r
m*n
```

```
## [1] 110
```

The key equation in BLAST's E values is

u = ln(K$m$n)/lambda

This can be changed to

[ln(K) + ln(mn)]/lambda

We can check this

```r
K <- 1
m <- 10
n <- 11
lambda <- 110

log(K*m*n)/lambda
```

```
## [1] 0.04273164
```

```r
(log(K) + log(m*n))/lambda
```

```
## [1] 0.04273164
```

```r
log(K*m*n)/lambda == (log(K) + log(m*n))/lambda
```

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

# Part I

# Appendices

# Appendix 01: Getting access to R

## 45.1 Getting Started With R and RStudio

- R is a piece of software that does calculations and makes graphs.
- RStudio is a GUI (graphical user interface) that acts as a front-end to R
- Your can use R directly, but most people use a GUI of some kind
- RStudio has become the most popular GUI

The following instructions will lead you click by click through downloading R and RStudio and starting an initial session. If you have trouble with downloading either program go to YouTube and search for something like "Downloading R" or "Installing RStudio" and you should be able to find something helpful, such as "How to Download R for Windows".

### 45.1.1 RStudio Cloud

TODO: Add RStudio cloud

### 45.1.2 Getting R onto your own computer

To get R on to your computer first go to the CRAN website at https://cran.r-project.org/ (CRAN stands for "comprehensive R Archive Network"). At the top of the screen are three bullet points; select the appropriate one (or click the link below)

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

Each page is formatted slightly differently. For a current Mac, click on the top link, which as of 8/16/2018 was "R-3.5.1.pkg" or click this link. If you have an older Mac you might have to scroll down to find your operating system under "Binaries for legacy OS X systems."

For PC select "base" or click this link.

When its downloaded, run the installer and accept the defaults.

### 45.1.3 Getting RStudio onto your computer

RStudio is an R interface developed by a company of the same name. RStudio has a number of commercial products, but much of their portfolio is freeware. You can download RStudio from their website www.rstudio.com/ . The download page (www.rstudio.com/products/rstudio/download/) is a bit busy because it shows all of their commercial products; the free version is on the far left side of the table of products. Click on the big green DOWNLOAD button under the column on the left that says "RStudio Desktop Open Source License" (or click on this link ).

This will scroll you down to a list of downloads titled "Installers for Supported Platforms." Windows users can select the top option RStudio 1.1.456 - Windows Vista/7/8/10 and Mac the second option RStudio 1.1.456 - Mac OS X 10.6+ (64-bit). (Versions names are current of 8/16/2018).

Run the installer after it downloads and accept the default. RStudio will automatically link up with the most current version of R you have on your computer. Find the RStudio icon on your desktop or search for "RStudio" from your task bar and you'll be read to go.

### 45.1.4 Keep R and RStudio current

Both R and RStudio undergo regular updates and you will occasionally have to re-download and install one or both of them. In practice I probably do this about every 6 months.

# Getting started with R itself (or not)

## Vocabulary

- console
- script editor / source viewer
- interactive programming
- scripts / script files
- .R files
- text files / plain text files
- command execution / execute a command from script editor
- comments / code comments
- commenting out / commenting out code
- stackoverflow.com
- the rstats hashtag

## R commands

- c(...)
- mean(...)
- sd(...)
- ?
- read.csv(...)

This is a walk-through of a very basic R session. It assumes you have successfully installed R and RStudio onto your computer, and nothing else.

Most people who use R do not actually use the program itself - they use a GUI (graphical user interface) "front end" that make R a bit easier to use. However, you will probably run into the icon for the underlying R program on your desktop or elsewhere on your computer. It usually looks like this:

ADD IMAGE HERE

The long string of numbers have to do with the version and whether is 32 or 64 bit (not important for what we do).

If you are curious you can open it up and take a look - it actually looks a lot like RStudio, where we will do all our work (or rather, RStudio looks like R). Sometimes when people are getting started with R they will accidentally open R instead of RStudio; if things don't seem to look or be working the way you think they should, you might be in R, not RStudio

#### 45.1.4.1 R's console as a scientific calculator

You can interact with R's console similar to a scientific calculator. For example, you can use parentheses to set up mathematical statements like

```
5*(1+1)
```

```
## [1] 10
```

Note however that you have to be explicit about multiplication. If you try the following it won't work.

```
5(1+1)
```

R also has built-in functions that work similar to what you might have used in Excel. For example, in Excel you can calculate the average of a set of numbers by typing "=average(1,2,3)" into a cell. R can do the same thing except

- The command is "mean"
- You don't start with "="
- You have to package up the numbers like what is shown below using "c(…)"

```
mean(c(1,2,3))
```

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

Where "c(…)" packages up the numbers the way the mean() function wants to see them.

If you just do the following R will give you an answer, but its the wrong one

```
mean(1,2,3)
```

**This is a common issue with R – and many programs, really – it won't always tell you when somethind didn't go as planned. This is because it doesn't know something didn't go as planned; you have to learn the rules R plays by.**

#### 45.1.4.2 Practice: math in the console

See if you can reproduce the following results

**Division**

```
10/3
```

```
## [1] 3.333333
```

**The standard deviation**

```
sd(c(5,10,15)) # note the use of "c(...)"
```

```
## [1] 5
```

### 45.1.4.3 The script editor

While you can interact with R directly within the console, the standard way to work in R is to write what are known as **scripts.** These are computer code instructions written to R in a **script file.** These are save with the extension **.R** but area really just a form of **plain text file.**

To work with scripts, what you do is type commands in the script editor, then tell R to **excute** the command. This can be done several ways.

First, you tell RStudio the line of code you want to run by either * Placing the cursor at the end a line of code, OR * Clicking and dragging over the code you want to run in order highlight it.

Second, you tell RStudio to run the code by * Clicking the "Run" icon in the upper right hand side of the script editor (a grey box with a green error emerging from it) * pressing the control key ("ctrl)" and then then enter key on the keyboard

The code you've chosen to run will be sent by RStudio from the script editor over to the console. The console will show you both the code and then the output.

You can run several lines of code if you want; the console will run a line, print the output, and then run the next line. First I'll use the command mean(), and then the command sd() for the standard deviation:

```
mean(c(1,2,3))
```

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

```
sd(c(1,2,3))
```

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

### 45.1.4.4 Comments

One of the reasons we use script files is that we can combine R code with **comments** that tell us what the R code is doing. Comments are preceded by the hashtag symbol #. Frequently we'll write code like this:

```r
#The mean of 3 numbers
mean(c(1,2,3))
```

If you highlight all of this code (including the comment) and then click on "run", you'll see that RStudio sends all of the code over console.

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

Comments can also be placed at the *end* of a line of code

```r
mean(c(1,2,3)) #Note  the use of c(...)
```

Sometimes we write code and then don't want R to run it. We can prevent R from executing the code even if its sent to the console by putting a "#" *infront* of the code.

If I run this code, I will get just the mean but not the sd.

```r
mean(c(1,2,3))
#sd(c(1,2,3))
```

Doing this is called **commenting out** a line of code.

## 45.2   Help!

There are many resource for figuring out R and RStudio, including

- R's built in "help" function
- Q&A websites like **stackoverflow.com**
- twitter, using the hashtag #rstats
- blogs
- online books and course materials

### 45.2.1   Getting "help" from R

If you are using a function in R you can get info about how it works like this

```r
?mean
```

In RStudio the help screen should appear, probably above your console. If you start reading this help file, though, you don't have to go far until you start seeing lots of R lingo, like "S3 method","na.rm", "vectors". Unfortunately, the R help files are usually not written for beginners, and reading help files is a skill you have to acquire.

For example, when we load data into R in subsequent lessons we will use a function called "read.csv"

Access the help file by typing "?read.csv" into the console and pressing enter. Surprisingly, the function that R give you the help file isn't what you asked for,

but is read.table(). This is a related function to read.csv, but when you're a beginner thing like this can really throw you off.

Kieran Healy as produced a great cheatsheet for reading R's help pages as part of his forthcoming book. It should be available online at http://socviz.co/appendix.html#a-little-more-about-r

### 45.2.2 Getting help from the internet

The best way to get help for any topic is to just do an internet search like this: "R read.csv". Usually the first thing on the results list will be the R help file, but the second or third will be a blog post or something else where a usually helpful person has discussed how that function works.

Sometimes for very basic R commands like this might not always be productive but its always work a try. For but things related to stats, plotting, and programming there is frequently lots of information. Also try searching YouTube.

### 45.2.3 Getting help from online forums

Often when you do an internet search for an R topic you'll see results from the website www.stackoverflow.com, or maybe www.crossvalidated.com if its a statistics topic. These are excellent resources and many questions that you may have already have answers on them. Stackoverflow has an internal search function and also suggests potentially relevant posts.

Before posting to one of these sites yourself, however, do some research; there is a particular type and format of question that is most likely to get a useful response. Sadly, people new to the site often get "flamed" by impatient pros.

### 45.2.4 Getting help from twitter

Twitter is a surprisingly good place to get information or to find other people knew to R. Its often most useful to ask people for learning resources or general reference, but you can also post direct questions and see if anyone responds, though usually its more advanced users who engage in twitter-based code discussion.

A standard tweet might be "Hey #rstats twitter, am knew to #rstats and really stuck on some of the basics. Any suggestions for good resources for someone starting from scratch?"

## 45.3 Other features of RStudio

### 45.3.1 Ajusting pane the layout

You can adjust the location of each of RStudio 4 window panes, as well as their size.

To set the pane layout go to 1. "Tools" on the top menu 1. "Global options" 1. "Pane Layout"

Use the drop-down menus to set things up. I recommend 1. Lower left: "Console"" 1. Top right: "Source" 1. Top left: "Plot, Packages, Help Viewer" 1. This will leave the "Environment…" panel in the lower right.

### 45.3.2   Adjusting size of windows

You can clicked on the edge of a pane and adjust its size. For most R work we want the console to be big. For beginners, the "Environment, history, files" panel can be made really small.

## 45.4   Practice (OPTIONAL)

Practice the following operations. Type the directly into the console and execute them. Also write them in a script in the script editor and run them.

**Square roots**

```
sqrt(42)
```

```
## [1] 6.480741
```

**The date** Some functions in R can be executed within nothing in the parentheses.

```
date()
```

```
## [1] "Tue Jul 27 11:08:07 2021"
```

**Exponents** The ^ is used for exponents

```
42^2
```

```
## [1] 1764
```

**A series of numbers** A colon between two numbers creates a series of numbers.

```
1:42
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
```

**logs** The default for the log() function is the natural log.

```
log(42)
```

```
## [1] 3.73767
```

log10() gives the base-10 log.

```r
log10(42)
```

```
## [1] 1.623249
```

**exp() raises e to a power**

```r
exp(3.73767)
```

```
## [1] 42.00002
```

**Multiple commands can be nested**

```r
sqrt(42)^2
log(sqrt(42)^2)
exp(log(sqrt(42)^2))
```

# Chapter 46

# BLAST Summary

**By**: Nathan Brouwer

```
library(compbio4all)
```

I have discovered that there are many minor but annoying inconsistencies in how the BLAST algorithm is described in text books. Additionally, most gloss over key computational steps, such as how BLAST actually searches through sequences. While this doesn't have many - if any - consequences for a user of BLAST, it makes it very difficult to determine the computational steps involved and to appreciate exactly what is going on. The following notes are my attempt to summarize all the key features and computational steps of BLAST.

In a few cases I have not provided every detail, which I usually note. I also occasionally use my own analogy or terminology which I think provides a useful description.

```
In  the  combio4all package I have written code that simulates some
steps of the BLAST algorithm.  This code is meant to emulate
or evoke the process of BLAST and is NOT written as the actual
program is.  For example, as discussed below, BLAST's actual
searching of databases involves computational tool called a
**deterministic finite-state automata (DFSA)** (Zvelebil & Baum
2008).  This is a rather complicated sub-program, buts it general
behavior can be evoked using a simple regular expression orfor()`
```
loop. (I'd like to write code for a DFSA but have yet to find R or Python examples).

## 46.1   BLAST Overview

### 46.1.1   Why do a BLAST search?

BLASTing is a common activity by molecular biologists, bioinformaticians, computational biologists, and phylogenists. Some general reasons to do a BLAST

- Find conserved sequences (homologs)
- Find conserved regions within a gene which may be important to protein function
- Determine how similar a gene in a model species (mice, Drosophila) is to humans
- Scan a new genome sequence for conserved genes
- Ex: BLASTing sequence of newly discovered mouse gene (shroom) showed Hildebrand & Soriano (1999) that this gene wasn't just a mouse gene - it occurred in Humans AND mice
- Subsequent BLASTing of shroom indicates that shroom occurs in almost all multicellular organisms!

### 46.1.2   What does BLAST let you do?

- find homologous genes
- search DNA, protein, mRNA, whole genomes, etc
- search specific curated databases with high-quality data (eg **RefSeq**, **UniProt**)
- determine specific regions of homology via **local alignment**
- Search many many many sequences using efficient **heuristic** search technique
- determine how likely your results are to have occurred by chance (E-value)

## 46.2   BLAST recipe

- Input 1: The query: a sequence of interest (eg mouse shroom)
- Input 2: a database (eg, all known and predicted protein sequences, all reference protein sequences)
- Input 3: a scoring matrix (BLOSUM62, PAM250)
- Output 1: high-scoring local alignments between query and entries in the database
- Output 2: quality scores for each alignment

The BLAST results interface can also run other programs which can build **multiple sequence alignments**, **create simple phylogenetic trees**, link to relevant literature, etc.

## 46.3 Modules in the BLAST Workflow and Algorithm

The writers of BLAST divide it up into 3 modules (Camacho et al 2009). This is also how Pevsner frames it (page 138). The first two steps are by far the most important. The third appears to be mostly about doing book keeping and minor revisions, in part to clean up things that were done in step 1 to maximize speed and reduce memory usage.

1. **Setup**: Converting query sequence to **words**, scoring the words, determining words above neighbor hood threshold
2. **Scanning**: Comparing query word list against database, create initial **ungapped alignment** and subsequent **gapped alignment** for **high-scoring sequence pairs**.
3. **Traceback**: Produce, revise and report full **gapped alignment**

### 46.3.1 Module 1: Setup query sequence (Protein BLAST oriented)

- Read **query sequence**
- Filter **low complexity regions** with little information (eg, AAAAAAAAAA, ATATATATAT, etc)
- Determine all overlapping **words** occurring in the query.
- The default is for 3-letter amino acid words; users can modify this.
- A word may occur more than once; each instance gets processed
- Score query words against **master word list** with **scoring matrix** (**PAM** or **BLOSUM**)
- The master word list for 3-letter words contains 20 x 20 x 20 = 8000 possible words!
- Use a cutoff **T** (default = 11) to determine which words in master words list are in an desirable evolutionary **neighborhood** of the words in the query sequence.
- **T** is sometimes called the **Neighborhood word score threshold** (Mount 2007)
- Typically for each word in a query sequence there are ~50 words with a score >11
- A 100 protein with amino acid will have 98 overlapping words. 98*50 = 4900. These the sequence database will then be scanned for each of the 4900 words!

NOTE 1: *BLAST also does some special data organization and data structures to make this and downstream processing more efficient (for those with a CS background: this involves a look-up table/HASH table). The code I've used to demonstrate parts of the BLAST process are evocative of what goes on, but are not accurate in the data structures and workflow.*

NOTE 2: *The term "master word list" isn't used by anyone, but I think describes*

*what this element is.*

## 46.3.2  Module 2:  Scanning database for high-scoring words

The scanning process in BLAST involves two important steps: find the 1st "hit" between a word from the query (or a neighboring word), then find a 2nd hit. The Baxevanis and Ouellette (2005) description does not indicate this.

The scanning process is done with a **deterministic finite-state automata (DFSA)** (aka a **finite state machine (FSM)**). The only textbook I've found which mentions this and discusses it is Zvelebil; and Baum (2008). A DFSA produces a similar result as a regular expression, though its probably much faster! Most peer review papers except the original BLAST paper (Alschul et al 1990) also don't mention this.

Note: you don't need to know what a DFSA or FSM are. You do need to know that BLST searches sequences for matching words.

### 46.3.2.1  Scanning for the 1st hit

- Scan sequences in database for each word in the **neighborhood** of a word from the query
- Scanning is done using a special computational tool called a **deterministic finite-state automata** (aka a **finite state machine**).
- (**Regular expressions** can be converted to this format; BLAST does not, however, use regular expressions or simple loops to do this search!).
- Each time a word from the neighborhood matches with a sequence in the database BLAST records its position as a **hit**.
- Through the late 1990s, once scanning was done matched words served as the starting point for an ungapped alignment extending in both directions from the match (Some sources such as Baxevanis and Ouellette 2005 still describe BLAST this way);

### 46.3.2.2  Scanning for the 2nd hit

- Since the late 1990s, BLAST has used the **two hit method**
- Once scanning is done, BLAST locates pairs of hits that are within 40 amino acids of each other
- If a hit is isolated with no other hits within 40 amino acids upstream or downstream, its is ignored
- If 2 hits occur within 40 amino acids, BLAST builds and scores an ungapped alignment starting at the second word; overlapping words aren't considered)
- (Some Sources imply that BLAST builds an alignment that runs between these two hits, eg Mount 2007. Pevsner phrasing is also not accurate:

> "BLAST extends hits to find alignments called …Hasps" (pg 140, 2nd paragraph). These are either misstatements or errors).

Note: You don't need to memorize this value of 40; you should be able to reason through what might happen if it is increased or decreased.

### 46.3.2.3 Scoring initial ungapped alignment

Once BLAST finds a second hit, it starts building an **ungapped alignment**. If this alignment exceeds a certain threshold, a gapped alignment (see below). Baxevanis and Ouellette (2005) gloss over this two-step process.

- Starting from the 2nd hit of two adjacent hits, BLAST extends an ungapped alignment in both directions.
- BLAST monitors the score of this un-gapped alignment as it grows.
- Scoring is done with a BLOSUM, PAM or other matrix
- If this ungapped alignment exceeds a certain threshold (**S**), a **gapped alignment** is initiated.
- The value of S is (was?) determined by subjecting **random sequences** to the word matching, scanning, and ungapped alignment procedures just described. S is set so that short alignments occurring due to chance are unlikely to be proceed in the algorithm. (Mount 2007)
- If S is exceeded and a gapped alignment built, the resulting **gapped alignment** is called a **High scoring sequence pair (HSP)**

### 46.3.2.4 Scoring gapped alignment

- As the alignment of a HSP extends, its score is monitored.

- When an alignment has mostly identical or chemically similar matches, its score increases with its length.
- Gaps and matches between dissimilar amino acids cause the alignment score to begin to drop
- Small gaps and mis-matches might result in temporary declines in the alignment score, but the score may increase again after the gap or an area where sequences have diverged.
- If the score drops more than a specified threshold (X.u; usually ~20) then the alignment is stopped
- Once the threshold is exceed BLAST essentially backtracks to where the highest score occurred; this alignment is then stored as a preliminary HSP
- (To save time and memory BLAST uses some shortcuts during the creation and reporting

Note: You don't need to memorize this value of 20; you should be able to reason through what might happen if it is increased or decreased.

### 46.3.3   Module 3: Traceback

- During the traceback phase the alignment is finalized, minor improvements made, and reported
- The traceback procedures appears to be required mostly to deal with simplifications the algorithm used to save time and memory. (For example, only the location and length of preliminary HSPs are stored during the scanning part of the alignment. The full alignment is then built and reported during traceback. I assume this means the full alignment is rebuilt, but I am not yet sure of all the steps)

## 46.4   BLAST alignment statistics

BLAST reports the following things

- The **raw score** from **gapped alignment** between the query and matching sequence
- The proportion of the query sequence which was aligned
- The **percent identity** of the alignment (this involves some rule for how gaps are counted; we won't worry about this)
- The **E-value**
- The **Bit Score** (where?)

Note: BLAST reports both a Max Score and a Total Score, which are both **raw scores**. They are often identical, and we won't worry about exactly what they are.

Tp0751 is a syphilis (Treponema pallidum) outer membrane protein thought to be involved in parthenogenesis. I googled "Tp0751 accession number" and it took me to the UniProt page. I found the FASTA-formatted sequence and pasted it into a blastp search. The default for blastp it so search the "nr" (non-redundant) database. I changed this to RefSeq because this database has higher standard for inclusion.

```
">sp|O83732|Y751_TREPA Uncharacterized protein TP_0751 OS=Treponema pallidum (strain N:
MNRPLLSVAGSLFVAAWALYIFSCFQHGHVPPRRIPPHDTFGALPTAALPSNARDTAAHP
SDTADNTSGSSTTTDPRSHGNAPPAPVGGAAQTHTQPPVQTAMRIALWNRATHGEQGALQ
HLLAGLWIQTEISPNSGDIHPLLFFDREHAEITFSRASVQEIFLVDSAHTHRKTVSFLTR
NTAISSIRRRLEVTFESHEVIHVRAVEDVARLKIGSTSMWDGQYTRYHAGPASAPSP"
```

BLAST returns 27 sequences when I query RefSeq. All occur in *Treponema* species. I also tried the default **non-redundant (nr)** database and got 38 hits, also all in *Treponema* species. The **UniProt** database returns just 1, my original sequence, because no other related proteins have been studied well enough to deserve entry to UniProt.

I also tried PSI-BLAST, which is a modified (and slower) BLAST which is better at finding more distantly related proteins. PSI-BLAST returned 81 sequences.

Some of these are non-Treponema species, but they have E values > 0.05.

The highest scoring **hit** from the RefSeq search is to the sequence I submitted: it as 100% coverage and 100% Identity. This means the alignments between what I submitted and this database sequence are identical.

The second hit is to another species, *T. paraluisicuniculi.* This has 100% query coverage, which means no gaps were introduced. I can confirm this by clicking on the "Alignments" tab and looking at the reported pairwise alignment. Gaps are indicated by dashes, and the top of the alignment has a gap percentage stat.

The percent identity (PID) for this hit is 98.73%. Looking at the actual alignment, I can see only a few "+" symbols which indicate conservative differences between the sequence, and one space, which indicates a non-conservative difference.

The **E-value** is tiny. Its reported at 6e-170, which is $6*10\hat{}-170$. Below 0.05 E values and p-values are almost identical (Pevsner Table 4.3, page 144).

The lowest-scoring hit was from T. maltophilum. Only 55% of my query sequence matched with the T. maltophilum. sequence and the percent identity was 27.27%. Despite this low level of identity, the E value for this hit is still 5e-07, or $5*10\hat{}-7$.

An E value of 5e-07 is going to be very similar to the related p-value, but let's check. Pevsner provides a conversion equation (page 143, equation 4.8): p = 1-e^(-E)

You should memorize this equation (or include it on a note sheet if allowed) and be able to implement it in R as shown below.

In R I can code this as below. Note that instead of e^-E I have to use exp(-E)

```r
E <- 5e-07
p <- 1 - exp(-E)


p
```

```
## [1] 4.999999e-07
```

You can it this way too

```r
p <- 1 - exp(-5e-07 )
p
```

Or skip the assignment step

```r
1 - exp(-5e-07 )
```

Or write it out long-form

```r
1 - exp(-5*10^-7 )
```

# Chapter 47

# BLAST Alignment score distribution

**By**: Nathan Brouwer

```
library(compbio4all)
```

## 47.1   Packages

```
#install.packages("HistData")
library(HistData)
library(ggpubr)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggpubr'
```

```
## The following object is masked from 'package:ape':
##
##     rotate
```

```
## The following objects are masked from 'package:flextable':
##
##     border, font, rotate
```
```
#install.packages("evd")
library(evd)
```

```
##
## Attaching package: 'evd'
```

```
## The following object is masked from 'package:igraph':
##
##     clusters
```

## 47.2   Distributions of alignment scores

Imagine we could keep track of all of the steps of a BLAST search. At a certain point, BLAST will have made many **local alignments** of a few dozen to a couple hundred bases between the **query sequence** you submitted and sequences in the database. Each of these local, pairwise alignments met all the criteria for being worthy of consideration and so are recored as **hits**. Now we want to judge objectively how good these hits are and which ones are most worthy of further consideration. We can do this by working of an **E-value**.

With our collection of hits, we could take their scores and plot them. Many things in biology when plotted will take on a bell-curve or normal distribution. Scores for alignments, however, take on an **extreme value distribution** (EVD). That is, compared to a **symmetric** normal distribution an EVD has too few low numbers (lower alignment scores) and extra high scores.

Now imagine that instead of BLASTing our sequence against a database of real sequence, we made up a bunch of **random sequences**. We could invent a random sequene by writing all the letters representing amino acids, tossing them into a hat, and pulling out an letter and writiing it down. We would then toss the letter back in, mix up the letters, and pull out another one. Repeat this 100 times and we'd have a perfectly random sequence the would be about the length of a protein, but wouldn't code for anything (or be very very very very very unlikely to code for anything). If we invent, say 1000 sequence we could make a database of random sequences and would could aline our focal query sequence against our made up database. When could then score the local alignment between the query and each random sequence. Because these sequences are made up, the alignments will be pretty bad and teh scores low. However, occassionally there might be a fairly high score due to chance. So, just due to luck we could accidently invent a sequence that is a pretty good alignment with our query sequence. The question the **E-values** are trying to get at is how often this could happen, and how much better are our the scoures of our BLAST hits than the scores we'd get from BLASTing again random sequences.

The following code will illustrate these concepts.

## 47.3   Simulating data

The easiest way to think about simulating data is to roll a dice. Six-sided die allows you to draw numbers from a **uniform distribution**. With a uniform

distribution, all values have the same likelihood of occuring. On a die, you have a 1/6 change of getting a 1, a 1/6 change of a 2, etc.

We can simulate a 6-sided die in R using the runif() function, which doesn't mean "run-if" but rather "r-unif", for random-uniform.

We can simulate rolling a die once with this code. n is the number of rolls. Min is the minimum possible value, max is the max possible value. Note that the min is set to 0.5 and the max to 6. This will be explained in a second.

So, this code generates a number between 0.5 and 6.5

```
runif(n = 1, min = 0.5, max = 6.5)
```

```
## [1] 6.059917
```

This code generates 10 numbers from 0.5 to 6.5.

```
runif(n = 10, min = 0.5, max = 6.5)
```

```
##  [1] 2.923826 1.250543 2.165907 1.409444 3.973238 6.211611 5.283448 4.050456
##  [9] 3.678757 2.334643
```

runif() allows for non-integers, so we can make this like a real dice using round()

```
round(runif(n = 1, min = 0.50000000, max = 6.50000000))
```

```
## [1] 5
```

If we want to simulate what would happen if 50 people in class all rolled a dice we set n to 50

```
round(runif(n = 50, min = 0.5, max = 6.5))
```

```
##  [1] 2 3 3 4 4 3 3 4 1 5 3 2 6 2 2 6 4 2 3 1 6 6 4 2 5 2 3 1 3 1 3 1 3 6 6 3 5 3
## [39] 2 4 3 3 3 3 4 2 2 2 4 2
```

If we had 10000 people in our class:

```
round(runif(n = 10000, min = 0.5, max = 6.5))
```

```
##     [1] 2 4 3 1 1 4 4 4 2 3 3 3 6 6 6 4 4 6 3 1 2 4 2 2 5 5 4 6 6 4 2 6 3 5 6 6
##    [37] 1 5 1 2 1 2 3 6 6 4 4 5 3 5 6 5 4 4 3 6 6 1 4 5 1 1 1 2 3 5 2 6 2 4 6 1
##    [73] 4 6 2 4 5 2 6 1 5 4 5 3 6 5 4 4 1 4 4 5 6 2 2 1 6 4 5 4 1 6 5 3 1 2 1 3
##   [109] 3 6 1 1 5 6 1 3 4 4 2 3 5 3 3 6 3 4 4 3 4 6 3 4 3 2 2 4 2 3 6 5 4 3 4 4
##   [145] 4 6 1 6 1 1 2 3 6 5 6 5 6 2 2 5 5 6 2 1 6 4 4 5 4 2 4 3 2 1 6 2 3 6 3 6
##   [181] 1 3 1 6 2 1 5 6 1 5 2 2 3 6 5 5 6 1 6 3 5 1 6 6 4 3 4 5 2 4 6 6 6 5 1 2
##   [217] 5 3 5 4 3 3 3 6 6 5 6 1 4 2 4 2 4 2 6 4 1 1 6 3 3 5 6 3 5 5 4 5 4 3 2 6
##   [253] 2 4 4 4 4 1 5 6 1 3 2 2 4 1 1 5 6 2 5 6 6 2 5 4 6 2 1 1 5 3 1 2 2 1 5 6
##   [289] 5 6 1 1 5 1 1 6 2 2 5 4 5 3 4 2 5 4 3 6 2 2 1 4 2 4 2 5 6 6 4 1 2 2 6 1
##   [325] 2 2 5 3 1 5 6 5 5 6 5 3 3 4 6 4 4 6 2 2 1 2 2 5 6 5 1 3 3 1 3 4 4 6 5 4
##   [361] 3 5 5 3 4 3 5 1 3 1 4 4 4 4 1 6 2 6 5 2 1 4 3 2 5 5 6 2 3 1 4 4 6 4 6 2
##   [397] 1 5 2 4 6 3 6 6 4 1 4 5 4 1 4 6 3 2 5 3 6 5 1 3 3 2 3 3 6 6 2 2 3 1 5 1
```

```
##   [433] 2 6 6 4 2 5 3 6 4 3 6 4 1 4 1 2 2 6 3 3 6 2 2 4 2 2 1 3 5 6 3 6 6 2 6 3
##   [469] 3 6 3 1 3 4 2 5 3 4 4 2 3 3 1 3 2 3 5 6 4 3 5 1 5 3 6 4 5 6 1 2 4 1 3 5
##   [505] 2 4 5 3 1 6 5 6 4 6 5 1 4 4 4 4 1 3 2 2 1 1 1 2 5 2 5 3 4 1 4 3 3 3 4 5
##   [541] 6 5 2 6 2 1 1 5 6 2 6 6 3 6 2 5 2 2 4 4 5 6 2 5 5 4 6 4 6 5 5 2 2 1 2 2
##   [577] 2 4 3 5 1 2 6 6 4 6 4 2 4 4 3 5 1 4 5 2 4 3 2 5 5 2 6 6 3 6 5 6 2 3 4 1
##   [613] 5 4 4 2 3 3 1 2 1 2 4 1 2 2 3 5 5 2 4 1 1 6 2 6 1 2 4 5 4 2 6 3 2 2 3 1
##   [649] 6 1 6 6 4 4 3 3 6 1 1 5 3 2 1 5 2 2 2 2 2 2 1 5 3 1 6 6 1 4 5 4 3 1 3
##   [685] 6 3 3 3 4 6 5 5 6 2 2 1 6 4 4 1 5 1 3 1 3 5 2 1 3 5 2 4 6 3 4 2 5 6 5 5
##   [721] 3 3 5 6 6 4 5 6 2 6 4 6 3 5 5 5 5 4 6 6 2 2 4 5 1 3 6 1 3 2 2 5 2 6 2 3
##   [757] 4 5 3 5 6 1 5 1 3 1 5 5 2 2 1 6 4 5 6 1 3 2 3 3 2 1 1 2 2 1 1 2 5 2 6 2
##   [793] 6 5 5 1 3 3 2 2 1 3 6 5 3 2 1 3 1 5 2 4 4 2 5 4 2 3 5 5 2 3 5 2 1 4 2 6
##   [829] 5 3 4 1 5 3 1 3 1 4 4 1 5 2 4 5 4 3 6 1 6 2 5 5 2 6 6 2 4 3 1 1 3 5 5 2
##   [865] 1 6 5 1 1 3 5 4 1 2 4 5 1 4 5 4 4 6 5 4 1 5 6 6 3 6 1 6 1 1 4 6 6 3 6 5
##   [901] 3 2 4 2 5 2 1 2 1 3 4 2 3 5 2 2 5 1 5 6 2 5 3 3 5 1 2 4 5 1 3 4 3 1 3 6
##   [937] 3 2 5 4 4 5 4 1 5 2 1 1 5 5 4 4 1 3 4 3 2 5 1 6 2 5 5 5 1 6 5 3 2 3 5 2
##   [973] 5 2 4 6 4 2 5 5 5 6 5 2 3 4 2 5 2 6 4 6 5 1 6 4 2 2 3 4 3 3 5 2 6 5 6 2
##  [1009] 5 3 1 5 5 6 5 2 5 4 1 3 1 5 6 5 5 3 2 5 3 5 6 2 3 3 6 6 3 1 3 6 3 6 4 4
##  [1045] 5 6 6 1 6 1 2 1 3 2 1 2 4 4 2 3 6 5 1 2 2 5 1 6 1 2 3 3 6 4 1 1 2 6 3 6
##  [1081] 5 6 2 6 6 2 3 3 1 3 2 1 1 2 6 6 6 5 4 1 5 3 2 1 5 2 5 3 4 4 5 1 2 4 3 4
##  [1117] 1 1 1 5 5 4 4 5 5 1 3 1 2 1 2 5 2 3 2 2 4 3 1 4 6 2 1 2 3 5 3 2 3 6 2 4
##  [1153] 1 1 4 1 4 6 5 1 1 2 3 2 1 6 6 3 3 3 2 4 1 3 1 4 2 4 6 1 2 5 6 2 3 6 5 5
##  [1189] 3 1 1 6 4 1 5 3 1 3 2 5 5 3 1 1 1 6 4 3 5 1 3 1 2 6 4 4 5 6 4 5 2 2 6 1
##  [1225] 4 1 4 2 3 4 3 5 4 2 1 3 2 1 5 6 3 3 6 6 3 4 2 2 6 4 3 1 2 5 3 5 4 1 5 4
##  [1261] 5 6 5 5 4 1 2 4 5 6 6 2 1 6 2 1 3 3 4 5 4 2 1 4 3 5 3 3 6 5 4 4 1 5 2 4
##  [1297] 3 6 3 4 1 3 6 3 5 1 2 1 6 3 1 3 3 5 6 1 5 1 2 2 6 5 4 2 3 2 2 3 5 4 6 2
##  [1333] 6 6 1 2 3 2 3 1 2 6 5 6 5 3 2 3 1 1 2 4 4 1 3 6 6 1 4 5 4 6 6 4 2 5 6 4
##  [1369] 1 5 2 3 2 1 3 4 4 3 3 3 2 5 2 1 3 4 4 6 2 1 2 2 3 1 6 1 4 1 4 4 5 5 2 3
##  [1405] 1 3 1 5 1 3 2 2 2 6 5 5 5 5 2 3 2 2 2 3 2 3 4 2 4 2 5 2 5 2 1 2 5 6 3 5
##  [1441] 3 6 6 5 3 5 6 4 2 2 4 2 5 1 6 1 5 1 6 4 6 5 3 6 6 1 1 3 3 5 3 3 3 3 3 6
##  [1477] 6 1 2 4 4 6 2 2 1 5 2 3 4 1 5 5 3 6 4 3 1 3 1 6 4 2 1 3 6 2 4 3 2 6 4 5
##  [1513] 4 3 3 2 2 2 4 1 4 3 6 2 1 6 5 6 4 4 4 5 2 3 3 4 1 2 5 6 5 1 6 2 1 6 6 5
##  [1549] 4 6 5 4 3 3 4 3 3 2 1 6 5 2 5 4 5 5 3 6 4 4 5 6 3 3 1 2 3 6 4 3 4 3 3 4
##  [1585] 1 4 5 5 3 6 3 5 1 1 4 6 1 1 5 3 3 2 6 2 6 2 1 2 5 3 4 4 2 3 5 2 1 5 1 1
##  [1621] 3 5 5 2 4 2 1 6 2 6 6 4 3 3 4 1 4 3 2 6 6 1 3 1 4 4 4 2 3 3 4 3 1 4 6 2
##  [1657] 6 1 1 4 6 6 3 2 5 1 3 2 1 3 6 1 3 4 4 4 3 2 5 4 1 3 3 3 6 6 2 6 2 4 1 6
##  [1693] 6 1 1 5 6 2 5 1 1 2 4 3 2 2 5 3 5 3 1 3 4 4 6 1 4 6 3 4 4 4 3 6 4 4 2 1
##  [1729] 4 3 4 6 6 1 1 6 6 3 3 6 6 2 2 1 4 6 5 1 6 2 6 6 4 6 2 2 5 3 3 3 6 1 4 2
##  [1765] 3 4 6 4 4 5 3 3 1 3 5 5 2 1 4 6 5 6 2 6 4 5 6 1 4 3 3 4 6 1 2 3 3 5 6 2
##  [1801] 3 4 6 5 6 3 4 6 4 2 2 2 4 4 5 2 5 2 3 5 5 2 6 4 6 4 5 2 6 3 6 3 5 1 5 2
##  [1837] 2 3 2 3 6 1 2 3 1 3 6 5 2 1 3 2 5 6 2 2 2 1 5 2 2 1 2 4 4 3 3 4 1 5 6 6
##  [1873] 4 1 1 5 6 1 5 3 2 1 4 6 4 1 4 4 6 2 1 2 5 6 2 3 3 5 2 1 1 5 2 5 3 6 3 5
##  [1909] 5 1 1 4 1 1 4 6 5 1 1 3 1 6 3 1 1 1 6 2 6 3 3 4 5 2 2 6 4 2 3 3 2 5 5 6
##  [1945] 5 4 5 3 3 1 1 6 1 3 6 2 6 6 3 3 1 1 6 6 3 3 1 4 3 2 5 2 6 5 1 2 6 1 6 4
##  [1981] 2 2 4 3 5 5 4 6 6 6 4 3 1 3 4 5 3 5 5 6 2 4 1 1 1 3 1 4 3 2 1 1 2 3 6 3
##  [2017] 1 1 6 3 5 2 5 1 4 1 6 1 1 6 2 2 2 1 3 6 6 1 1 5 1 4 4 5 6 1 5 2 3 2 5 4
##  [2053] 1 3 4 1 3 4 1 1 2 4 4 3 3 3 2 4 2 2 6 6 1 2 1 5 2 2 1 5 4 2 5 2 4 5 4 4
```

```
## [2089] 5 3 6 3 4 3 3 3 3 1 6 3 6 1 2 2 3 6 4 3 5 1 2 6 1 3 5 5 6 2 3 1 2 2 3 3
## [2125] 1 1 2 3 4 2 1 2 1 4 5 3 3 3 4 5 6 5 3 1 6 5 2 6 6 2 3 1 1 4 4 3 3 6 6 2
## [2161] 6 1 1 6 4 6 6 1 1 2 1 3 3 4 1 5 1 3 6 5 2 1 2 5 1 6 3 5 5 5 2 2 6 2 2 6
## [2197] 4 5 4 1 5 5 5 4 3 4 2 2 1 5 5 6 2 6 4 2 6 3 5 1 2 3 6 4 4 3 6 6 2 3 6 3
## [2233] 6 2 4 2 5 3 3 2 2 2 6 1 1 6 3 6 6 3 6 6 5 5 4 3 2 6 3 4 1 2 5 1 5 3 3 1
## [2269] 2 2 5 6 5 5 5 5 4 2 5 3 3 3 5 4 6 6 5 5 5 4 5 2 2 5 3 6 6 5 4 3 3 1 1 4
## [2305] 4 5 5 3 6 2 5 2 2 6 4 4 5 6 1 2 5 3 6 2 3 5 2 1 1 4 6 2 4 4 2 6 1 6 1 3
## [2341] 5 1 5 3 2 6 2 5 6 2 6 3 4 2 3 3 4 3 2 1 6 5 1 3 5 2 1 1 2 3 3 3 5 1 1 2
## [2377] 5 2 3 3 5 6 3 6 6 2 4 4 2 4 1 3 3 3 1 1 5 1 5 4 2 2 1 4 5 5 2 5 2 3 4 2
## [2413] 5 1 2 6 2 3 1 1 2 1 4 3 3 2 2 4 4 6 5 5 2 4 6 5 5 1 5 2 5 1 2 2 4 1 1 1
## [2449] 5 1 4 2 2 3 4 3 3 6 2 3 4 6 6 5 6 4 1 2 4 3 2 6 2 6 4 6 2 2 3 6 2 1 5 6
## [2485] 4 1 1 2 4 5 6 6 3 3 2 6 6 5 2 5 2 5 3 2 5 1 2 1 4 2 2 6 6 1 6 4 2 2 5 6
## [2521] 3 4 5 2 5 4 1 3 3 2 2 6 3 1 5 6 4 3 6 1 2 6 2 3 3 4 2 3 6 2 3 3 2 3 5 6
## [2557] 4 5 5 6 5 2 1 4 2 3 6 2 5 5 3 3 4 4 2 2 4 4 6 1 4 4 1 2 2 2 2 5 1 6 4 4
## [2593] 5 1 1 2 4 5 2 5 6 5 6 2 5 6 3 6 6 1 3 1 1 2 5 3 4 4 4 1 1 6 2 2 3 6 1 2
## [2629] 3 5 5 1 6 6 4 3 6 6 5 6 2 6 4 2 1 2 4 1 1 1 1 2 5 1 4 5 3 3 4 2 2 6 4 3
## [2665] 2 6 6 3 4 1 3 6 1 1 2 1 5 6 6 2 1 4 4 1 1 4 5 2 6 4 2 6 2 4 1 3 3 3 4 4
## [2701] 5 5 3 3 4 3 4 6 4 4 3 3 4 5 2 4 5 5 1 3 3 5 5 3 2 3 5 5 2 3 5 1 5 5 6 5
## [2737] 3 1 4 1 5 1 3 6 1 1 1 3 2 6 3 1 4 5 2 5 4 5 3 1 2 6 3 2 3 1 1 2 1 1 2 1
## [2773] 1 5 2 5 2 3 6 3 5 4 1 1 6 5 3 3 6 3 1 2 2 4 4 1 4 4 2 6 1 6 6 2 5 1 1 3
## [2809] 3 2 2 6 3 3 5 2 6 2 4 3 2 4 4 1 1 1 4 6 3 5 6 3 4 6 3 5 5 3 6 1 3 5 5 4
## [2845] 5 2 3 6 6 1 4 4 1 1 6 1 3 3 1 1 6 1 5 5 1 6 1 3 5 6 4 2 6 6 6 5 3 6 2 2
## [2881] 3 6 1 6 6 4 5 3 3 5 5 2 6 3 2 2 5 1 5 2 4 2 5 4 4 2 6 6 2 1 6 2 3 3 3 1
## [2917] 4 5 1 4 6 2 1 5 3 3 5 6 3 4 1 2 5 4 6 1 6 1 3 1 2 4 1 3 6 6 2 2 2 5 4 6
## [2953] 2 5 3 5 2 1 1 2 4 1 5 5 4 1 1 2 5 1 1 1 5 3 3 5 6 5 3 1 3 1 6 4 3 1 5 4
## [2989] 5 2 2 3 4 1 4 5 5 6 3 2 1 4 6 2 6 4 4 6 5 1 5 1 1 1 5 1 3 1 1 3 5 5 4 2
## [3025] 1 6 5 3 1 4 1 6 6 4 2 2 5 4 1 5 6 3 4 4 3 3 4 1 5 2 5 6 5 3 1 2 1 3 1 6
## [3061] 5 4 4 4 6 2 5 1 4 1 3 2 3 3 2 2 5 3 1 6 5 5 2 1 2 6 3 4 6 5 2 3 6 3 5 5
## [3097] 4 3 2 1 6 3 1 4 2 4 3 6 5 5 6 1 4 6 3 3 4 5 6 5 2 3 6 2 4 1 3 1 6 5 2 2
## [3133] 4 6 1 2 5 5 3 6 2 4 1 1 2 3 5 6 6 5 5 2 5 2 3 3 6 5 4 4 1 1 2 6 5 6 3 2
## [3169] 4 4 2 3 6 6 5 2 1 2 3 5 3 5 5 3 1 2 3 6 3 2 2 6 4 6 1 3 2 4 1 1 5 5 1 2
## [3205] 1 2 5 4 5 2 5 3 3 1 4 1 3 4 5 4 6 6 5 6 5 1 1 2 2 5 2 6 1 4 3 1 1 2 3 5
## [3241] 5 5 5 6 5 3 5 1 2 6 3 6 1 5 5 6 4 5 3 1 1 6 5 3 6 1 4 3 2 5 6 3 2 5 3 5
## [3277] 1 2 1 2 2 2 3 4 5 2 2 6 2 3 2 1 4 3 4 2 4 5 4 1 1 4 1 6 2 6 2 5 6 4 5 6
## [3313] 1 2 2 6 6 1 2 5 2 4 6 3 2 5 5 4 2 6 6 4 2 4 6 1 3 5 2 6 1 3 3 5 2 1 2 6
## [3349] 1 2 4 1 5 5 5 2 6 4 5 1 5 5 1 2 5 3 2 6 1 4 5 5 2 1 3 5 5 2 2 4 4 2 4 4
## [3385] 4 4 4 4 2 1 2 3 6 3 5 3 6 4 2 3 4 1 1 2 6 4 4 5 3 3 4 5 1 1 2 3 2 4 1 3
## [3421] 2 1 5 4 5 2 5 3 5 6 5 3 6 3 1 2 3 6 3 1 5 2 1 4 5 3 2 2 5 6 2 4 4 2 1 5
## [3457] 3 6 1 2 1 5 6 1 6 6 1 2 1 2 1 3 2 1 3 1 6 6 1 5 3 5 5 6 4 1 3 6 1 6 4 2
## [3493] 5 4 1 6 4 6 5 3 4 4 4 4 6 6 3 4 1 3 4 5 4 5 2 6 5 2 1 5 1 6 1 2 1 3 6
## [3529] 1 6 5 2 6 1 3 1 4 3 4 4 5 3 3 4 5 6 1 5 6 5 5 5 1 3 2 6 2 3 1 5 1 2 4 3
## [3565] 6 5 1 6 3 4 6 6 2 6 4 2 4 3 6 4 1 6 5 5 2 5 1 1 3 4 3 3 6 4 1 4 3 3 1 5
## [3601] 1 3 5 5 6 1 2 2 5 5 3 6 1 1 4 2 6 2 2 2 4 3 1 4 4 5 4 1 3 2 5 3 5 3 3 1
## [3637] 2 4 1 4 5 4 5 4 6 5 1 3 5 6 4 3 1 5 1 5 4 2 6 3 2 1 4 1 6 3 5 4 3 6 1 6
## [3673] 1 6 5 2 3 2 5 1 6 3 4 2 6 4 1 5 6 3 5 1 4 1 5 5 3 1 6 4 2 4 3 1 5 4 3 6
## [3709] 5 2 4 6 4 6 5 5 6 2 5 1 1 2 5 5 1 4 1 1 1 1 5 4 1 6 3 1 3 2 4 4 5 6 5 4
```

```
## [3745] 2 4 2 3 6 5 2 6 4 1 1 1 1 1 4 5 3 3 2 2 3 6 4 2 1 3 1 5 4 3 1 3 5 4 6 1
## [3781] 4 5 5 2 3 1 1 4 1 4 2 6 1 6 3 3 1 1 1 1 4 1 5 6 4 4 2 2 3 2 3 6 1 2 3 3
## [3817] 6 3 1 1 2 2 4 5 6 4 4 1 6 6 6 1 6 1 1 5 5 1 2 6 5 3 5 5 5 2 4 4 6 4 3 3
## [3853] 4 3 2 1 4 5 4 1 3 2 2 1 3 5 4 2 5 6 2 5 3 1 5 5 6 6 3 3 2 5 6 4 5 3 2 6
## [3889] 1 4 4 6 2 5 4 4 2 6 5 6 3 1 1 4 5 4 4 3 3 5 4 2 3 2 3 3 4 4 6 6 5 5 4 2
## [3925] 5 4 4 2 5 1 4 4 4 3 6 3 4 6 6 2 1 6 3 4 4 3 6 4 2 2 4 3 6 5 4 4 5 4 6 3
## [3961] 3 5 6 5 2 3 6 2 2 5 4 3 5 6 1 6 6 1 5 5 5 2 1 4 4 2 5 3 2 6 5 6 6 1 6 4
## [3997] 2 6 5 6 5 6 6 5 6 5 5 6 3 3 6 2 3 4 2 5 4 1 4 4 1 1 4 2 5 6 5 4 5 3 1 3
## [4033] 2 1 4 4 4 1 1 2 4 4 4 5 6 1 5 1 3 4 5 6 4 5 3 1 6 4 3 3 1 2 5 6 6 3 3 5
## [4069] 2 3 5 5 6 3 4 2 2 6 6 3 5 6 1 5 3 5 3 4 3 2 4 2 4 6 4 5 3 1 6 3 2 3 5 1
## [4105] 3 3 6 5 2 3 4 5 6 5 1 3 2 5 3 4 4 5 4 6 6 5 5 5 5 5 6 4 5 4 1 3 1 4 4 3
## [4141] 1 4 4 2 2 5 6 1 3 3 1 4 4 2 6 6 5 1 1 2 6 1 4 6 5 3 5 5 3 4 5 3 6 3 4 1
## [4177] 3 1 5 3 4 2 3 6 5 2 4 2 3 1 6 4 6 6 6 4 3 6 5 6 5 1 4 6 2 6 4 5 5 3 4 1
## [4213] 2 1 1 3 3 5 1 2 5 6 3 5 4 3 3 2 4 3 6 5 5 2 1 5 3 2 1 5 5 2 2 6 5 6 3 2
## [4249] 4 5 2 4 1 1 4 5 4 3 6 2 1 6 1 5 4 5 2 5 6 3 5 2 1 6 4 1 6 3 6 6 5 6 4 2
## [4285] 2 1 5 1 6 4 6 5 1 4 1 6 4 4 2 4 2 3 3 1 4 1 4 5 2 4 2 6 1 4 4 2 3 6 4 2
## [4321] 5 3 6 2 5 2 3 3 2 3 5 1 1 5 1 4 5 2 2 4 4 5 5 3 4 1 1 3 6 2 3 3 6 1 1 5
## [4357] 2 3 2 4 3 5 4 2 5 5 2 5 5 4 5 3 5 2 1 2 4 5 3 2 4 5 3 1 5 1 5 6 5 6 3 1
## [4393] 4 3 5 1 1 1 6 2 6 4 2 2 2 4 2 6 4 3 6 6 4 6 2 3 2 3 5 5 3 4 5 6 2 6 6 6
## [4429] 1 2 2 4 4 1 3 6 3 3 3 6 2 1 6 3 1 1 1 2 2 6 4 3 5 2 1 5 3 6 5 3 1 4 3 2
## [4465] 1 4 6 3 2 5 3 6 3 4 5 1 6 1 6 6 5 5 4 6 3 1 5 1 5 3 3 3 4 4 4 5 1 2 4 6
## [4501] 6 1 5 5 2 6 3 3 3 1 1 3 2 1 4 5 6 2 5 2 5 1 2 5 4 2 6 6 4 1 1 1 5 3 5 4
## [4537] 4 6 5 5 1 2 3 2 6 5 2 2 5 1 5 1 6 2 3 6 4 2 5 3 5 1 4 5 4 5 4 2 5 1 1 1
## [4573] 3 4 5 4 1 3 5 4 3 6 3 2 1 6 4 6 5 3 4 2 1 1 4 4 6 4 6 6 2 2 3 4 3 2 3 5
## [4609] 6 1 5 2 6 4 2 6 5 3 4 1 3 6 6 3 4 3 5 5 4 2 1 4 3 1 5 5 5 6 5 2 3 6 6 1
## [4645] 3 4 2 4 2 6 1 5 4 4 6 5 6 6 2 3 1 4 2 1 6 6 5 5 4 6 5 1 4 6 5 3 5 2 5 6
## [4681] 4 3 4 1 2 3 3 5 2 3 1 5 5 4 1 2 4 6 6 4 2 3 3 2 4 5 1 5 5 4 5 4 3 5 5 6
## [4717] 4 6 1 4 6 4 1 2 2 5 6 2 1 3 4 3 5 4 6 4 4 1 1 2 4 2 5 3 2 6 2 4 5 5 3 5
## [4753] 3 6 2 4 4 5 5 3 2 2 5 6 4 3 6 2 1 6 5 6 4 3 3 4 5 1 1 4 3 1 3 3 3 5 3 6
## [4789] 2 6 6 1 5 1 4 3 1 4 6 6 3 2 2 2 5 6 5 4 2 6 3 3 5 2 5 4 1 2 2 6 3 3 2 4
## [4825] 1 2 1 1 1 4 6 5 4 6 6 6 6 3 2 2 3 1 6 1 2 6 3 6 3 2 4 1 4 4 5 3 3 3 2 5
## [4861] 5 3 2 5 5 6 5 4 6 1 6 4 5 3 1 5 2 5 4 4 5 2 5 2 5 6 1 4 4 1 1 3 6 5 1 5
## [4897] 2 6 3 5 6 5 2 3 1 6 3 6 5 6 5 1 1 2 1 2 1 2 5 2 4 5 1 3 6 6 5 5 6 4 6 5
## [4933] 5 5 2 5 1 1 5 4 1 4 5 4 6 3 3 4 4 3 1 3 5 6 2 4 3 1 2 5 3 4 3 4 1 3 3 5
## [4969] 6 2 3 1 6 1 2 3 3 3 6 2 3 3 5 3 3 4 4 5 6 5 1 6 2 6 3 1 1 3 1 2 3 2 6 6
## [5005] 1 4 3 4 1 1 1 1 4 2 2 1 1 5 2 6 5 4 1 6 5 3 6 3 6 6 4 3 1 5 4 3 4 2 5 6
## [5041] 3 2 1 3 3 5 6 6 4 6 4 4 2 6 1 5 3 3 6 3 2 2 2 1 4 6 5 4 3 5 6 4 4 4 2 3
## [5077] 3 6 3 5 6 5 4 4 4 2 6 4 3 2 1 2 1 5 5 5 3 3 2 6 2 3 3 5 6 4 2 6 1 1 2 2
## [5113] 6 4 2 5 2 2 5 1 2 2 6 4 4 4 4 6 1 1 5 6 5 4 5 2 6 4 5 2 1 3 1 6 5 2 4 5
## [5149] 1 2 3 4 1 2 5 2 2 2 6 6 5 3 3 2 3 5 4 3 1 1 1 5 2 1 5 1 6 1 4 5 1 6 5 6
## [5185] 1 1 6 6 5 3 6 5 1 6 4 5 2 5 4 4 1 2 6 5 1 1 6 1 3 3 4 2 1 1 3 2 5 6 4 1
## [5221] 6 1 1 1 2 4 2 5 4 1 1 2 4 1 1 1 1 5 3 1 6 4 5 1 2 3 5 3 3 3 3 2 4 4 5 3
## [5257] 6 3 6 3 6 4 4 6 5 6 6 1 1 1 5 5 1 2 6 6 3 6 4 5 6 6 5 6 2 3 1 5 1 4 3 4
## [5293] 3 6 2 4 3 1 5 3 6 1 2 4 4 4 5 3 6 6 4 2 1 2 6 2 5 4 6 5 1 3 5 5 3 6 5 6
## [5329] 4 4 6 3 5 1 1 2 6 2 6 1 5 4 4 3 2 1 6 6 6 1 1 4 3 3 1 3 1 6 1 6 4 4 4 1
## [5365] 6 5 2 1 6 1 4 5 4 6 5 3 5 4 2 2 2 3 5 6 5 2 3 3 6 3 6 6 2 3 6 2 2 5 6 2
```

```
## [5401] 2 1 4 1 3 6 1 3 1 5 6 5 2 4 2 5 3 3 4 3 5 3 3 6 6 4 1 6 1 6 4 6 1 2 2 4
## [5437] 2 1 2 2 2 3 4 3 6 1 3 3 4 3 6 3 5 1 6 4 6 1 1 3 5 4 6 4 3 2 1 3 1 2 5 2
## [5473] 2 2 2 4 2 6 6 2 4 5 3 2 6 4 3 1 1 5 2 5 1 3 6 3 1 6 2 1 5 1 1 5 1 6 1 1
## [5509] 5 3 3 5 6 1 6 6 2 1 4 6 6 6 4 6 4 6 2 3 1 3 5 3 5 6 2 6 4 2 5 5 3 3 3 4
## [5545] 1 2 5 5 4 3 2 4 2 5 6 5 1 6 2 2 1 1 2 6 3 6 2 4 4 5 4 6 6 5 5 4 3 5 1 4
## [5581] 3 2 1 6 4 1 2 4 1 2 6 1 2 5 2 2 5 2 4 4 2 2 2 1 4 2 6 4 1 5 4 3 1 6 3 4
## [5617] 3 2 1 3 4 6 4 1 2 2 6 2 1 6 6 6 1 6 6 6 3 6 4 3 6 3 5 5 6 1 2 3 2 3 1 3
## [5653] 3 2 1 6 1 6 1 4 3 2 6 6 4 4 6 2 2 5 2 1 1 2 2 2 6 1 6 5 4 4 1 4 4 5 1 5
## [5689] 2 4 2 6 6 3 6 3 1 6 6 4 1 3 3 6 2 6 6 1 1 1 4 6 1 6 3 6 3 5 3 5 3 6 2 4
## [5725] 4 3 6 2 4 3 6 3 3 6 6 3 2 4 2 6 4 2 5 3 6 5 3 2 3 6 5 1 4 3 1 4 4 1 2 1
## [5761] 6 1 5 1 4 1 6 4 5 4 3 1 2 4 6 4 3 5 2 3 5 3 6 6 1 4 1 3 2 3 2 2 6 6 6 4
## [5797] 6 1 3 4 6 1 3 5 3 5 1 6 4 2 1 5 2 2 6 3 6 6 4 2 2 4 5 1 5 4 2 2 6 5 4 5
## [5833] 2 4 6 3 3 4 4 5 4 4 1 6 6 6 2 5 3 6 2 5 6 2 3 3 2 2 4 5 4 2 2 6 1 1 3 2
## [5869] 3 3 5 3 4 2 5 4 5 4 2 1 2 2 3 1 5 5 6 3 5 6 6 3 6 3 3 2 4 4 1 1 1 5 2 4
## [5905] 5 6 3 3 6 2 6 2 5 3 4 2 5 3 6 3 6 5 5 3 6 3 3 6 6 3 6 4 1 5 5 3 6 6 6 5
## [5941] 3 6 1 1 2 5 4 5 6 5 2 6 5 6 6 4 6 1 3 5 5 3 1 4 5 1 6 4 3 3 5 1 4 5 6 1
## [5977] 5 4 6 1 4 2 6 4 6 1 3 3 4 2 3 6 2 1 2 5 3 1 6 2 3 6 2 1 5 1 4 6 5 3 3 3
## [6013] 2 3 6 3 1 6 5 4 3 6 2 6 5 5 1 6 1 1 5 2 6 2 6 3 6 6 2 6 6 1 3 1 2 5 4 5
## [6049] 4 3 2 4 4 5 4 5 2 1 2 6 6 5 5 6 2 1 5 1 6 2 6 2 3 5 1 2 5 1 1 4 5 3 3 6
## [6085] 1 4 3 1 3 3 5 4 4 1 4 6 6 6 3 1 5 1 2 3 5 2 6 2 2 3 5 1 3 4 3 4 2 1 5 5
## [6121] 1 3 5 6 5 1 5 5 6 2 4 6 3 2 2 1 6 3 4 5 6 2 2 2 2 4 5 2 6 1 2 1 6 6 4
## [6157] 4 1 4 4 3 6 5 3 6 3 2 6 6 3 1 4 1 5 2 6 1 2 3 3 2 2 3 5 6 4 2 5 3 4 4 1
## [6193] 4 2 2 2 6 1 4 4 6 6 4 3 4 4 1 3 2 5 2 6 1 5 1 1 6 3 1 3 4 4 4 2 1 5 5 3
## [6229] 2 6 4 2 6 3 2 2 6 1 3 5 6 2 4 1 1 1 5 4 5 6 4 2 3 5 6 1 2 6 4 2 5 3 6 1
## [6265] 2 3 4 1 2 2 3 5 1 4 2 4 4 4 5 6 6 2 2 6 2 6 3 2 4 5 3 3 5 1 1 4 2 4 5 6
## [6301] 4 3 2 4 5 1 4 2 3 5 3 1 3 6 5 4 3 3 4 6 2 1 4 4 6 2 6 3 2 1 3 4 4 2 5 3
## [6337] 1 6 1 6 4 1 2 3 4 2 3 4 6 1 5 3 6 5 6 2 5 1 5 5 1 6 5 2 2 5 3 4 6 6 1 4
## [6373] 1 1 4 3 2 1 1 1 6 1 1 4 2 5 2 1 1 6 4 1 1 5 4 4 4 1 4 5 4 4 4 5 3 1 4 2
## [6409] 6 4 2 3 5 3 3 6 2 6 4 3 2 1 2 6 5 6 5 1 5 1 6 5 1 3 5 6 1 5 1 5 5 5 5 5
## [6445] 2 3 6 5 2 4 3 4 3 5 3 5 6 1 2 1 5 5 2 3 3 1 6 6 5 6 4 4 1 5 1 4 2 5 2 5
## [6481] 1 4 6 2 6 1 5 3 1 2 2 5 1 4 1 2 3 4 6 5 2 4 1 3 6 2 6 5 4 4 5 6 1 1 4 3
## [6517] 2 4 1 5 5 6 6 5 2 3 6 2 5 1 2 6 4 4 3 2 2 4 6 6 1 6 3 1 3 6 2 3 3 6 5 4
## [6553] 5 1 4 5 4 1 2 1 3 3 3 5 1 6 4 6 5 3 2 4 1 6 6 3 6 3 5 5 2 4 4 1 1 2 5 1
## [6589] 2 5 3 5 3 6 6 3 5 3 3 1 2 4 1 5 3 5 5 2 4 3 5 6 2 2 3 2 3 4 2 6 4 5 3 6
## [6625] 4 6 1 5 6 4 5 1 4 4 1 1 2 5 5 2 4 5 2 5 5 2 4 4 5 4 4 2 5 2 1 1 3 1 5 1
## [6661] 1 2 1 4 5 2 4 2 2 4 3 1 3 5 2 6 5 6 3 4 2 5 6 5 1 6 6 3 4 6 1 6 5 1 1 2
## [6697] 1 3 6 1 4 5 6 2 3 3 6 2 2 3 4 3 2 3 2 4 3 6 1 4 4 5 1 1 5 1 3 5 2 5 1 2
## [6733] 2 6 4 5 3 1 5 4 5 1 2 1 1 2 6 1 6 1 6 2 3 5 2 6 6 2 2 4 3 4 1 6 1 1 2 6
## [6769] 5 1 6 2 3 3 6 4 3 3 4 5 1 1 6 4 6 5 6 4 4 3 3 6 5 4 2 5 4 4 2 6 5 1 3 1
## [6805] 2 1 4 2 4 4 5 6 1 4 4 1 2 5 6 5 4 2 5 3 2 4 5 5 2 3 4 6 1 5 4 2 3 4 5 3
## [6841] 5 3 3 6 5 6 4 6 1 5 6 5 3 1 6 5 3 2 4 5 1 6 2 5 3 5 6 1 1 1 5 2 1 5 3 1
## [6877] 5 6 2 2 2 2 3 1 3 5 5 5 5 1 2 2 3 4 4 1 4 1 6 4 6 1 6 2 2 1 3 4 5 3 2 1
## [6913] 6 2 6 2 5 5 3 6 2 2 4 6 3 5 1 2 4 4 6 6 5 5 2 3 3 3 6 6 2 6 4 6 2 6 4 1
## [6949] 6 5 2 1 2 2 2 5 6 5 3 2 2 6 1 4 6 4 3 3 4 6 2 1 3 6 5 2 5 2 6 2 2 1 6 6
## [6985] 3 6 3 5 4 1 3 1 1 4 2 4 1 2 5 6 6 3 6 4 3 2 4 5 3 3 2 3 1 6 4 6 6 1 5 1
## [7021] 2 2 4 4 5 4 5 2 3 5 4 2 4 4 3 3 4 1 5 6 4 1 3 4 3 3 4 4 6 4 1 2 2 4 1 1
```

```
## [7057] 2 6 6 6 5 6 5 6 3 3 2 3 3 3 3 5 3 5 1 5 3 5 3 1 1 5 1 5 5 5 2 4 5 6 6 3
## [7093] 2 1 6 4 6 1 2 2 5 1 3 1 4 2 6 4 5 3 5 1 3 3 2 3 2 4 1 4 2 3 2 2 1 2 5 2
## [7129] 6 1 3 6 1 5 4 2 5 6 5 1 1 4 5 1 3 5 1 5 2 4 2 6 2 3 4 4 3 1 2 3 6 6 2 1
## [7165] 1 5 2 2 6 6 2 5 3 6 5 3 6 6 2 5 2 1 6 4 4 2 6 6 4 3 4 2 4 3 6 4 3 3 3 4
## [7201] 4 5 5 1 6 6 3 6 3 2 4 3 6 3 3 5 2 2 1 6 6 6 3 2 2 4 3 5 3 3 5 6 1 1 2 1
## [7237] 1 3 4 2 2 1 6 2 4 6 4 4 1 4 2 3 1 3 3 5 2 6 1 3 1 1 6 1 6 5 2 6 1 4 5 1
## [7273] 3 3 5 1 4 5 4 3 1 3 5 5 2 3 5 1 1 4 5 2 6 3 3 1 3 5 1 1 1 1 4 5 6 6 4 5
## [7309] 2 6 1 6 5 6 6 3 3 3 1 1 1 2 4 6 6 2 4 3 5 2 6 2 5 1 5 5 6 5 1 3 4 5 5 5
## [7345] 2 1 2 3 2 3 2 1 3 5 2 2 1 4 3 5 6 2 4 6 5 3 3 3 4 5 5 1 3 6 5 2 2 4 6 5
## [7381] 3 6 1 5 1 3 5 3 1 4 3 6 2 2 6 5 1 4 1 2 5 2 2 1 6 6 5 6 5 1 3 3 3 1 6 6
## [7417] 1 3 3 2 4 2 6 1 4 5 3 2 6 3 6 6 5 5 3 2 2 5 4 5 2 5 5 6 5 5 1 3 4 1 3 2
## [7453] 3 2 3 1 5 1 3 1 5 5 2 2 2 5 3 2 5 2 6 1 2 2 3 4 1 1 1 6 5 1 3 2 3 4 2 2
## [7489] 3 2 2 4 3 6 4 2 1 1 2 4 1 2 2 2 3 6 1 5 2 4 2 4 6 4 4 1 5 2 4 6 3 3 3 1
## [7525] 2 4 2 5 3 6 6 1 6 4 4 1 4 5 6 5 3 5 6 1 4 3 1 6 4 2 1 1 6 1 2 4 2 3 2 2
## [7561] 3 4 2 5 5 3 5 2 3 3 5 2 3 2 2 1 2 4 2 3 1 6 1 5 3 5 2 3 2 4 1 3 4 4 5 6
## [7597] 3 3 2 6 6 5 2 4 4 2 3 5 3 3 2 5 4 5 3 6 5 6 6 5 4 4 1 5 3 3 5 6 5 6 2 1
## [7633] 4 6 2 3 5 5 4 2 1 1 2 4 6 1 4 5 2 3 6 1 4 3 1 3 6 3 1 6 5 4 5 6 2 6 2 2
## [7669] 5 4 6 4 2 4 3 3 3 2 3 5 5 4 4 3 1 6 3 2 2 2 4 5 3 2 3 3 3 4 6 6 1 5 5 4
## [7705] 2 6 4 5 2 5 4 5 6 1 2 2 5 5 3 2 3 5 3 4 4 2 5 4 1 4 6 1 6 3 2 5 4 4 6 6
## [7741] 4 1 6 1 2 2 2 6 6 3 1 3 2 5 1 5 1 2 5 5 3 1 6 6 3 3 2 3 3 2 4 2 6 3 2 3
## [7777] 6 3 4 3 6 2 6 6 1 1 3 2 5 5 3 5 5 1 3 4 4 2 5 2 4 4 1 6 2 4 3 2 1 4 3 2
## [7813] 1 2 4 5 5 6 6 4 3 5 5 3 6 5 5 3 2 1 2 6 3 5 6 5 6 4 5 5 2 4 3 2 3 6 6 1
## [7849] 5 4 1 5 1 1 6 1 6 2 1 5 1 2 5 3 3 6 2 1 5 5 4 6 5 6 1 2 3 2 3 4 1 4 1 4
## [7885] 4 2 2 3 3 5 1 1 2 3 6 4 1 5 3 6 1 6 5 5 3 5 6 1 6 6 3 6 6 5 4 1 5 1 3 6
## [7921] 1 6 4 3 5 2 3 3 5 6 1 1 6 4 1 5 2 6 2 6 5 2 5 5 2 5 5 6 1 6 5 4 5 3 2 3
## [7957] 1 5 6 5 2 6 3 6 3 5 3 4 3 6 3 1 5 1 5 1 3 4 4 2 5 2 1 2 2 5 4 2 5 3 1 2
## [7993] 5 4 3 6 6 6 3 4 5 4 5 4 2 4 2 2 2 5 3 6 3 6 6 3 2 6 2 4 3 5 6 1 4 6 1 5
## [8029] 5 5 6 1 6 3 3 1 4 1 6 4 4 6 5 3 3 1 5 5 6 5 4 4 1 6 2 2 4 2 5 3 4 3 2 2
## [8065] 1 1 3 6 1 6 1 5 5 6 3 6 4 3 4 1 4 5 6 1 1 6 6 6 2 1 4 2 5 6 4 3 4 6 6 5
## [8101] 5 6 3 1 5 3 3 2 4 3 6 1 4 2 3 2 6 1 6 2 4 5 4 6 1 4 6 3 1 3 5 5 3 1 2 5
## [8137] 5 1 2 5 1 6 5 4 1 3 4 4 3 5 4 6 2 3 6 5 1 5 1 6 4 2 6 3 3 1 3 1 1 4 5 4
## [8173] 3 3 4 3 5 2 5 1 6 5 4 2 3 2 4 5 6 3 2 6 3 1 5 4 6 4 6 4 3 6 1 1 5 3 1 5
## [8209] 6 5 3 4 5 6 1 2 5 3 6 1 1 4 6 1 3 4 3 3 2 1 3 3 3 5 1 3 3 6 6 2 6 5 5 6
## [8245] 6 1 4 4 3 3 1 2 2 1 3 4 6 3 1 1 4 2 3 5 3 3 5 3 5 5 4 4 4 6 2 2 1 2 1 5
## [8281] 3 5 3 6 5 3 2 6 1 5 6 1 4 3 1 5 1 1 5 1 6 2 4 5 2 1 5 1 3 1 5 3 3 6 3 6
## [8317] 6 6 4 1 1 5 5 3 1 2 2 4 5 4 4 4 1 4 2 2 6 6 6 2 1 4 5 3 4 5 3 1 3 6 3 2
## [8353] 3 4 6 4 4 1 5 3 4 5 5 4 5 1 3 3 4 2 1 6 1 6 3 4 1 5 5 2 4 3 4 2 3 2 1 4
## [8389] 6 1 6 4 5 3 6 1 5 1 6 1 5 1 5 3 1 6 6 5 5 2 4 5 2 1 5 5 2 3 6 2 6 5 3 5
## [8425] 5 4 1 5 3 6 5 2 2 2 1 4 6 5 4 3 3 2 1 3 6 4 6 3 2 1 5 2 5 3 4 4 3 4 6 5
## [8461] 6 4 5 5 2 4 3 1 4 3 4 4 3 5 2 2 6 4 4 1 4 4 5 5 5 1 2 5 3 4 6 1 5 4 6 5
## [8497] 4 4 3 6 4 1 3 1 5 5 5 3 4 6 4 1 3 6 4 4 5 1 3 2 4 6 2 3 5 1 4 6 1 4 3 2
## [8533] 4 6 1 1 5 1 5 5 1 2 3 2 4 1 5 3 1 3 6 2 5 5 3 1 5 4 2 6 5 3 5 4 6 1 2 6
## [8569] 4 3 6 4 1 4 2 3 4 6 2 2 6 2 3 2 3 6 5 1 3 1 2 6 4 4 6 6 1 1 4 1 5 6 6 5
## [8605] 2 2 2 3 1 4 6 3 4 6 3 3 4 4 1 2 1 2 6 2 4 5 6 5 3 3 4 3 6 2 2 3 6 4 4 6
## [8641] 3 6 2 1 4 2 5 4 4 5 5 4 3 2 2 3 2 6 3 6 1 5 4 6 1 5 6 4 3 2 5 4 3 5 4 4
## [8677] 5 5 6 2 3 2 1 6 1 1 5 4 5 3 5 3 6 6 2 2 4 1 5 5 3 2 5 6 3 2 2 3 4 1 5 4
```

```
## [8713] 6 1 2 2 6 6 3 6 2 4 6 5 1 4 4 2 3 4 1 3 5 5 5 4 2 2 2 4 5 3 3 3 4 4 2 1
## [8749] 5 4 2 5 4 5 1 4 2 1 5 4 4 1 1 2 3 3 3 4 1 2 6 2 2 2 1 5 2 2 2 1 2 2 4 1
## [8785] 2 4 3 2 4 2 1 2 6 1 6 5 6 3 3 2 1 3 5 3 2 5 6 1 2 6 2 4 5 5 5 6 4 5 5 3
## [8821] 6 6 3 2 5 1 6 2 1 3 5 2 1 3 6 4 2 5 2 6 4 4 6 1 6 6 4 6 3 6 2 2 4 4 1 4
## [8857] 2 5 1 2 3 3 4 6 1 3 2 6 6 1 1 6 5 5 4 3 5 2 5 4 5 4 1 1 3 6 4 2 2 4 1 2
## [8893] 5 2 5 3 1 6 1 2 5 1 2 2 4 3 5 4 2 2 3 4 4 1 2 3 1 1 4 6 5 4 4 5 3 6 3 3
## [8929] 5 1 4 4 1 5 1 6 6 2 4 2 6 3 2 4 6 3 6 3 5 1 1 3 5 6 3 3 1 5 4 4 5 1 3 1
## [8965] 1 1 5 6 5 3 4 6 4 5 6 6 5 1 2 4 2 6 4 2 6 1 4 1 3 2 4 1 6 3 5 5 3 6 5 2
## [9001] 1 2 3 4 6 5 4 6 4 1 4 5 1 6 4 2 4 5 6 4 4 6 1 2 3 5 3 6 4 2 6 4 5 2 5 6
## [9037] 5 4 4 2 3 3 2 5 5 6 5 5 5 6 6 5 3 4 1 5 3 1 6 4 6 4 6 6 5 6 4 1 4 5 4 5
## [9073] 1 5 2 1 6 5 1 5 3 5 4 3 2 1 4 5 6 3 2 3 4 4 6 4 3 5 2 1 4 5 1 2 2 5 5 3
## [9109] 5 6 1 3 5 1 3 5 4 3 1 3 1 6 4 1 2 5 4 2 6 6 5 2 2 3 6 6 4 6 4 1 5 5 6 1
## [9145] 6 1 4 1 3 3 3 2 2 1 6 2 5 5 6 3 5 4 6 4 4 1 6 1 2 6 1 2 4 2 5 6 4 5 1 5
## [9181] 5 6 2 4 3 1 6 5 6 3 1 3 2 1 1 3 3 3 4 2 1 1 3 4 6 4 6 1 3 1 2 4 1 3 3 4
## [9217] 4 6 5 4 2 3 2 5 5 5 5 3 2 5 6 5 1 4 4 1 2 4 5 1 2 5 5 6 3 4 5 3 6 3 5 1
## [9253] 3 6 6 3 3 6 2 4 2 5 3 2 5 3 3 6 6 5 5 5 2 5 5 2 4 2 5 6 6 3 5 1 2 2 5 6
## [9289] 5 1 6 3 6 3 6 2 5 4 5 6 5 2 2 5 1 4 5 2 3 1 2 2 5 2 6 2 1 1 3 1 3 2 3 1
## [9325] 4 3 4 1 4 3 2 4 5 3 3 3 5 4 1 3 6 3 5 5 1 1 2 3 4 5 1 3 6 6 5 5 5 6 2 2
## [9361] 5 1 4 3 3 6 5 1 6 3 5 4 1 5 6 1 2 2 2 1 3 2 2 5 6 5 2 2 4 3 2 4 1 1 2 6
## [9397] 3 2 3 3 1 5 1 1 1 4 2 3 2 4 6 1 4 2 3 6 6 5 1 5 6 6 2 5 3 5 1 4 4 4 3 5
## [9433] 6 5 6 3 6 4 2 1 1 5 5 6 6 6 5 1 3 3 6 4 3 3 4 1 4 3 1 3 5 3 6 4 1 5 6 3
## [9469] 1 1 6 2 1 6 4 4 6 4 4 6 2 5 5 5 6 6 5 1 3 1 5 6 3 6 5 2 3 1 3 2 2 2 2 5
## [9505] 1 4 1 6 4 4 4 6 2 6 3 4 6 5 5 4 2 3 4 2 3 1 6 4 2 1 3 1 5 2 4 4 5 6 5 3
## [9541] 6 4 1 2 3 1 5 4 5 3 3 3 1 3 5 4 2 4 2 2 5 2 4 1 6 1 2 3 3 6 5 6 6 4 1 5
## [9577] 6 2 4 4 6 4 1 1 6 5 1 5 4 3 3 4 1 4 2 2 1 3 5 4 3 4 5 1 4 5 2 6 1 6 1 4
## [9613] 5 2 3 3 6 1 5 6 2 5 4 6 6 6 4 2 2 5 5 2 3 2 3 4 2 5 4 6 3 2 2 5 6 2 3 5
## [9649] 3 3 3 5 5 4 6 1 2 1 2 2 1 4 2 6 5 6 1 6 3 1 1 5 6 3 6 4 5 5 3 3 6 5 5 4
## [9685] 6 4 2 2 5 6 1 1 5 6 3 5 1 1 4 4 3 4 1 4 5 2 5 6 1 2 2 1 5 2 6 3 1 1 4 3
## [9721] 5 3 1 1 6 1 4 2 2 2 1 1 2 3 2 4 6 2 1 6 5 2 3 1 4 3 4 5 2 2 2 1 4 6 5 3
## [9757] 5 5 4 1 4 5 6 1 4 2 6 1 3 1 3 4 3 4 5 5 6 2 3 6 6 4 2 4 5 5 4 1 6 4 5 2
## [9793] 1 5 2 2 1 2 2 4 3 1 5 3 2 3 4 5 6 1 4 1 2 4 5 3 2 4 1 3 3 2 2 3 4 6 4 2
## [9829] 6 2 6 2 4 5 4 3 3 6 2 2 6 3 3 5 6 1 6 4 2 1 1 6 5 1 2 3 3 5 6 3 3 1 4 4
## [9865] 1 2 4 4 1 6 1 6 6 6 6 5 6 6 3 6 2 4 2 3 6 1 1 4 1 6 3 1 5 1 3 5 5 4 5 4
## [9901] 3 4 4 4 2 3 6 4 2 1 6 1 1 2 5 2 4 1 3 3 6 6 4 1 4 4 5 6 2 1 4 6 2 1 5 2
## [9937] 1 2 4 6 3 2 1 6 5 2 5 2 3 2 4 1 4 3 2 6 4 5 5 3 4 3 1 2 4 3 4 2 6 3 6 3
## [9973] 1 1 5 6 6 1 5 5 5 2 1 6 3 2 6 3 2 4 1 4 1 2 2 5 5 5 4 6
```

We can save this and plot it:

```
# save it
random.uniform <- round(runif(n = 10000, min = 0.5, max = 6.4999999))

# plot it
hist(random.uniform)
```

**Histogram of random.uniform**
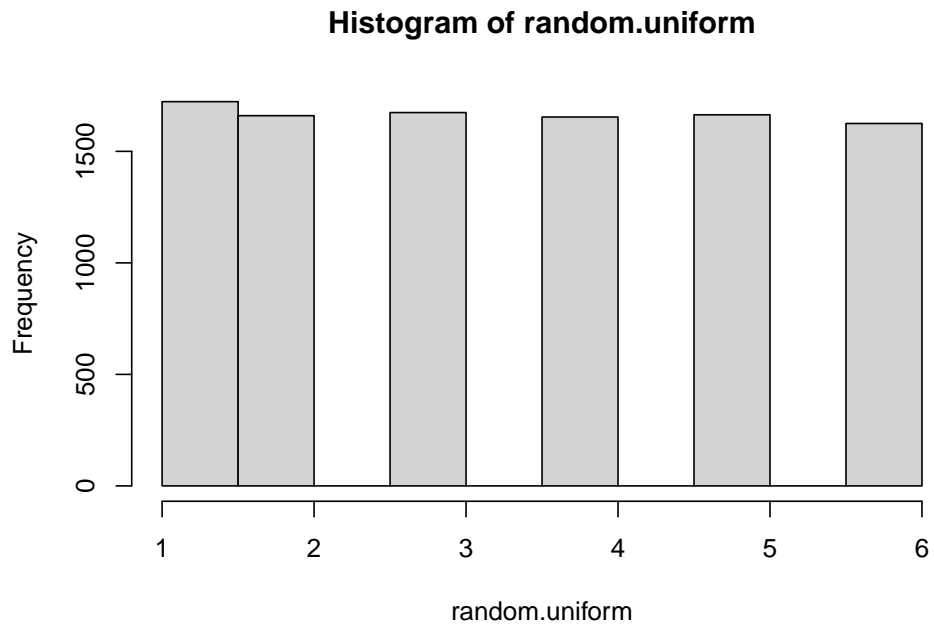


The height of the histogram is almost level, indicating that this is a uniform distribution.

```r
# save it
random.uniform <- round(runif(n = 1000, min = 0.5, max = 6.5))

# plot it
hist(random.uniform)
```

## Histogram of random.uniform
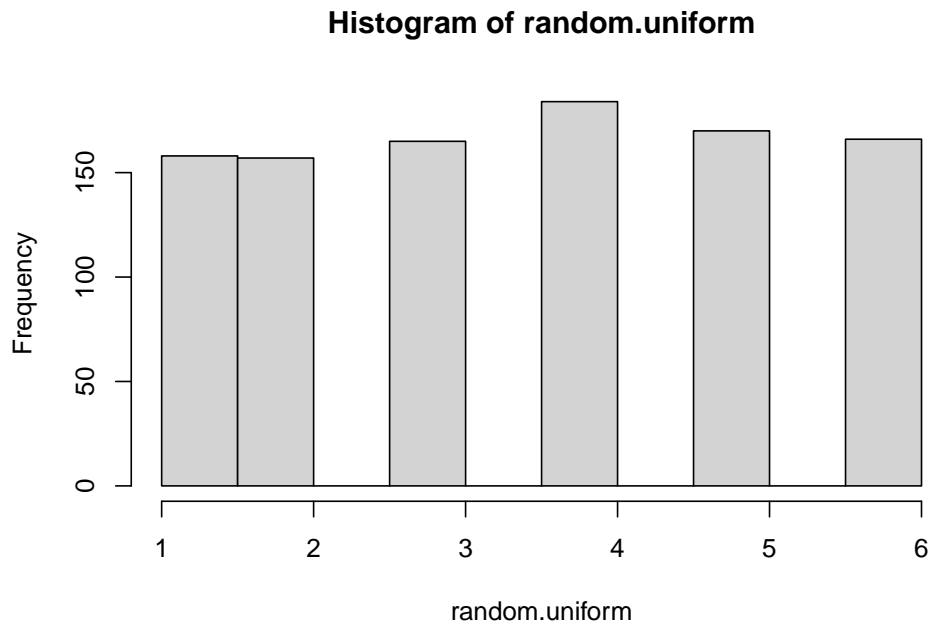


The code above works if the probability of each number (1, 2, 3...) is the same for all values, as for a fair dice.

We can get a similar result more dirctly using the function sample(), which acts like a virtual "pulling numbers from a hat" or rolling a dice.

First I make a vector to represent my die

```
dice <- c(1,2,3,4,5,6)
```

I can roll it once like this

```
sample(dice, size = 1)
```

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

I can roll it twice like this; replace = T means that the same value can occur (its like pulling numbers from a hat and putting each number back in to make it available again).

```
sample(dice, size = 1, replace = T)
```

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

I can simulate 50 dice like this:

```
sample(dice, size = 50, replace = T)
```

```
##  [1] 5 6 2 2 3 5 1 3 6 5 4 2 2 2 6 3 1 5 2 2 3 4 2 1 6 4 6 5 5 1 2 5 1 5 6 3 4 2
## [39] 5 4 5 3 5 4 5 6 4 6 3 3
```

If I want to represent an unfair dice I can add a new arguement which allows me to define the probabilites.

A fair dice would be this

```
probs.fair <- c(1/6, 1/6,1/6,1/6,1/6,1/6)
sample(dice, size = 1, replace = T,  prob = probs.fair)
```

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

A dice biased for 1could look like this

```
probs.biased <- c(2/6, 1/6, 1/6, 1/6, 1/6, 1/6)
sample(dice, size = 1, replace = T,  prob = probs.biased)
```

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

Let me see if its biased. I simulate 1000 dice

```
x <- sample(dice, size = 1000, replace = T,  prob = probs.biased)
hist(x)
```

**Histogram of x**
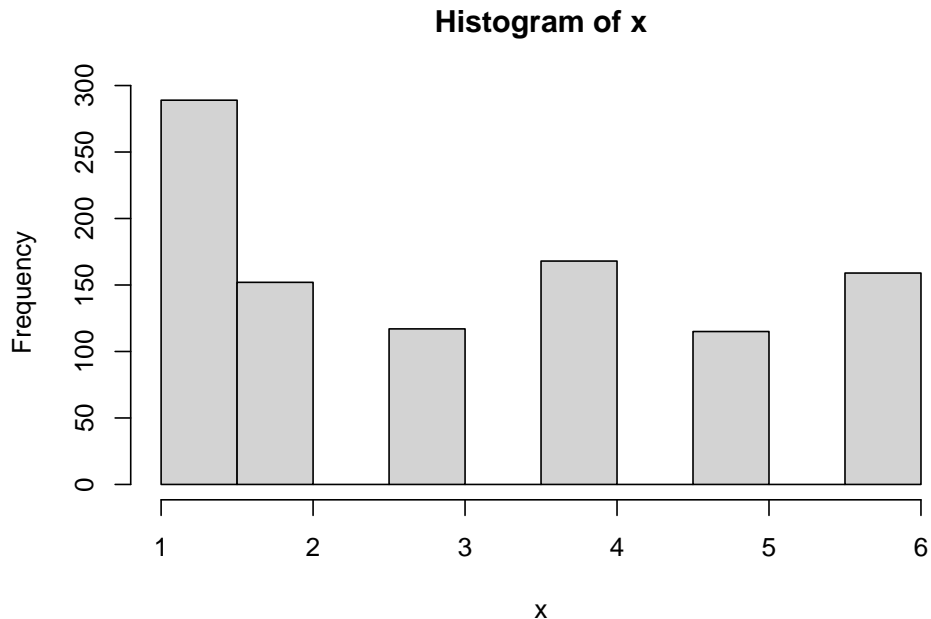


## 47.4 Lots of data in biology is normal

Data from a dice are discrete: you can only have certain numbers (1, 2, 3...) and can't have values in between (0.6). Also dice produce uniform striguation.

In contrast, the normal distribution is common in the world. Many things are normal or normal-ish. The follow code plots human heights (note: this doesn't

distinquish XY from XX but is still fairly normal)

```
data(Galton)
library(ggpubr)
gghistogram(data = Galton, x = "child")
```

```
## Warning: Using `bins = 30` by default. Pick better value with the argument
## `bins`.
```
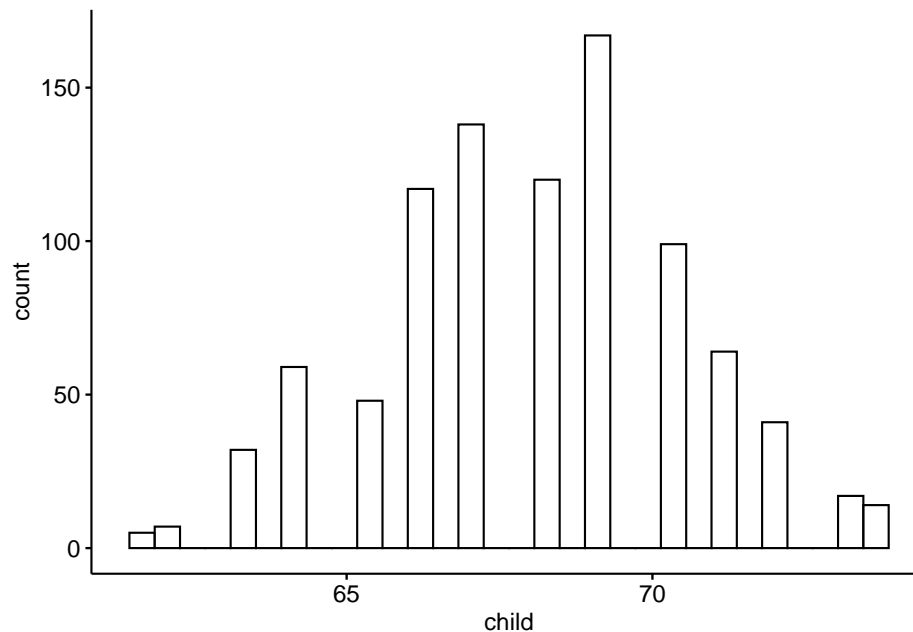


The code below plots data on the size of flower petals by species. The middle panel is fairly normal. The left panel is normalish, but squished becuase the values are small and you can't have flower sizes $<0$.

```
data(iris)
gghistogram(data = iris, x = "Petal.Length", facet.by = "Species")
```

```
## Warning: Using `bins = 30` by default. Pick better value with the argument
## `bins`.
```
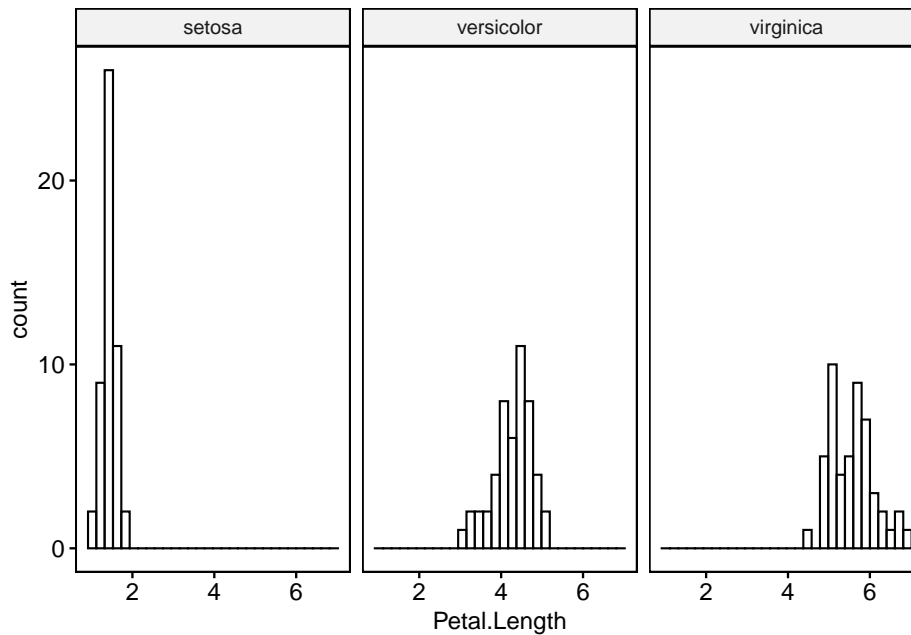
## 47.5  Simulating the normal distribution

We can simulate normal data in R using the rnorm function. Normal data is described by a mean (set here to 0) and a standard deviation (set here to 1). It has a typical bell-shaped curve. It is symmetrical - left of the mean has the same general shape as the data right of the mean. There is a mathematical formula that describes the shape of the curve, and you can generate random numbers from this. When you look at a normal distribution, the higher a given part of a curve is, the more likely a value under that part is to be randomly select. So values near the mean occur most commonly, while values far from the mean become increasinly less likely to occur.

```
n <- 1000
mean0 <- 0
sd1 <- 1

normal.data <- rnorm(n = n,
                     mean = mean0,
                     sd = sd1)
```

We can quickly plot it with hist()

```
hist(normal.data)
abline(v = mean0, col = 2)
```

**Histogram of normal.data**
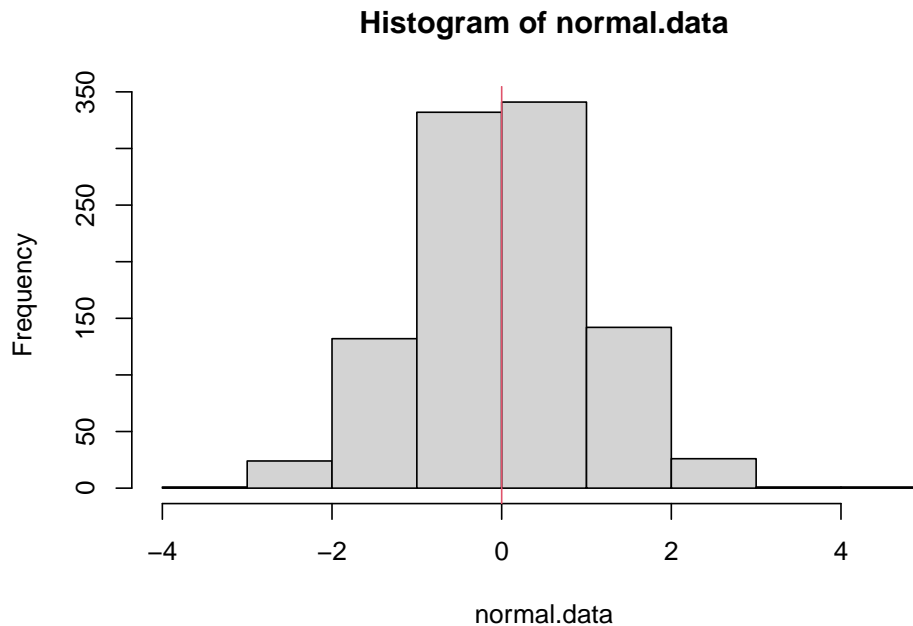


## 47.6 The extreme value distribution

Extreme value distributions are **skewed** and have a **long tail**.

The evd package allows us to simulate extreme value data s BLAST theory uses a particular extreme value distribution, the Gumbel distribution. We can simulate data from this distribution with rgumbel()

```
extreme.data <- rgumbel(n = n, loc=mean0, scale=1)
```

We can plot the data like this

```
hist(extreme.data)
abline(v = mean0, col = 2)
```

**Histogram of extreme.data**
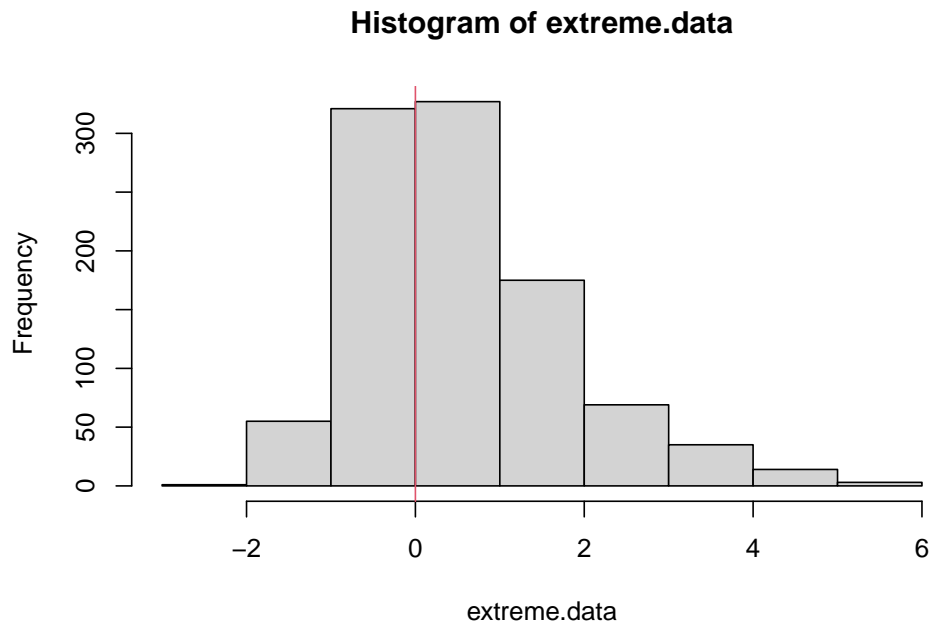


If you've taken a stats class you may have taken the log of a data set to make it more normal-like. If you do this with data that follows the extreme value distribution it doens't help - the data still don't look normal. Instead of being skewed one way, the skew is just flipped the other.

```
hist(log(extreme.data))
```

```
## Warning in log(extreme.data): NaNs produced
```

```
abline(v = mean0, col = 2)
```

**Histogram of log(extreme.data)**
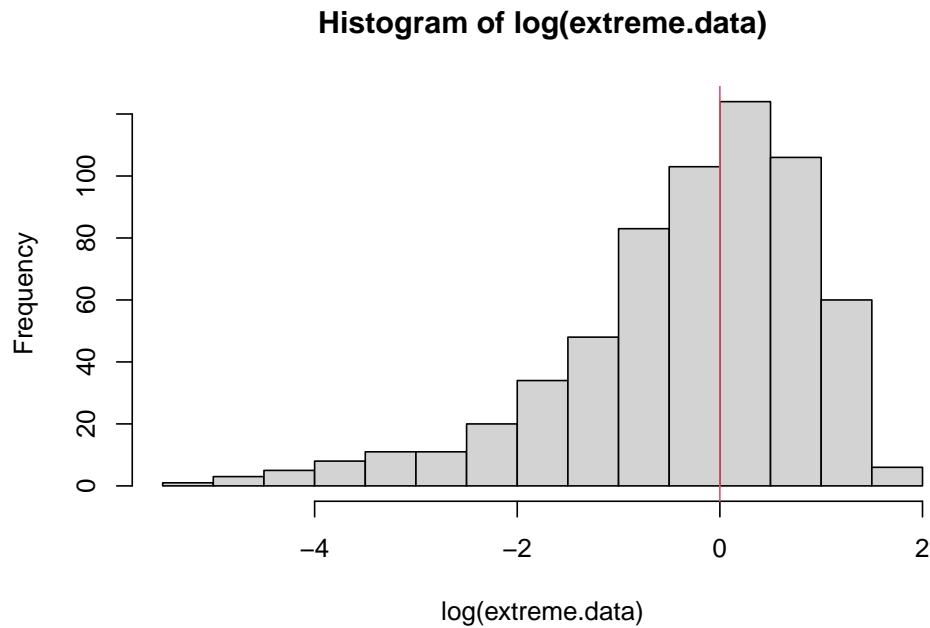


## 47.7 Replicating Figure 4.14

This code evokes Pevsner Figrure 4.14. You don't need to know these functions.

Note that the main differences correspond to values around -2 and 2.

```r
range.x.axis <- seq(from = -5,to = 5,length.out = 1000)
norm.curve    <- dnorm(  x = range.x.axis, mean = mean0, sd = sd1)
extreme.curve <- dgumbel(x = range.x.axis, loc = mean0, scale = 1)

plot(norm.curve~range.x.axis,type = "l")
points(extreme.curve~range.x.axis,type = "l", lty = 2, col = 2)
```
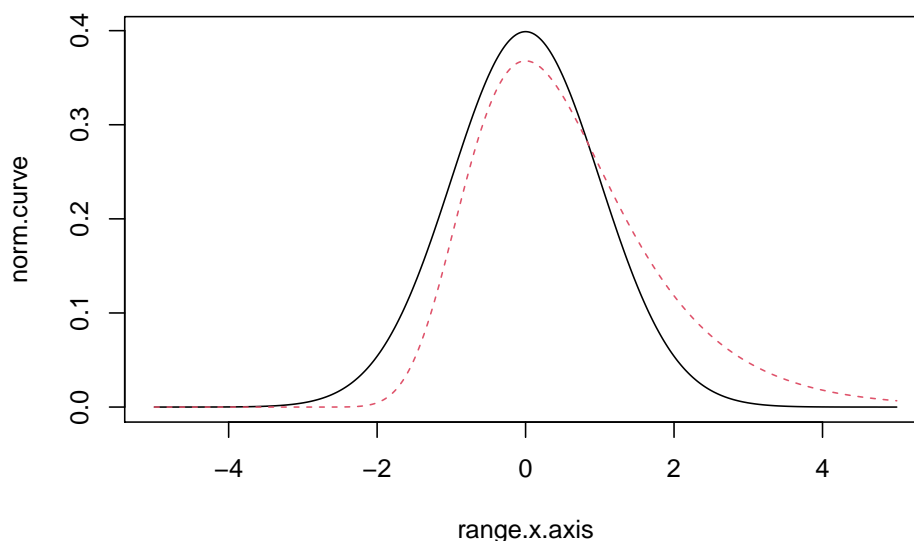
## 47.8   Why do we care?

The starting point for most statistics is the normal distribution. This is where we get z-scores, most p-values, standard errors, and most 95% confidence intervals. When we don't use normal distributions, we use other common distributions (eg the binomial distribution for binary data, the poisson for count data). None of these common distributions work if we want to do any statistics with DNA alginment; we need tow rok with the extreme value distribution. (Indeed, in 10 years of spending lots of time doing statiscs, this is the only application I've run in to of the EVD)

## Chapter 48

# Calculating E values, Bit scores, and P-values from BLAST output

**By:** Nathan Brouwer

**NOTE**: This has NOT be edited and NOT been spell checked

## 48.1 Introduction

BLAST gives you a number of statistics for understanding the output of a search. Below I R code showing how this output relates to the underlying equations used to derive them.

```
library(Biostrings)
library(seqinr)
data(BLOSUM62)
```

## 48.2 Case Study

To explore BLAST we'll do a protein "BLAST 2 sequences" search comparing an isoform of Drosophila enabled (NP_725859.2) against its human homolog (NP_060682.2). We'll use the default BLOSUM 62 scoring matrix, but modify a few things

- Use Gap costs of existence = 12 and extension = 1
- No compositional adjustment
- No masking low complexity sequences

Here are the sequences:

```
d.ena <- c("MAMKKLYAKTSFTSKKPSSAANSTPILAYHQQQHQQPGNGICEFQVVAPGHSGELMIRRSQSMHHKMSPPVGGLQ
KGLKYNQATATFHQWRDSKFVYGLNFSSQNDAENFARAMMHALEVLSGRVANNPGGPPTNGNGYEEDMGY
RTMTSEDAAILRQNNSIGGHVTPSAQTPTSQTNQNNIPQSPPTPQGHHRTSRNSLSAPPAPQPQQQQQQQ
QAQQMGQPGSHYGPTGNGPTSNGLPQQVNSQIPPAPQQQPQQQQFQQQQQQQQYQQMVQAGYAPSQQYQQ
PHYVLSNSNPNLTVHQYPTQQAQQQPPQAPQPPLQNGGMYMVGHGHLPSSASANSVVYASQQQMLPQAHP
QAPQAPTMPGPGYGGPPVPPPQQQAENPYGQVPMPPPMNPSQQQQPGQVPLNRMSSQGGPGGPPAPAPPP
PPPSFGGAAGGGPPPPAPPQMFNGAPPPPAMGGGPPPAPPAPPAMGGGPPPAPGGPGAPPPPPPPPGLGG
APKKEDPQADLMGSLASQLQQIKLKKNKVTTSAPENSGSSTSSGGSGNYGTIGRSSNGMASMMDEMAKTL
ARRRAQAEKKDPDPEAEVKKRPWEKSNTLPHKLSGGAGSGSAGSGHEGANGNSGGAGSNTTNSGGESPRP
MRKRFGSASEETILKQVNGDGLSLALSNGDLDTLKAEIVREMRLEIQKVKNEIIDAIKSEFNRR
")

h.ena <- c("MSEQSICQARAAVMVYDDANKKWVPAGGSTGFSRVHIYHHTGNNTFRVVGRKIQDHQVVINCAIPKGLKYNQATQ
ERQERLERQERLDRERQERQERERLERLERERQERERQEQLEREQLEWERERRISSAAAPASVETPLNSV
LGDSSASEPGLQAASQPAETPSQQGIVLGPLAPPPPPPLPPGPAQASVALPPPPGPPPPPPLPSTGPPPP
PPPPPLPNQVPPPPPPPPAPPLPASGFFLASMSEDNRPLTGLAAAIAGAKLRKVSRMEDTSFPSGGNAIG
VNSASSKTDTGRGNGPLPLGGSGLMEEMSALLARRRRIAEKGSTIETEQKEDKGEDSEPVTSKASSTSTP
EPTRKPWERTNTMNGSKSPVISRPKSTPLSQPSANGVQTEGLDYDRLKQDILDEMRKELTKLKEELIDAI
RQELSKSNTA")
```

We have to remove the newline characters:

```
d.ena <- gsub("\n","",d.ena)
h.ena <- gsub("\n","",h.ena)
```

## 48.3   BLAST Statistics

- E values = "Expected values"
- Bit scores = Scores which can be compared between searches with different parameters. BLAST output just refers to these as "scores" though sometimes indicates that the units is "bits". In equations this is S' ("S-prime").
- Scores (S) = raw scores derived from scoring matrix and gap penalties. Available when looking at the individual local alignments.

On the main output screen BLAST gives the "max score", "total score", and "E-value." This should be read as "max bit score" and "total bit score." The total bit score is the sum of the bit scores of all the high-scoring local alignments that were created within each database sequence.

For our case study, E is returned as 8e-70, the max bit score is 236, and the total bit score is 256 (again, BLAST is showing BIT scores, not raw scores). We can save these in R object

```
E <- 8e-70
S.bit.max <- 236
S.bit.total <- 256
```

The main page also tells use the size of our query sequence, which we'll call m. This is the length of our Drosophila enabled protein

```
m <- 834
```

Because we did a "BLAST two sequences" search we are also shown "Subject length"; this is the length of our human enabled homolog.

```
n <- 570
```

Since we specified a single sequence for comparison, this second sequence is the totality of our database for this search.

m x n is our **search space**. As will be discussed below, this is our raw search space; BLAST will use a small effective search space for actual calculations.

```
mn <- m*n
```

```
mn
```

```
## [1] 475380
```

m x n = 475380; this is the total number of amino acids compared by the search.

## 48.4 Dotplot

BLAST produces dotplots which are quick visual representations of alignments. When only a two sequences are compare with BLAST 2 sequences, the dotplot is a visual represent representation of the search space.

### 48.4.1 Dotplot in R - optional, experimental code

We can make a dotplot in R, but its kind of clunky and I have yet to be able to really reproduce BLASTs output.

First set up the data (don't worry about the details)

```
d.ena.vect <- unlist(strsplit(d.ena,""))
h.ena.vect <- unlist(strsplit(h.ena,""))
```

Then make the dotplot (this is sloooooooow, and I'm not happy with the results. I don't recommend running it.).

```
dotPlot(seq1 = d.ena.vect,
        seq2 = h.ena.vect,
        wsize = 3,
```

```
        wstep = 1,
        nmatch = 3)
```

I started writing my own dotplot code but haven't gotten very far.

```
df <- matrix(data = 0,
       nrow = length(h.ena.vect),
       ncol = length(h.ena.vect))

for(i in 1:length(h.ena.vect)){
  for(j in 1:length(h.ena.vect)){
    n.i <- d.ena.vect[i]
    m.j <- h.ena.vect[j]
    score.ij <- BLOSUM62[n.i, m.j]
    df[i,j] <- ifelse( score.ij> 0,score.ij,0)
  }
}

image(x = 1:length(h.ena.vect),
       y = 1:length(h.ena.vect),
       z = df)
```

## 48.5   BLAST "Search Summary"

BLAST provides a "Search summary" tab. At the top of the results three tabs are listed

Edit Search | Save Search | Search Summary

The search summary summarizes key aspects of how you defined the search (matrix, gap penalties, etc) and also the underlying parameters (lambda, K) defined in the equations below.

The "effective search space" is smaller than just m*n; after the test we'll talk about the difference between the search space and the effective search space.

I'll save the effective search space as an object

```
mn.effective <- 428800
```

Lambda can be obtained from the "search summary" tab. We want the value in the right-hand column.

```
lambda <- 0.283
```

K is also in the search summary tab.

```
K <- 0.059
```

## 48.6 BLAST Alignments

On its main results page BLAST report the bit scores. BLAST reports raw scores derived directly from the scoring matrix on the "Alignments" tab.

When two sequences are compared BLAST can find more than one high-scoring, non-overlapping alignment. The Alignment tab shows us the two alignments for this search which passed BLASTs thresholds.

The first alignment has the annotation under "Score" as "236 bits(570)". This is "bit score = 236 (raw score = 570)". Bit score is S' (S prime) and raw score is just S in equations. To be clear we'll call this S.bit.max. S we'll call S.max. We made S.bit.max above.

```
S.max <- 570
```

There's a second alignment which has the annotation "19.6 bits(38).

```
S.bit.other <- 19.6
S.other <- 38
```

Note that

```
S.max+S.bit.other
```

```
## [1] 589.6
```

```
round(S.max+S.bit.other)
```

```
## [1] 590
```

This is the total bit score (rounded up) that BLAST reports on the main results page

```
S.bit.total == round(S.max + S.bit.other)
```

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

The local alignments also have an E value listed. The E-value for the best alignment is identical to the E-value from the main results page.

## 48.7 Equations

Equation numbers refer to Pevsner

### 48.7.1 E value (eq 4.5, pg 142)

The E value is calculate as: $E = Kmne^{(-lambda*S)}$

- K = a constant dependent on the scoring matrix (eg K for PAM200 is different than BLOSUM62)
- m and n are the size of the query sequence and the size of the database

- m*n = the search space: all possible aligned positions under consideration; in practice this gets tweaked to be the effective search space.

- e = e^...; the exp() function in R
- lambda = A constant; depends on scoring matrix (eg lambda for PAM200 is different than BLOSUM62)
- S = raw score from local BLAST alignment (raw score, NOT bit score)

BLAST uses an **effective search space** which can be obtained from the Search Summary tab.

We can try to replicate BLASTs calculation with this equation. Instead of m*n we'll use mn.effective, but ignore why for now. We'll use S.max instead of S to be clear which of the 2 searches we're looking at.

I'll do the calculation and save it into an object called "E from K" to distinguish it from the official E from BLAST

```
E.from.K <- K*mn.effective*exp(-lambda*S.max)
```

```
E.from.K
```

```
## [1] 2.223637e-66
```

This result is fairly close but not exactly to what BLAST reports as the E value.

```
E
```

```
## [1] 8e-70
```

This could be within rounding error. For example, perhaps lambda or K have been rounded off. It could also potentially be due to R's limitations on working with a value like e-66.

Let's check the other S value to see what we get

```
K*mn.effective*exp(-lambda*S.other)
```

```
## [1] 0.5403858
```

This is very close the the value reported in BLAST.

## 48.8 Bit Scores (S')

Bit scores can be compared between BLAST searches.

S' = (lambda*S - ln K)/(ln 2)

Recall that the natural log in R is just log()

We can try to reproduce BLASTS bit score. This doesn't require any exceptional calculations on R's part so there's should be numeric problems.

```
(lambda*S.max - log(K))/log(2)
```

```
## [1] 236.8043
```

```
S.bit.max
```

```
## [1] 236
```

This is very very close to what BLAST reports, but is just a bit off. The alignment score (S.max) should be reported exactly because scoring matrices use integer values. For example

```
data("BLOSUM62")
BLOSUM62[1:10,1:10]
```

```
##     A  R  N  D  C  Q  E  G  H  I
## A   4 -1 -2 -2  0 -1 -1  0 -2 -1
## R  -1  5  0 -2 -3  1  0 -2  0 -3
## N  -2  0  6  1 -3  0  0  0  1 -3
## D  -2 -2  1  6 -3  0  2 -1 -1 -3
## C   0 -3 -3 -3  9 -3 -4 -3 -3 -1
## Q  -1  1  0  0 -3  5  2 -2  0 -3
## E  -1  0  0  2 -4  2  5 -2  0 -3
## G   0 -2  0 -1 -3 -2 -2  6 -2 -4
## H  -2  0  1 -1 -3  0  0 -2  8 -3
## I  -1 -3 -3 -3 -1 -3 -3 -4 -3  4
```

Lambda and/or K might be reported or rounded just a bit off.

## 48.9 E from Bit scores

E = mn*2^-S'

For our best score

```
E.from.bit <- mn.effective*2^(-S.bit.max)
```

```
E
```

```
## [1] 8e-70
```

```
E.from.bit
```

```
## [1] 3.883075e-66
```

```
E.from.K
```

```
## [1] 2.223637e-66
```

The official E from BLAST is larger by a bit, while E from the bit score and E from the K (and Lambda) are the same order of magnitude. Not sure what's

going on here.

We can see what happens if we calculate "E from K" using the total bit score

```
K*mn.effective*exp(-lambda*(S.max+S.other))
```

```
## [1] 4.749644e-71
```

From our E value we can back calculate our (effective) search space

From

E = mn*2^-S'

we divide by both sides by 2^-S' to get. E/2^-S' = mn

So mn is

```
E/(2^-S.bit.max)
```

```
## [1] 88.34235
```

Which is way off.

But

```
E.from.K/(2^-S.bit.max)
```

```
## [1] 245551.7
```

```
mn
```

```
## [1] 475380
```

is kinda close.

## 48.10   P values

P = 1 - e^-E

For the second of the two alignments

```
E.other <- 0.53
1 - exp(-E.other)
```

```
## [1] 0.411395
```

You can convert from a p-value if you had it to an E-value

P = 1 - e^-E -1+P = -e^-E 1-P = e^-E log(1-P) = -E -1*log(1-P) = E

output: html_document editor_options: chunk_output_type: console — # Calculating the number of possible phylogenetic trees

```
library(compbio4all)
```

## 48.11 Repliminaries

## 48.12 Concepts

### 48.12.0.1 Biology Concepts / Vocab

- toplogy
- rotation
- Number of possible phylogenetics trees

## 48.13 Functions

- factorial

## 48.14 Introduction

**Phylogenetics** is one of the original fields of computational biology. This is because as the number of **taxa** that need to be organized on a tree increases, the number of *unique* phylogenetic trees describing possible relationships among those taxa increases rapidly.

The following refers to **bifurcating** tree which always branch into two branches at each **node**. There are the type of branches built for phylogenies.

## 48.15 Number of rooted trees

The number of possible rooted phylogenetic trees is calculated using the equation below, where n is the number of taxa:

In text, the equation looks like this: (2n-3)!/(2^(n-2))(n-2)!

Written in a math book it would look like this:

$\frac{(2*n-3)!}{2^{n-2}*(n-2)!}$

This equation is occasionally mis-printed (as I did in the part of this original assignment) so its worth being explicit:

The numerator is: (2n-3)!, where "!" is factorial The denominator is: (2^n-2)*(n-2)!; the factorial is evaluated first.

So, for n = 3 taxa

(2n-3)!/(2^(n-2))(n-2)! = (2x3-3)!/(2^(3-2))x(3-2)! = (6 -3)!/(2^(1))x(1)! = (3)!/(2)x(1)! = (1 x 2 x 3)/(2 x 1) = 6 / 2 = 3

One of our first tasks will be to translate this equation into R code.

## 48.16   Number of rooted trees in R

In R we take factorials using the `factorial()` function

```
factorial(3)
```

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

```
3*2*1
```

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

```
factorial(4)
```

```
## [1] 24
```

```
4*3*2*1
```

```
## [1] 24
```

We can re-write our text equation above as:   factorial(2n-3)/(2^(n-2))*(factorial(n-2))

In this would be

```
n<-3
```

```
factorial(2*n-3)/((2^(n-2))*factorial(n-2))
```

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

All of the parentheses make this a bit nutty. Let me write this out as a separate numerator and denominator

```
#numerator
numerator   <- factorial(2*n-3)

#denominator
denominator <- 2^(n-2)*factorial(n-2)

#division
numerator/denominator
```

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

### 48.16.1   Unique topologies versus rotations

Note that this applies to all tree typologies. For n = 3 species the three typologies are

1. ((Human, Chimp), Gorilla)
2. ((Human, Gorilla), Chimp)
3. ((Chimp, Gorilla), Human)

In this discussion Since we are referring to evolutionary *unique* typologies. Therefore order within a clade does *not* matter. Therefore, these two are identical typologies and don't get counted among the three unique typologies the equation is telling use exist:.

1. ((Human, Chimp), Gorilla)
2. ((Chimp, Human), Gorilla)

## 48.17   Bilography

This topic is discussed on page 187-189 of Baum & Smith Tree Thinking. They discuss it in the concept of **maximum parsimony** estimation of phylogenetic trees.

# Chapter 49

# UPGMA the hard way

```
library(compbio4all)
```

## 49.1 Introudction to UPGMA

- UPGMA is a basic **clustering algorithm** which can be used for building phylognetic tree from **distance matrices**.

- It still has some applications because it is fast, but it has mostly beenn replaced by Neighbor Joining and Minimum Evolution methods.
- UPGMA is still useful for teaching the basic steps involved in data clustering and building trees.
- UPGMA is a particular variant of a general class of algorithms. Another varient is WPGMA.

## 49.2 Preliminaries

### 49.2.1 R Libraries

The phangorn package contains the functions

- upgma()
- wpgma()

The ape package has the function nj() for neighbor joinging, which is useful for comparison to UPGMA.

#### 49.2.1.1  Phylogenetic libraries

```
#install.packages("ape")
#install.packages("phangorn")
library(ape)
library(phangorn)
```

```
##
## Attaching package: 'phangorn'

## The following object is masked from 'package:igraph':
##
##     diversity
```

#### 49.2.1.2  Plotting libraries

```
#install.packages("plotrix")
library(plotrix)
```

### 49.2.2  Custom function we'll use

To help us understand clustering and UPGMA I've written a cusome function.

This function, plot_dist_as_nmds(), will make a 2-dimensional (2D) representation of the distance matrices we'll be working with. This is only an approximation and will occassionally result in distorions.

What this function does is take a **distance matix** and attempt to plot the data as if they were in 2-dimensional space. If we were giving it the distance between building on campus it would plot a reasonable map. Since we'll be plotting data that isn't physical distance but genetic distances this will not be a real evolutionary tree. The intent here is to help you build your intuition about what a distane matrix is.

```
## Load ALL of this code

# TODO: ylim and xlim are current hard coded

plot_dist_as_nmds <- function(mat, x.lim = c(-20,20), y.lim = c(-20,20)){
  matno0 <- mat

matno0[is.na(matno0)] <- 0


d2  <- stats::cmdscale(matno0, add =T)
x <- d2$points[,1]
y <- d2$points[,2]
```

```
plot(x,y, main = "APPROXIMATE 2D distances",
     xlim = x.lim,
     ylim = y.lim)
text(x, y, labels = row.names(d2$points), cex=3)

return(d2)


}
```

## 49.3 Data matrix

We'll use data from a study that built a tree using rRNA from bacteria.

The data and a worked example are available at https://en.wikipedia.org/wik
i/UPGMA#First_step

This particular matrix is set up as the number of bases that are different between
each sequence. (In the next unit we'll discuss how they didn't use just a simple
count of the differences).

### 49.3.1 Building the matrix

One way to make the data is to use the rbind() individual vectors as rows

```
# The data
#      a   b    c    d    e
a <- c(0,    17, 21, 31, 23)
b <- c(17, 0,     30, 34, 21)
c   <- c(21, 30, 0,   28, 39)
d   <- c(31, 34, 28,    0, 43)
e   <- c(23, 21, 39, 43,    0)

# Bind into a matrix
dmat1 <- rbind(a,b,c,d,e)
```

rbind() takes each vector (a through e) and glues them together into a matrix

```
is(dmat1)
```

```
## [1] "matrix"          "array"          "mMatrix"         "structure"
## [5] "vector"          "vector_OR_Vector" "vector_OR_factor"
```

As a challenge, can you take these data and make these data into a matrix using
the matrix() function?

The finished matrix looks like this

```
dmat1
```

```
##   [,1] [,2] [,3] [,4] [,5]
## a    0   17   21   31   23
## b   17    0   30   34   21
## c   21   30    0   28   39
## d   31   34   28    0   43
## e   23   21   39   43    0
```

Add names using colnames()

```
colnames(dmat1) <- c("a","b","c","d","e")
```

The matrix is **symmetrical**.

**Challenge questions:** How would we interpret the diagonal? Why is it not 1? Why is this matrix symmetrical?

Its easier if you make redundant elements NA because we don't need them.

We can access the **diagonal** of the matrix directly and set it to NA using the diag() function

```
diag(dmat1) <- NA
```

Now the matrix looks like this

```
dmat1
```

```
##    a  b  c  d  e
## a NA 17 21 31 23
## b 17 NA 30 34 21
## c 21 30 NA 28 39
## d 31 34 28 NA 43
## e 23 21 39 43 NA
```

We only need the lower left-hand part of the matrix. We can set the upper right-hand part of the matrix to NA like this:

```
dmat1[upper.tri(dmat1)] <- NA
```

(don't worry about exactly what's going on here)

Matrix now looks like this

```
dmat1
```

```
##    a  b  c  d  e
## a NA NA NA NA NA
## b 17 NA NA NA NA
## c 21 30 NA NA NA
## d 31 34 28 NA NA
## e 23 21 39 43 NA
```

## 49.4 What does this distance matrix represent?

### 49.4.1 Genetic distances conceptualized as pairwise linear distances

- Each element of the matrix is a **pairwise distance.**

- These distances were computed using **pairwise alignments** of each sequence.
- In this case, it is the number of nucleotides that are different between two sequences.
- As is standard, these alginments ignored any **indels.**
- There are 5 sequences, and if each is compared against the other then there are 10 total **distances.**

- Mathematically these are called **Hamming distances**.

Each distance is the distance between two species, and it can be conceptualized on its own as a linear distance.

You could visualize a single **pairwise distance** like a linear distance like this:

(note: you don't need to understand all of this code, but you do need to understand how it relates to the distance amtrix)

```
par(mfrow = c(1,1))
plot(0, 0, xlim = c(0,max(dmat1,na.rm = T)),
     ylim = c(0,12),
     type = "l",
     xlab = c("Distance (number of mutations)"),
     ylab = "Pairwise comparison",
     main = "Linear distance a vs. b")
points(0, 1, pch = "a")
points(dmat1[2,1], 1, pch = "b")
segments(x0 = 0, y0 = 1,
         x1 = dmat1[2,1],y1 = 1)
abline(h = 1.5, col = 2)
```

**Linear distance a vs. b**
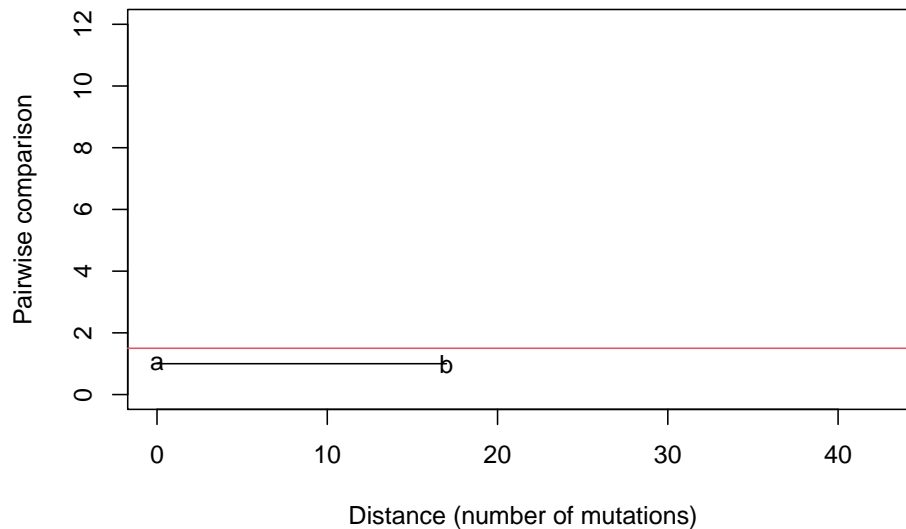


We can look at all of the distance Note that each line is *independent* and cannot be compared (the y axis has no meaning). This is NOT a 2D map!

```r
# This code is complex and the goal is to assess the plot, not understand the code
plot(0, 0, xlim = c(0,max(dmat1,na.rm = T)),
     ylim = c(0,12),
     type = "l",
     xlab = c("Distance (number of mutations)"),
     ylab = "Pairwise comparison")
pair.k <-0
for(i in 1:nrow(dmat1)){
  for(j in 1:nrow(dmat1)){

dist.ij <- dmat1[i,j]
if(is.na(dist.ij) == FALSE){
  pair.k <- pair.k+1


spp.i <- rownames(dmat1)[i]
spp.j <- colnames(dmat1)[j]
points(-0.5,                pair.k, pch = spp.j, col = which(LETTERS %in% toupper(spp.j)
points(dist.ij+0.75,       pair.k, pch =spp.i, col = which(LETTERS %in% toupper(spp.i)
segments(x0 = 0, y0 = pair.k,
         x1 = dist.ij, y1 = pair.k)
abline(h = pair.k+0.5, col = 2)
}
```

```
    }
}
```
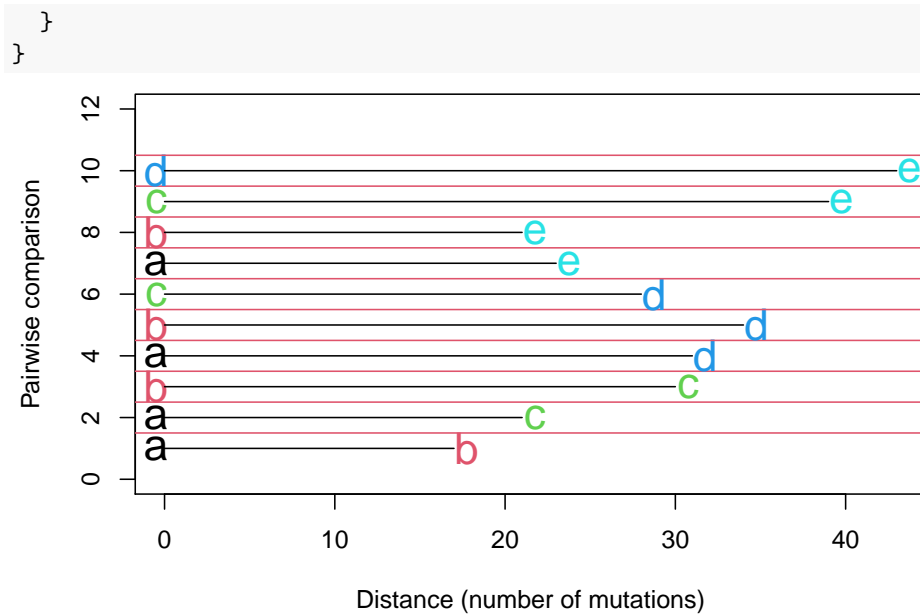


- We can see that the distance between a and b is the smallest, at 17 amino acid changes.
- UPGMA, WPGMA and related algorithms all begin by identifying the two taxa that are closest together.
- These 2 taxa form a clade (cluster). THe algorithm then proceeds by finding the next taxa (eg, c, d or e) which is closest to that clade of a and b
- We can summarize this first clad using **Newick notation** as: (a, b)

## 49.4.2 Genetic distances conceptualized as 2D map of linear distances

- We can extend the analogy of linear distance to two dimensions.

- With some fancy math we can make a 2D plot where the distances in 2D space are *similar* to the distances in the matrix.

- This is NOT exact and should not be taken literally. (See Pevsner Chapter 7 for a similar use of a 2D map)
- This is just a tool for teaching and not something done as part of an actual analysis.

```
par(mfrow = c(1,1), mar= c(2,2,2,2))
dmat1no0 <- dmat1

dmat1no0[is.na(dmat1no0)] <- 0
```

```
d2  <- cmdscale(dmat1no0, add =T)
x <- d2$points[,1]
y <- d2$points[,2]
plot(x,y, main = "APPROXIMATE 2D distances",xlim = c(-20,20), ylim = c(-20,20))
text(x, y, labels = row.names(d2$points), cex=3)
```

**APPROXIMATE 2D distances**
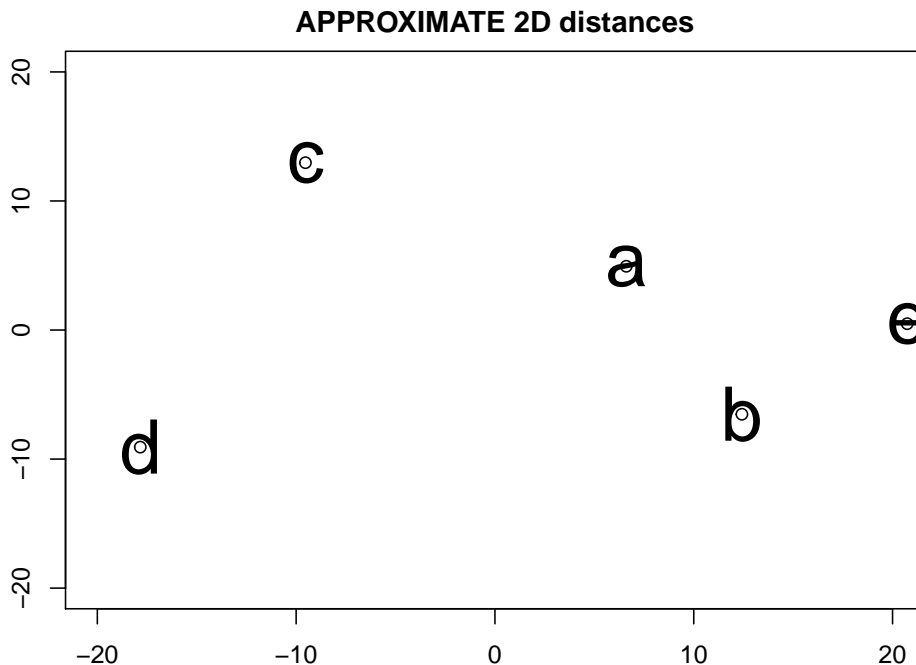
- In this plot, things appear to have been distorted a bit.
- a and b are close to each other as expect,
- but b looks closest than e (to me).

- It does show how a and b are close, and d and e are further apart.

## 49.5  From linear distances to "tree distances"

**Genetic distances** aren't actually linear - we are assuming a "tree shape" to represent the divergence of two species genetically (and possibly geographically).

The simplest methods of building trees assume that all branch lengths terminate at the same distance from the root, and the total mutational distance traversed by each species is the same. (This is the assumption or constraint related to **ultrametricity**).

The distance between a and b is 17. We assume that both species a and species b have undergone the same number of mutations since they split, so the branch

lengths for each are $17/2 = 8.5$. (You obviously can't have 8.5 mutations, but this is how the math works).

The following plots shows how the linear distnace of 17 between a and b (bottom) gets split into two branches, each of 8.5.

```r
# This code is dense; run it and focus on the plot
par(mfrow = c(1,1), mar= c(2,2,2,2))
plot(0, 0, xlim = c(0,max(dmat1,na.rm = T)),
     ylim = c(0,12),
     type = "l",
     xlab = c("Distance (number of mutations)"),
     ylab = "Pairwise comparison")
points(0, 1, pch = "a")
points(dmat1[2,1], 1, pch = "b")
segments(x0 = 0, y0 = 1,
         x1 = dmat1[2,1],y1 = 1)
abline(h = 1.5, col = 2)


branch.length <- dmat1[2,1]/2
points(0, 4)
points(branch.length+0.75, 4, pch = "a")
points(0, 6)
points(branch.length+0.75, 6, pch = "b")

segments(x0 = 0, y0 = 4,
         x1 = branch.length,y1 = 4)
segments(x0 = 0, y0 = 6,
         x1 = branch.length,y1 = 6)

segments(x0 = 0, y0 = 4,
         x1 = 0, y1 = 6, col = 1, lty = 2)

abline(h = 8, col = 2)
```
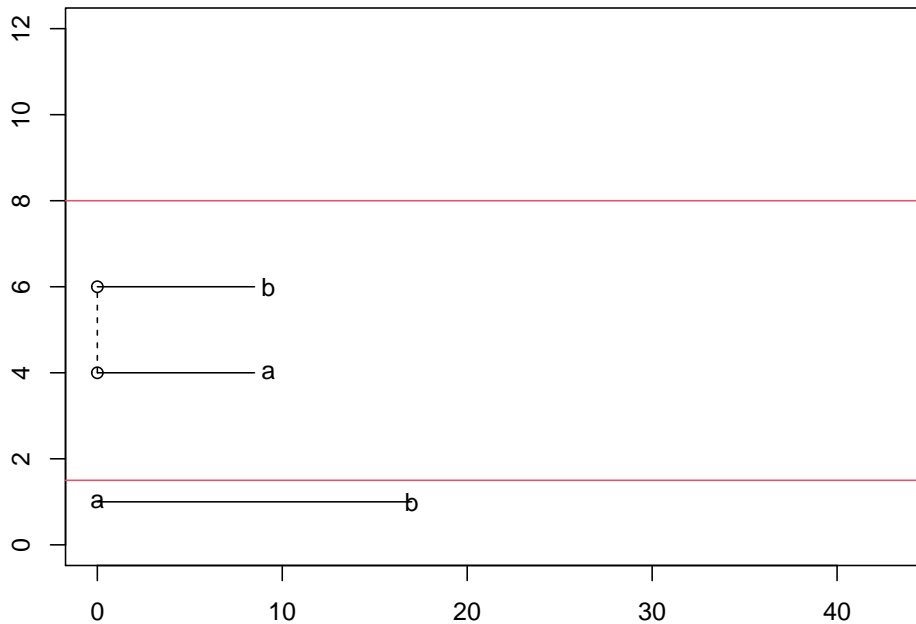
Again, in **Newick notation**, we'd write (a, b) to indicate that a and b are a clade.

## 49.6   The UPGMA algorithm

*The UPGMA algorithm is described by Swofford et al (1996?, page 486) as below.*
The word "cluster" means "taxa" in some cases and "clade" in others.
* This is a bit tricky at first but once you get a sense for the whole algorithm it will make more sense. * To keep track of everything we'll use subscripts i and j, represented in plain text as e.g. d.ij for the distance (d) in row i and column j.

### 49.6.1   Step 1: find the 2 taxa that are currently closest together

1. "Given a matrix of **pairwise distances**, find the clusters (taxa) i and j such the d.ij is the minmum value in the table." (d.ij is the distance between taxa i and taxa j)

In our example, d.ij = d.ba = 17, with i = b and j = a.

```
dmat1["b","a"]
```

```
## [1] 17
```

### 49.6.2  Step 2: calculate the branch lengths connect the taxa to their shared node

2. "Define the depth of the branching between i and j (l.ij) to be d.ij/2" ("depth of branching" = **branch length**; so i is attached to a branch of d.ij/2 and j is attached to a separate branch of d.ij/2)

d.ab/2 = 8.5

### 49.6.3  Step 3a: determine if the algorithm is done

3. "If i and j were the last two clusters, the tree is complete"

### 49.6.4  Step 3a: Continue on if needed and combine the previous cluster into cluster "u"

3. If there is more to do, "create a new cluster called u". This new cluster is a clade of a and b will represent the combined features of a and b.

What this means is that now that we've mapped taxa a and b into a clade we need to update our matrix to determine how far all the remaining taca (c, d and e) are from that clade.

### 49.6.5  Step 4: Determine the distance from u to the other remaining clusters.

4. "Define the distance from [the new cluster] u to each other [remaining] cluster (k, with k==i or j) [here cluster = clade in the first part of the sentence and taxa in the other] to be an **average** of the distnace d.ki and d.jk". That is, calculate the distance from the new clade u to all the remaining taxa, which we'll generically call "k". The new distance will be called d.ku, for "distance from taxa k to clade u."

The exact calculation of the average can take different forms. This will be unpacked below.

### 49.6.6  Step 5: Create a new matrix

5. "Go back to step 1 with one less cluster; cluster i and j have been eliminated, and cluster u has been added." The distance matrix is now shrunk with all distance related to i and j removed and replaced by information about u.

### 49.6.7  Key step: calculating the distance from a clade to the remaining taxa

For step 4, UPGMA uses the equation below to calculate distances. This will require some notation:

- Ti = the number of taxa in cluster i. This is 1 if the "cluster" is a single taxa (eg. 1 species), or it can be >1 if a previous iteration has created a clade.

- Tj = the number of taxa in cluster j.
- d.ku = the distance from k to u. k and u can be individual taxa OR clades created by the algorithm.

The key quantity d.ku is calculated as:

d.ku = (Ti x d.ki + Tj x d.kj)/(Ti + Tj)

(You should know this equation AND you should be able to write R code to calcualte it)

You can think of this as a **weighted mean** where the **weights** are the number of taxa in each clade. Its a very annoying that the "U" in UPGMA means "un-weighted" even though there is, from one perspective weighting, but this is how the math people say we should think about it (the un-weighted part has to do with some other aspect of the math).

In the above outline of the algorithm, if i and j are the first two taxa grouped then the equation would be

d.ku = (Ti x d.ki + Tj x d.kj)/(Ti + Tj) Ti = 1, Tj = 1, so d.ku = (1 x d.ki + 1 x d.kj)/(1 + 1) d.ku = (d.ki + d.kj)/2

## 49.7   Calculate 2D representation

We can think about this using a 2D representation.

First, plot the 2D map.

```
dmat1.out<- plot_dist_as_nmds(dmat1)
```

**APPROXIMATE 2D distances**
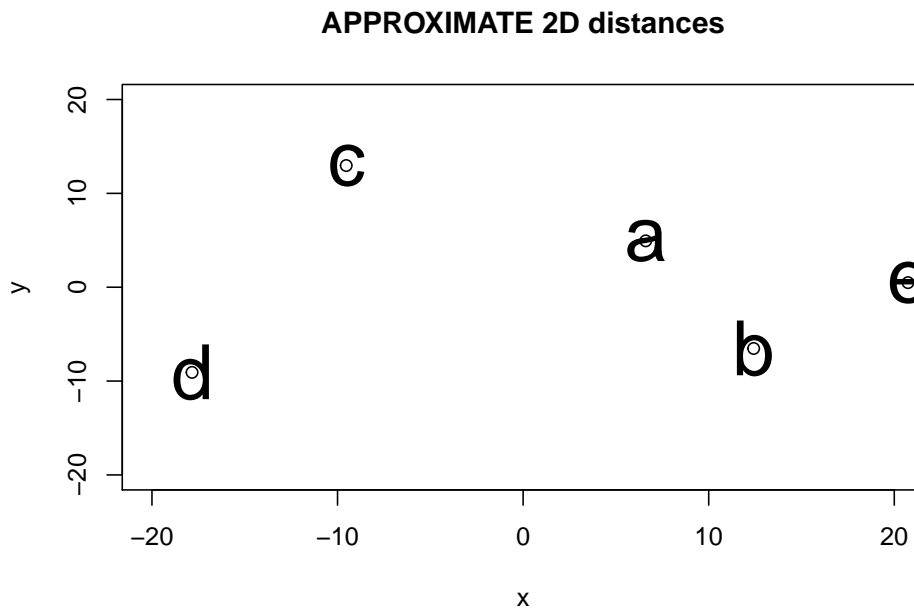


a and b are closest together, so we group them

I'll show this in the code below; you don't need to understand exactly how this code works.

```
dmat1.out<- plot_dist_as_nmds(dmat1)

#get x y coord of a and b

a.<-dmat1.out$points["a",]
b.<-dmat1.out$points["b",]

plot_dist_as_nmds(dmat1)
```

```
## $points
##          [,1]       [,2]
## a    6.606653   4.9400937
## b   12.421702  -6.5267890
## c   -9.532025  12.9665271
## d  -17.839642  -9.0739515
## e   20.738283   0.4897976
##
## $eig
## NULL
##
## $x
## NULL
```

```
##
## $ac
## [1] -7.934712e-15
##
## $GOF
## [1] 0.6755273 0.9367305
```

```
draw.ellipse(x = mean(c(a.[1],b.[1])),
             y = mean(c(a.[2],b.[2])),
             a =3,
             b = 10,
             angle = 30
             )
```

**APPROXIMATE 2D distances**
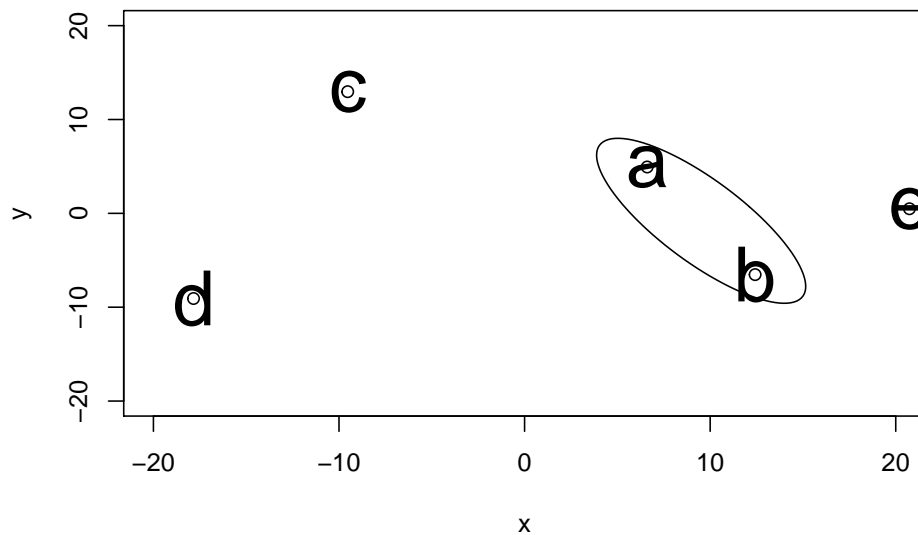


Once we have created a cluster (clade) we can consider the distance from the members of clade to all the remaining points. For example, here's the distnace from a and b to c

```
# distance from ab to c
c.<-dmat1.out$points["c",]

plot_dist_as_nmds(dmat1)
```

```
## $points
##          [,1]        [,2]
## a    6.606653  4.9400937
## b   12.421702 -6.5267890
## c   -9.532025 12.9665271
```

```
## d -17.839642 -9.0739515
## e  20.738283  0.4897976
##
## $eig
## NULL
##
## $x
## NULL
##
## $ac
## [1] -7.934712e-15
##
## $GOF
## [1] 0.6755273 0.9367305
```

```
segments(x0 = a.[1],x1 = c.[1],
         y0 = a.[2],y1 = c.[2])
segments(x0 = b.[1],x1 = c.[1],
         y0 = b.[2],y1 = c.[2])
```

**APPROXIMATE 2D distances**
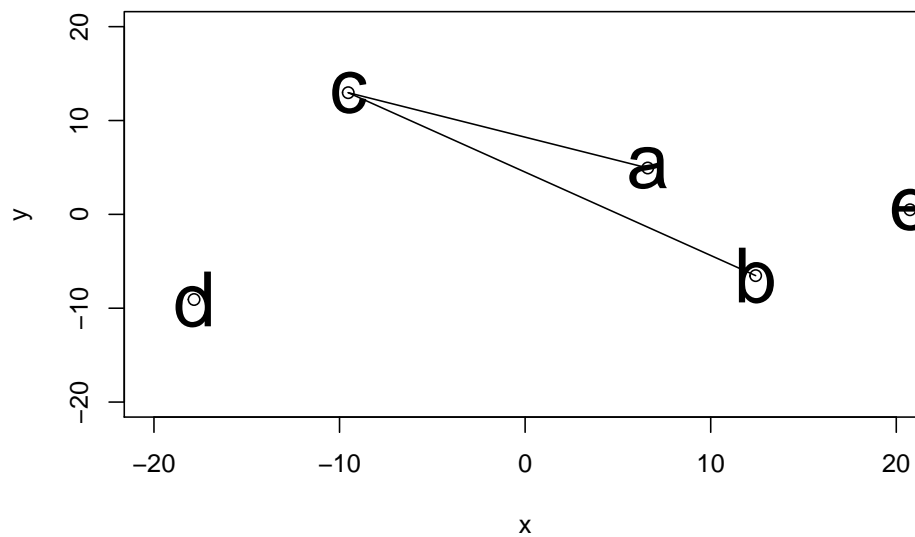


In UPGMA, we first define a clade. We then need to see which taxa is closest to that clade. Visually it looks like this

```
par(mfrow = c(2,2), mar = c(2,2,2,2))

# ab to c
plot_dist_as_nmds(dmat1)
```

```
## $points
##          [,1]        [,2]
## a   6.606653  4.9400937
## b  12.421702 -6.5267890
## c  -9.532025 12.9665271
## d -17.839642 -9.0739515
## e  20.738283  0.4897976
##
## $eig
## NULL
##
## $x
## NULL
##
## $ac
## [1] -7.934712e-15
##
## $GOF
## [1] 0.6755273 0.9367305
```

```r
#get x y coord of a and b
a.<-dmat1.out$points["a",]
b.<-dmat1.out$points["b",]

draw.ellipse(x = mean(c(a.[1],b.[1])),
             y = mean(c(a.[2],b.[2])),
             a =3,
             b = 10,
             angle = 30)
c.<-d2$points["c",]
segments(x0 = a.[1],x1 = c.[1],
         y0 = a.[2],y1 = c.[2])
segments(x0 = b.[1],x1 = c.[1],
         y0 = b.[2],y1 = c.[2])

# ab to d
plot_dist_as_nmds(dmat1)
```

```
## $points
##          [,1]        [,2]
## a   6.606653  4.9400937
## b  12.421702 -6.5267890
## c  -9.532025 12.9665271
## d -17.839642 -9.0739515
## e  20.738283  0.4897976
##
```

```
## $eig
## NULL
##
## $x
## NULL
##
## $ac
## [1] -7.934712e-15
##
## $GOF
## [1] 0.6755273 0.9367305
```

```
draw.ellipse(x = mean(c(a.[1],b.[1])),
             y = mean(c(a.[2],b.[2])),
             a =3,
             b = 10,
             angle = 30)
d.<-d2$points["d",]
segments(x0 = a.[1],x1 = d.[1],
         y0 = a.[2],y1 = d.[2])
segments(x0 = b.[1],x1 = d.[1],
         y0 = b.[2],y1 = d.[2])

# ab to e
plot_dist_as_nmds(dmat1)
```

```
## $points
##         [,1]        [,2]
## a    6.606653   4.9400937
## b   12.421702  -6.5267890
## c   -9.532025  12.9665271
## d  -17.839642  -9.0739515
## e   20.738283   0.4897976
##
## $eig
## NULL
##
## $x
## NULL
##
## $ac
## [1] -7.934712e-15
##
## $GOF
## [1] 0.6755273 0.9367305
```

```
draw.ellipse(x = mean(c(a.[1],b.[1])),
             y = mean(c(a.[2],b.[2])),
             a =3,
             b = 10,
             angle = 30)
e.<-d2$points["e",]
segments(x0 = a.[1],x1 = e.[1],
         y0 = a.[2],y1 = e.[2])
segments(x0 = b.[1],x1 = e.[1],
         y0 = b.[2],y1 = e.[2])
```

**APPROXIMATE 2D distances**



**APPROXIMATE 2D distances**



**APPROXIMATE 2D distances**
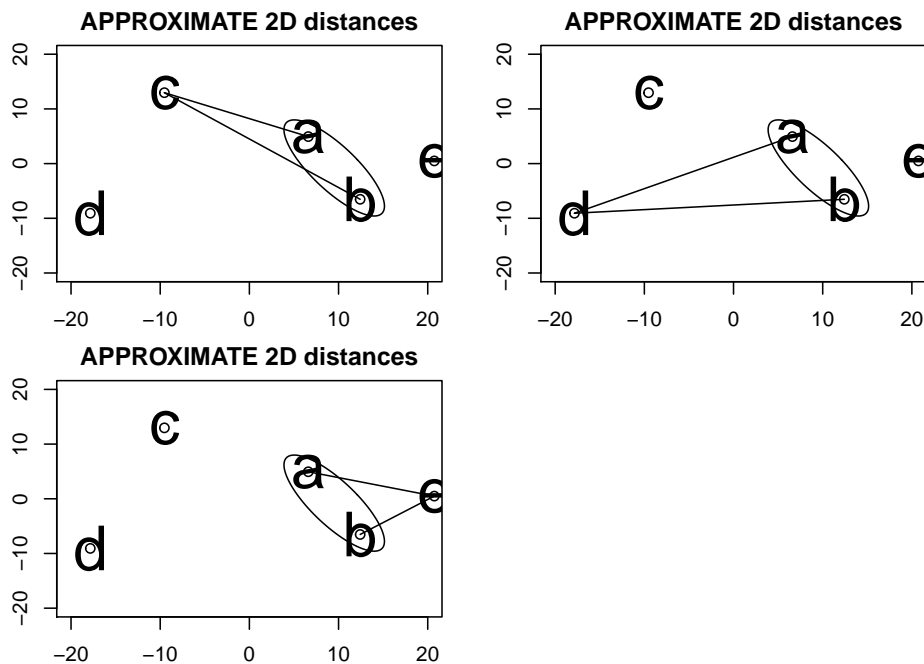


## 49.8   Algorithm - round 1

We will now implement the algorithm.

NOTE: I am not using a consistent system of nomeclature - I am mixing my own conventions with hose of Swofford et al. Sorry :(

First we'll need a dataframe to hold tings

- it = iteration
- clust.i, clust.j = the two taxa or clades being combined
- dist.ij = value from distnace matrix between the taxa
- clust.u = name of the new combined cluster (clade)
- branch.l.ij = branch lengths

A matrix to hold things as I work

```r
n.taxa <- nrow(dmat1)
UPGMA_output <- data.frame(it = c(1:n.taxa),
                           clust.i = NA,
                           clust.j = NA,
                           dist.ij = NA,
                           clust.u = NA,
                           branch.l.ij = NA)
```

## 49.8.1 Determine minimum value in current matrix

The minimum distance using the min() function

```r
dist.min.i <- min(dmat1, na.rm = T)
```

The index value of the minimum, using which()

```r
index.min.i <- which(dmat1 == dist.min.i, arr.ind =T)
index.row.i <- index.min.i[1]
index.col.i <- index.min.i[2]
```

We can get the names of our taxa using rownames() and colnames()

```r
cluster.i2 <- rownames(dmat1)[index.row.i]
cluster.i1 <- colnames(dmat1)[index.col.i]
```

Add output to dataframe

```r
# the clusters we're working iwth
UPGMA_output[1,"clust.i"] <- cluster.i1
UPGMA_output[1,"clust.j"] <- cluster.i2

# the distance between them
UPGMA_output[1,"dist.ij"] <- dist.min.i
```

## 49.8.2 Combine species into clade

Combine the two taxa names into a new name. This can be done a couple ways.

```r
# name of our clade

## hard-code "ab"
clade.i <- "ab"

## use paste() to generate name
clade.i <- paste("a","b",sep = "")

## general, re-usable code using paste() and object names
```

```r
clade.i <- paste(cluster.i1,cluster.i2, sep = "")
```

We'll add this to the dataframe

```r
UPGMA_output[1,"clust.u"] <- clade.i
```

Look at things so far

```r
UPGMA_output
```

```
##   it clust.i clust.j dist.ij clust.u branch.l.ij
## 1 1       a       b      17      ab          NA
## 2 2    <NA>    <NA>      NA    <NA>          NA
## 3 3    <NA>    <NA>      NA    <NA>          NA
## 4 4    <NA>    <NA>      NA    <NA>          NA
## 5 5    <NA>    <NA>      NA    <NA>          NA
```

### 49.8.3   Calculate branch length

Branch length is the distance between the two taxa: distance/2 or d.ij/2

We can access the distance in our first row of the matrix like this

```r
UPGMA_output[1,"dist.ij"]
```

```
## [1] 17
```

We can divided the distance betyween a and b by 2 to get our branch length
like this

```r
UPGMA_output[1,"dist.ij"]/2
```

```
## [1] 8.5
```

Now store the distance d in the dataframe

```r
UPGMA_output[1,"branch.l.ij"] <- UPGMA_output[1,"dist.ij"]/2
```

We've now completed our first round of calculations

```r
UPGMA_output
```

```
##   it clust.i clust.j dist.ij clust.u branch.l.ij
## 1 1       a       b      17      ab         8.5
## 2 2    <NA>    <NA>      NA    <NA>          NA
## 3 3    <NA>    <NA>      NA    <NA>          NA
## 4 4    <NA>    <NA>      NA    <NA>          NA
## 5 5    <NA>    <NA>      NA    <NA>          NA
```

### 49.8.4   Calculate distance from clade to all other points

Distance (ab) to all other points

- ab to c
- ab to d
- ab to e

### 49.8.4.1  ab to c

This is calculated at the average of the distance

- from a to c, and
- from b to c.

In the Wikipedia article the math is framed like this

D(ab to c) = [D(a to c) + D(b to c)] / 2 D(ab to c) = [ 21 + 30 ] / 2 D(ab to c) = [ 21 + 30 ] / 2 D(ab to c) = 25.5

Dab_c = [Da_c + Db_c] / 2 Dab_c = 25.5

The math in the wikipedia article is more accurately written out as this, which is more similar to Swofford's notation.

D(ab to c) = [T.i*D(a to c)* + T.j*D*(b to c)] / 2 D(ab to c) = [1*D(a to c)* + 1D(b to c)] / (1+1) D(ab to c) = [ 1*21* + 1*30* ] / (1+1) D(ab to c) = [ 1*21* + 1*30* ] / (1+1) D(ab to c) = 25.5

Dab_c = [1*Da_c* + 1Db_c] / (1+1) Dab_c = 25.5

Using Swofford et al's notation we'd do this:

d.ku = (Ti x d.ki + Tj x d.kj)/(Ti + Tj)

I'll just switch the subscripts to be consistent with the Wikipedia notation

d.uk = (Ti x d.ik + Tj x d.jk)/(Ti + Tj)

where

u = our new cluster ab k = the remaining clusters; we'll start with c Ti, Tj = the number of clusters the constitute the clusters that were just combined. This starts out at 1. Ti = Ta = 1 Tj = Tb = 1 d.ik = the distance from cluster i to the reamining clusters; we'll start with c d.ik = d.ac = 21 d.jk = d.bc = 30

d.ku = (Ti x d.ik + Tj x d.jk)/(Ti + Tj) d.ku = (1 x 21 + 1 x 30 )/( 1 + 1 ) d.ku = ( 21 + 30 )/(2) d.ku = 51/2 d.ku = 25.5

The Swofford notation is more generic and useful because in the next iteration we'll have a cluster of ab which will have two taxa in it.

```
Ta <- 1
Tb <- 1
Da_c <- 21
Db_c <- 30
Dab_c <- (Ta*Da_c+Tb*Db_c)/2
```

#### 49.8.4.2  ab to d

We now continue for the other distances. I'll stick to the wikipedia notation for now.

D(ab to d) = [D(a to d) + D(b to d)] / 2 D(ab to d) = [ 31 + 34 ] / 2 D(ab to d) = [ 31 + 34 ] / 2 D(ab to d) = 32.5

Dab_d = [Da_d + Db_d] / 2 Dab_d = 32.5

```
Da_d <- 31
Db_d <- 34
Dab_d <- (Da_d+Db_d)/2
```

#### 49.8.4.3  ab to e

D(ab to d) = [D(a to d) + D(b to d)] / 2 D(ab to d) = [ 23 + 21 ] / 2 D(ab to d) = [ 23 + 21 ] / 2 D(ab to d) = Dab_e

Dab_d = [Da_d + Db_d] / 2 Dab_d = Dab_e

```
Da_e <- 23
Db_e <- 21
Dab_e <- (Da_e+Db_e)/2
```

### 49.8.5  Update matrix with new distance

Old matrix has distances from a and b to all other points: we want to replace these. distances among c, d, and e are still the same values.

```
dmat1
```

```
##     a  b  c  d  e
## a NA NA NA NA NA
## b 17 NA NA NA NA
## c 21 30 NA NA NA
## d 31 34 28 NA NA
## e 23 21 39 43 NA
```

#### 49.8.5.1  Illustration

Values in parenthesese will be REPLACED

```
a <- c(NA, NA, NA, NA, NA)
b <- c("(17)", NA, NA, NA, NA)
c   <- c("(21)", "(30)", NA, NA, NA)
d   <- c("(31)", "(34)", "28", NA, NA)
e   <- c("(23)", "(21)", "39", "43",    NA)
dmat1.alt <- rbind(a,b,c,d,e)
```

```
colnames(dmat1.alt) <- c("a","b","c","d","e")
dmat1.alt
```

```
##   a      b      c    d     e
## a NA     NA     NA   NA    NA
## b "(17)" NA     NA   NA    NA
## c "(21)" "(30)" NA   NA    NA
## d "(31)" "(34)" "28" NA    NA
## e "(23)" "(21)" "39" "43"  NA
```

Create smaller matrix with clade ab

```
#        ab    c   d   e
ab  <- c(NA,    NA,  NA, NA)
c      <- c(Dab_c, NA,  NA, NA)
d      <- c(Dab_d, 28,     NA, NA)
e      <- c(Dab_e, 39,  43, NA)

dmat2 <- rbind(ab,c,d,e)
colnames(dmat2) <- c("ab","c","d","e")
```

### 49.8.5.2  Estimate 2 D

Estimate of 2D representation of the new matrix.

```
plot_dist_as_nmds(dmat2)
```

**APPROXIMATE 2D distances**
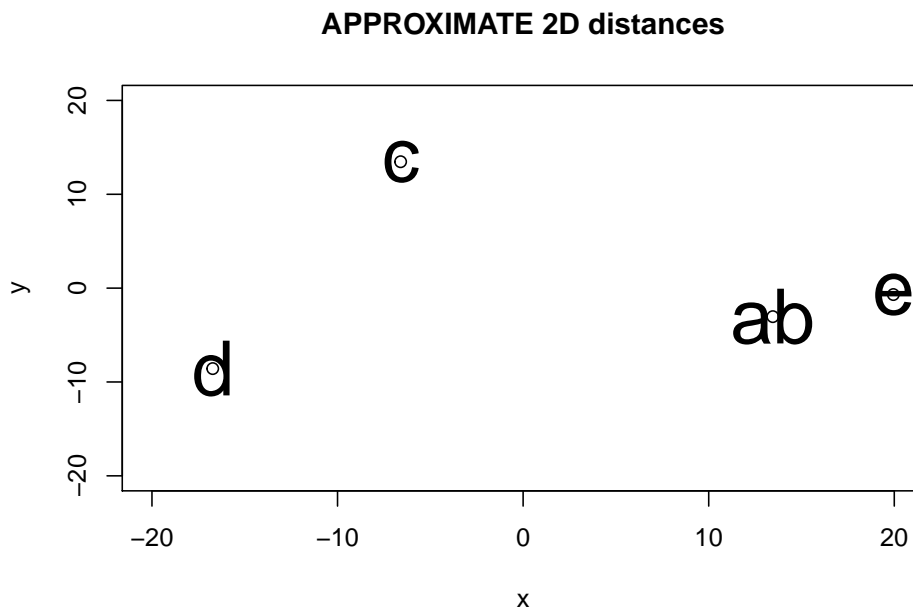


```
## $points
```

```
##             [,1]        [,2]
## ab  13.453890 -3.0373250
## c    -6.600828 13.4680501
## d   -16.726270 -8.5744114
## e    19.948871 -0.6844397
##
## $eig
## NULL
##
## $x
## NULL
##
## $ac
## [1] -4.570935e-15
##
## $GOF
## [1] 0.7573625 1.0000000
```

## 49.9   Algorithm - next round

We've combined a and b and calculated the disnace of this clade (a,b) to the remaining taxa. What is now the minimum distance?

The minimum distance of the current matrix

```
dist.min.i <- min(dmat2, na.rm = T)
```

The index value of the current minimum

```
index.min.i <- which(dmat2 == dist.min.i, arr.ind =T)
index.row.i <- index.min.i[1]
index.col.i <- index.min.i[2]
```

We can get the names of our taxa using the

```
cluster.i2 <- rownames(dmat2)[index.row.i]
cluster.i1 <- colnames(dmat2)[index.col.i]
```

Add output to dataframe

```
UPGMA_output[2,"clust.i"] <- cluster.i1
UPGMA_output[2,"clust.j"] <- cluster.i2
UPGMA_output[2,"dist.ij"] <- dist.min.i
```

### 49.9.1   Combine species into clade

Combine the two taxa names into a new name. This can be done a couple ways.

```
clade.i <- "abe"
clade.i <- paste("ab","e",sep = "")

clade.i <- paste(cluster.i1,cluster.i2, sep = "")
```

We'll add this to the dataframe

```
UPGMA_output[2,"clust.u"] <- clade.i
```

### 49.9.2 Calculate branch length

Branch length is distance/2 or d.ij/2 = d.ab.e = 22/2

```
UPGMA_output[2,"branch.l.ij"] <- UPGMA_output[2,"dist.ij"]/2
```

### 49.9.3 Recalculate distance from abe to all remainign taxa

In the current matrix, the smallest distance is between ab and e

We therefore want to form a clade between ab and e (abe), the measure the distance from this clade to allthe other species.

Visualize this as an approximate 2D situation

```
par(mfrow = c(1,1))
dmat2.out <- plot_dist_as_nmds(dmat2)

#get x y coord of ab and e
ab.<-dmat2.out$points["ab",]
e.<-dmat2.out$points["e",]

draw.ellipse(x = mean(c(ab.[1],e.[1])),
             y = mean(c(ab.[2],e.[2])),
             a =7,
             b = 7,
             angle = 0)
c.<-dmat2.out$points["c",]

segments(x0 = ab.[1],x1 = c.[1],
         y0 = ab.[2],y1 = c.[2])
segments(x0 = e.[1],x1 = c.[1],
         y0 = e.[2],y1 = c.[2])
```

**APPROXIMATE 2D distances**
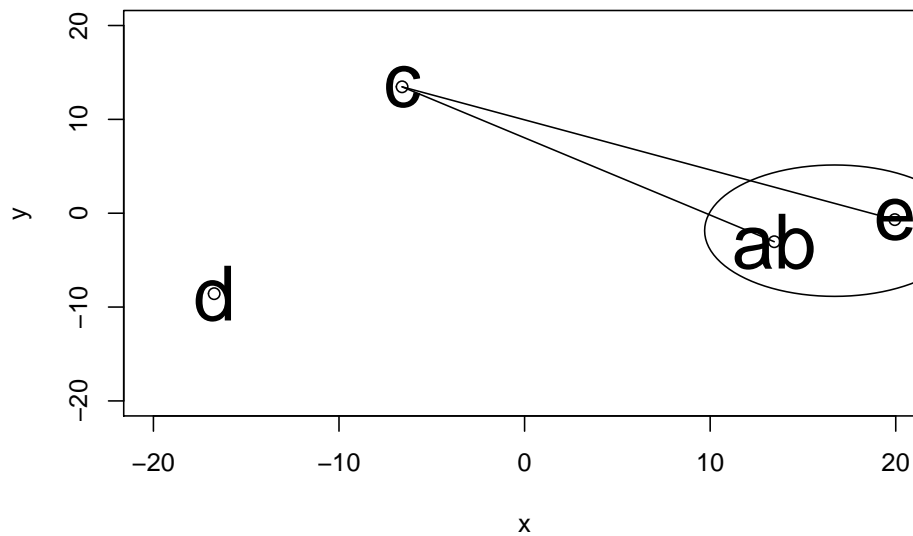


#### 49.9.3.1   abe to c

NOTE: calculations are "weighted" in proportion number of species in clade. So D(ab to c) is multiplied by two because ab is 2 species

The denominators is 3 because of the weights used in the numerator.

The notation in the Wikipedia article is:

D(abe to c) = [D(ab to c)*2 + D(e to c)*1] / (2+1) D(abe to c) = [ 25.5*2 + 39*1 ] / (2+1) D(abe to c) = [ 25.5*2 + 39*1 ] / (2+1) D(abe to c) = 32.5

Dabe_c = [Dab_c*2 + De_c*1] / (2+1) Dabe_c = 32.5

```
Dab_c <- 25.5
De_c  <- 39
Dabe_c <- (Dab_c*2 + De_c*1)/(2+1)
```

#### 49.9.3.2   abe to c

For abe to d

```
Dab_d <- 32.5
De_d  <- 43
Dabe_d <- (Dab_d*2 + De_d*1)/(2+1)
```

```
abe <- c(NA,     NA,  NA)
c      <- c(Dabe_c, NA,  NA)
d      <- c(Dabe_d, 28,  NA)


dmat3 <- rbind(abe,c,d)
colnames(dmat3) <- c("abe","c","d")
```

## 49.10  Next iteration

### 49.10.0.1  Estimate 2 D

a, b, e are now combined into a clade

```
par(mfrow = c(1,1))
dmat3.out <- plot_dist_as_nmds(dmat3)

#get x y coord of ab and e
c. <-dmat3.out$points["c",]
d.<-dmat3.out$points["d",]

draw.ellipse(x = mean(c(c.[1],d.[1])),
             y = mean(c(c.[2],d.[2])),
             a =14,
             b = 4,
             angle = 105)
abe.<-dmat3.out$points["abe",]

segments(x0 = c.[1],x1 = abe.[1],
         y0 = c.[2],y1 = abe.[2])
segments(x0 = d.[1],x1 = abe.[1],
         y0 = d.[2],y1 = abe.[2])
```

**APPROXIMATE 2D distances**
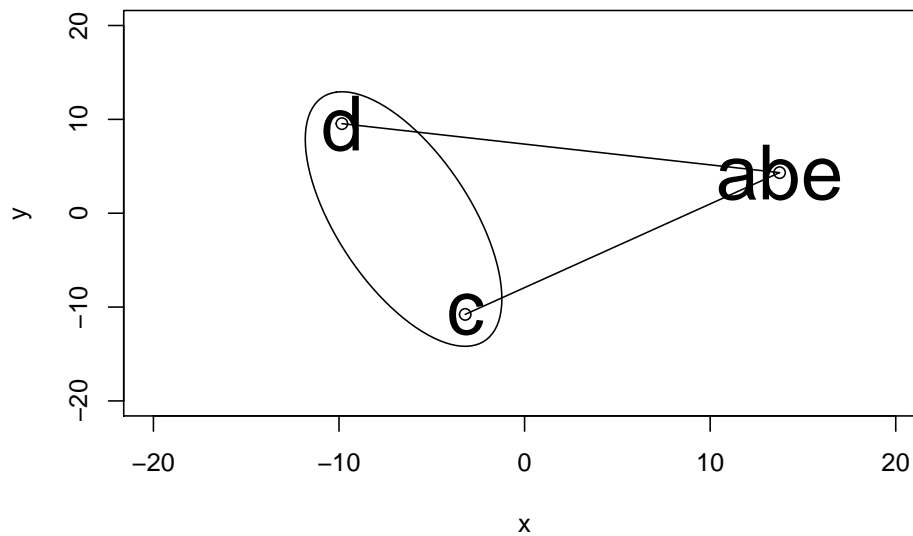


### 49.10.1   Calculate branch lengths

The minimum distance of the current matrix

```
dist.min.i <- min(dmat3, na.rm = T)
```

The index value of the current minimum

```
index.min.i <- which(dmat3 == dist.min.i, arr.ind =T)
index.row.i <- index.min.i[1]
index.col.i <- index.min.i[2]
```

We can get the names of our taxa using the

```
cluster.i2 <- rownames(dmat3)[index.row.i]
cluster.i1 <- colnames(dmat3)[index.col.i]
```

Add output to dataframe

```
UPGMA_output[3,"clust.i"] <- cluster.i1
UPGMA_output[3,"clust.j"] <- cluster.i2
UPGMA_output[3,"dist.ij"] <- dist.min.i
```

### 49.10.2   Combine species into clade

Combine the two taxa names into a new name. This can be done a couple ways.

```
clade.i <- "cd"
clade.i <- paste("c","d",sep = "")
```

```
clade.i <- paste(cluster.i1,cluster.i2, sep = "")
```

We'll add this to the dataframe

```
UPGMA_output[3,"clust.u"] <- clade.i
```

### 49.10.3   Calculate branch length

Branch length is distance/2 or d.ij/2 = d.d.e = 28/2

```
UPGMA_output[3,"branch.l.ij"] <- UPGMA_output[3,"dist.ij"]/2
```

### 49.10.4   Distance of c to abe and d to abe

```
Dc_abe <- 30
Dd_abe <- 36

# abe is composted of 3
Ddc_abe <- (Dc_abe*3 + Dd_abe*3)/(3+3)

# factor out 3
Ddc_abe <- 3*(Dc_abe*1 + Dd_abe*1)/6

# simplify
Ddc_abe <- 1*(Dc_abe*1 + Dd_abe*1)/2
Ddc_abe <- (Dc_abe*1 + Dd_abe)/2
```

Create matrix

```
abe <- c(NA,      NA)
cd   <- c(Ddc_abe, NA)



dmat4 <- rbind(abe,cd)
colnames(dmat4) <- c("abe","cd")
```

## 49.11   Finish up

The final entry of the matrix is 33. This means that the distnace from the clade abe to the clade de is 33, with branch length of $33/2 = 16.5$

The minimum distance

```
dist.min.i <- min(dmat4, na.rm = T)
```

The index value of the minimum

```
index.min.i <- which(dmat4 == dist.min.i, arr.ind =T)
index.row.i <- index.min.i[1]
index.col.i <- index.min.i[2]
```

We can get the names of our taxa using the

```
cluster.i2 <- rownames(dmat4)[index.row.i]
cluster.i1 <- colnames(dmat4)[index.col.i]
```

Add output to dataframe

```
UPGMA_output[4,"clust.i"] <- cluster.i1
UPGMA_output[4,"clust.j"] <- cluster.i2
UPGMA_output[4,"dist.ij"] <- dist.min.i
```

### 49.11.1   Combine species into clade

Combine the two taxa names into a new name. This can be done a couple ways.

```
clade.i <- paste(cluster.i1,cluster.i2, sep = "")
```

We'll add this to the dataframe

```
UPGMA_output[4,"clust.u"] <- clade.i
```

### 49.11.2   Calculate branch length

Branch length is distance/2 or d.ij/2

```
UPGMA_output[4,"branch.l.ij"] <- UPGMA_output[4,"dist.ij"]/2
```

## 49.12   Finalizing branch lengths

We now have a final tree structure. In Newick format it would be

```
full.tree <- "((c,d),(e,(b, a)));" # semi colon!
full.tree <-read.tree(text=full.tree)
plot(full.tree, main = "3-taxa tree")
```

# 3–taxa tree
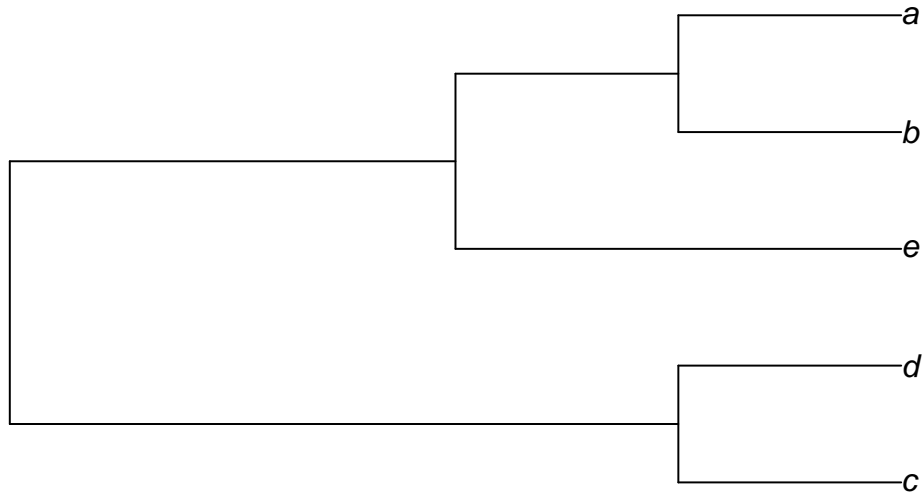


While we've done our clustering we've calcualte branch lengths and put them in the column branch.l.ij of the dataframe UPGMA_output

```
UPGMA_output
```

```
##   it clust.i clust.j dist.ij clust.u branch.l.ij
## 1  1       a       b      17      ab         8.5
## 2  2      ab       e      22     abe        11.0
## 3  3       c       d      28      cd        14.0
## 4  4     abe      cd      33   abecd        16.5
## 5  5    <NA>    <NA>      NA    <NA>          NA
```

The following code adds them to the tree.

First, the brnach lenghts from a to b

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
```
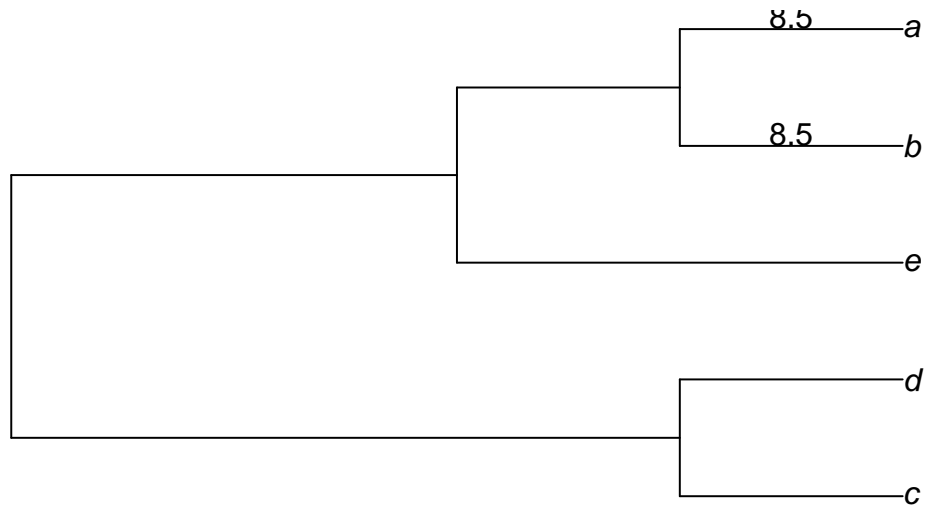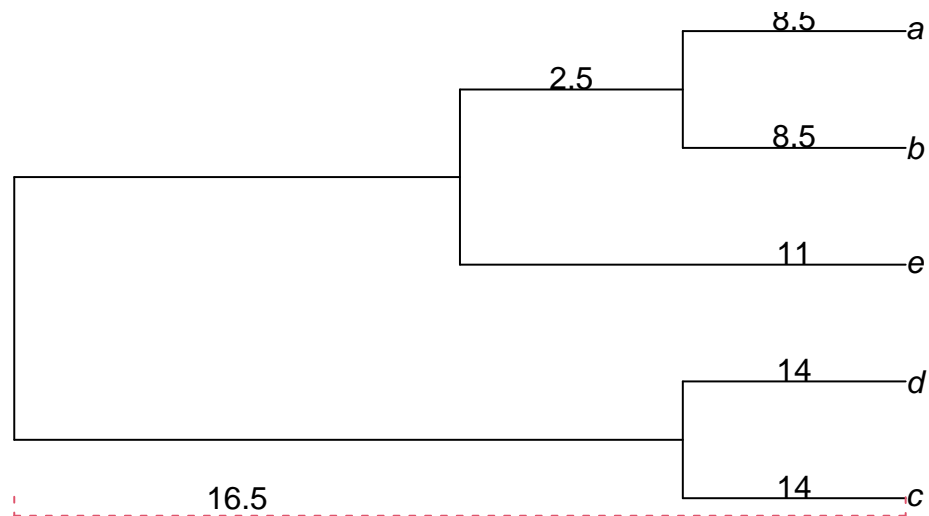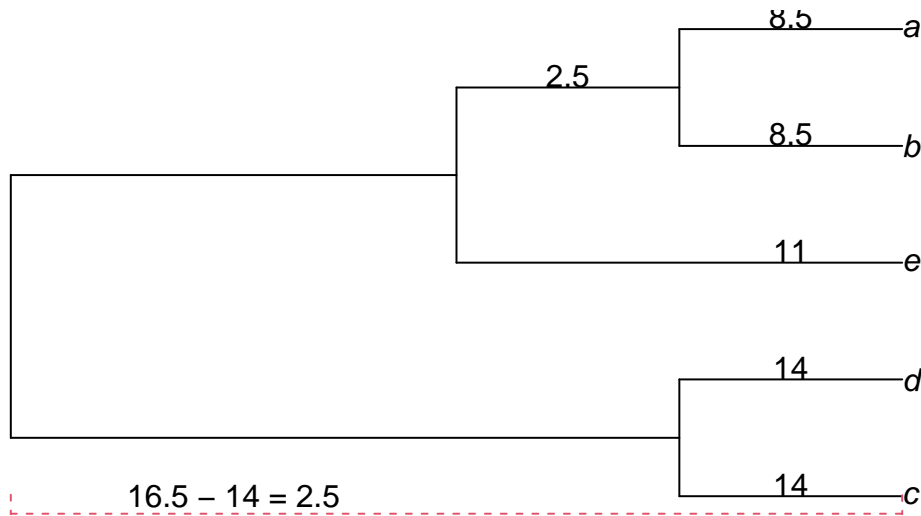
now a to b, and ab to e

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)
```
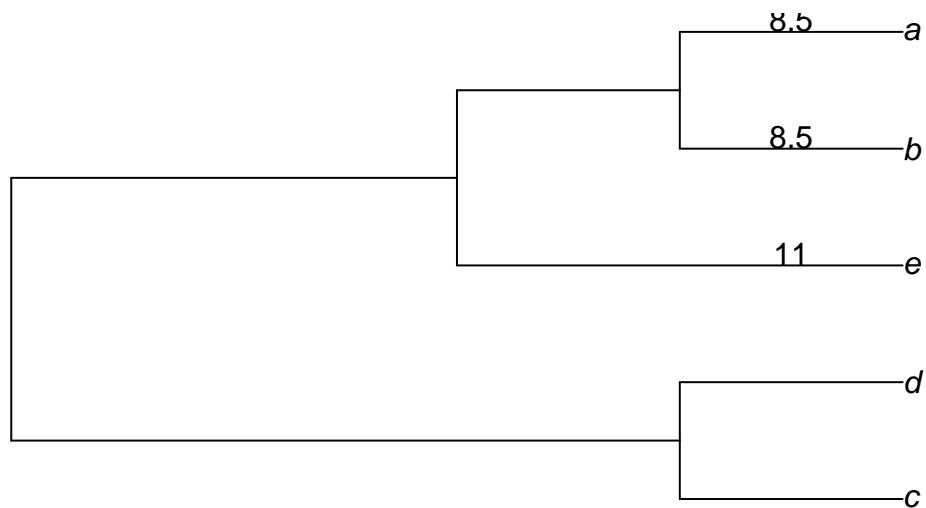


There's a branch between the clade ab and the node that connects with e.
What's its length? FOr a simple ultrametric tree calcualted with UPGMA
we can calculate this by subtraction

```
11-8.5
```

```
## [1] 2.5
```

We'll added this short branch of 2.5. Note this things ARE NOT to scale.

```r
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)
```



To visualize exactly what is going on here its useful to add this

```r
par(mar = c(1,1,1,1))
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 4.9,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

arrows(x0 = 2,x1 = 4,y0 =5.15,y1=5.15, code =3,angle = 90,length =0.1, lty =2, col = 2)
text(x = 2.5,y = 5,labels = "2.5+8.5=11")
```
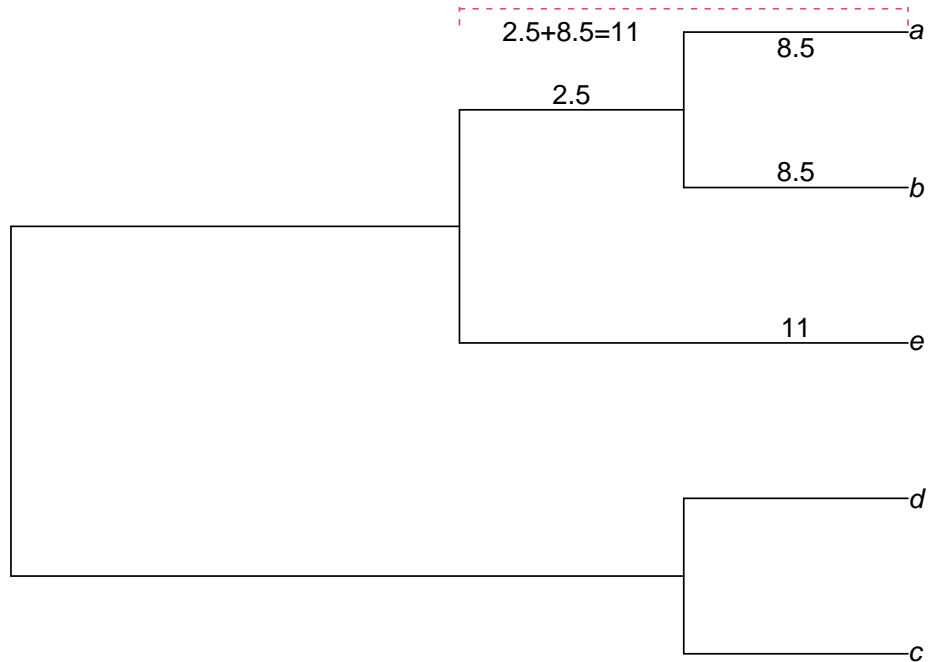
We can now add the d-e branch of 14 (note again - not to scale!)

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

# d-e branch
text(x = 3.5,y = 1.1,labels = 14)
text(x = 3.5,y = 2.1,labels = 14)
```
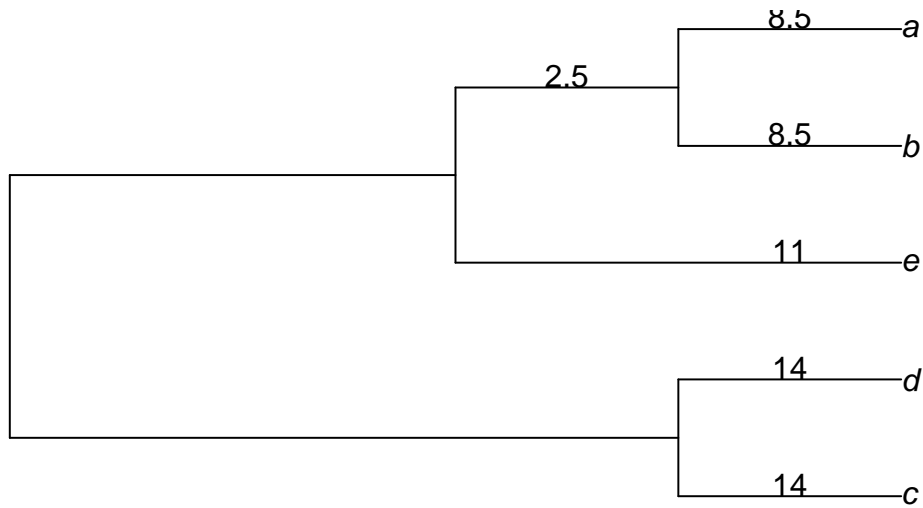
The distance from abe to de is 16.5 We can visualize this as

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

# d-e branch
text(x = 3.5,y = 1.1,labels = 14)
text(x = 3.5,y = 2.1,labels = 14)


arrows(x0 = 0,x1 = 4,y0 =0.85,y1=0.85, code =3,angle = 90,length =0.1, lty =2, col = 2)
text(x = 1,y = 1,labels = 16.5)
```

We can calcualte the distance of the remaining short branch from de to the root as 16.5-14 = 2.5. When we plot this again, it won't be to scale

```r
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

# d-e branch
text(x = 3.5,y = 1.1,labels = 14)
text(x = 3.5,y = 2.1,labels = 14)


arrows(x0 = 0,x1 = 4,y0 =0.85,y1=0.85, code =3,angle = 90,length =0.1, lty =2, col = 2)
text(x = 1,y = 1,labels = "16.5 - 14 = 2.5")
```

We can calculate the remaining short branch from abe to the root as 16.5-8.5-2.5 =16.5-11 =5.5

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

# d-e branch
text(x = 3.5,y = 1.1,labels = 14)
text(x = 3.5,y = 2.1,labels = 14)


arrows(x0 = 0,x1 = 4,y0 =0.85,y1=0.85, code =3,angle = 90,length =0.1, lty =2, col = 2)
text(x = 1,y = 1,labels = "16.5 - 14 = 2.5")

arrows(x0 = 0,x1 = 4,y0 =5.1,y1=5.1, code =3,angle = 90,length =0.1, lty =2, col = 2)
text(x = 1,y = 5,labels = "16.5 - 8.5 - 2.5 = 5.5")
```

16.5 − 8.5 − 2.5 = 5.5

8.5

a

2.5

8.5

b

11

e

14

d

16.5 − 14 = 2.5

14

c
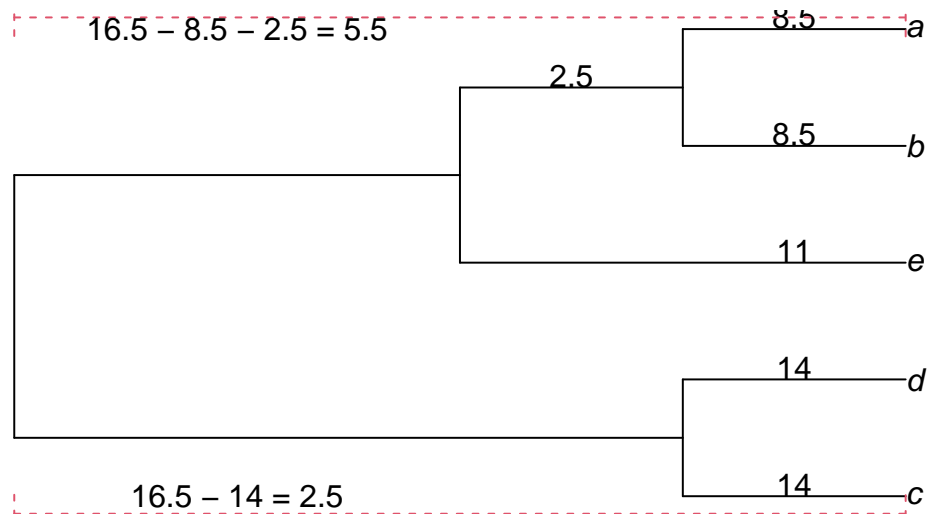
So the final tree has branch lengths (not to scale!) of

```
plot(full.tree, main = "")
# a-b branch lengths= 8.5
text(x = 3.5,y = 5.1,labels = 8.5)
text(x = 3.5,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 3.5,y = 3.1,labels = 11)

# short branch
text(x = 2.5,y = 4.6,labels = 2.5)

# d-e branch
text(x = 3.5,y = 1.1,labels = 14)
text(x = 3.5,y = 2.1,labels = 14)


text(x = 1,y = 1.6,labels = "2.5")


text(x = 1,y = 4,labels = "5.5")
```
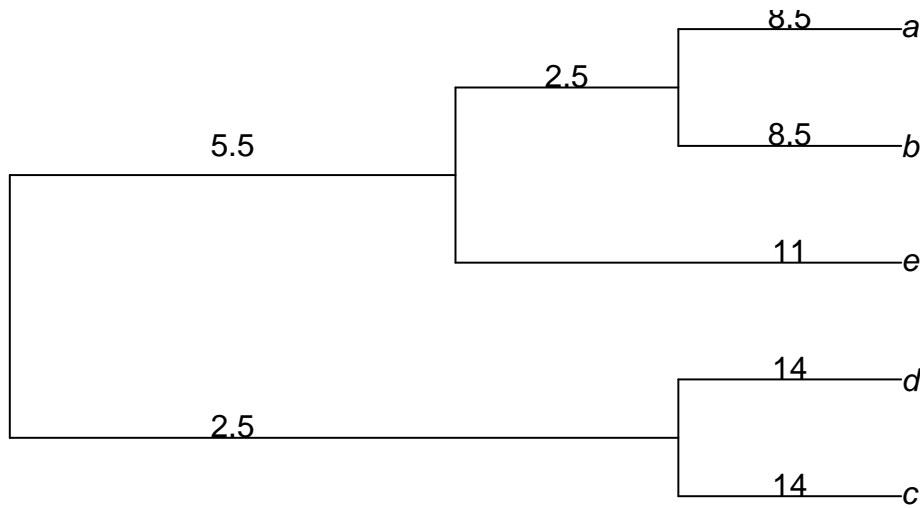
Most of the time we don't include branch lengths in Newick format, but there is a way to do it. You don't need to know how to do this - this is just for illustration

```
full.tree <- "((c:14,d:14):2.5,(e:11,(b:8.5, a:8.5):2.5):5.5);" # semi colon!
full.tree <-read.tree(text=full.tree)
plot(full.tree, main = "")

text(x = 10,y = 5.1,labels = 8.5)
text(x = 10,y = 4.1,labels = 8.5)
# ab-e branch lengths= 1
text(x = 10,y = 3.1,labels = 11)

# short branch
text(x = 7,y = 4.6,labels = 2.5)

# d-e branch
text(x = 10,y = 1.1,labels = 14)
text(x = 10,y = 2.1,labels = 14)



text(x = 1.5,y = 1.6,labels = "2.5")


text(x = 1.5,y = 3.9,labels = "5.5")
```
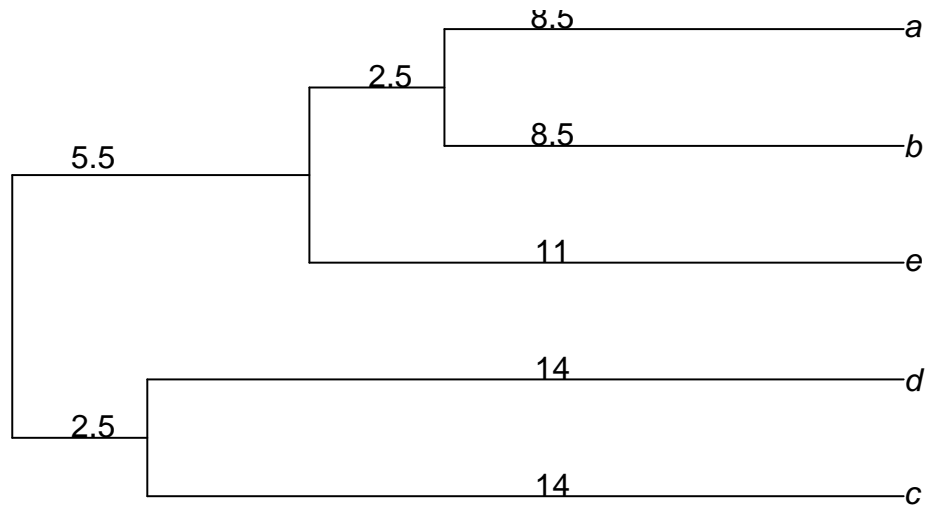
# Chapter 50

# Representing phylogenetic trees in Newick format

**By*: Nathan Brouwer

## 50.1   Vocab

- Newick notation
- topology
- clade

## 50.2   Introduction

**Newick format** is a common way to store information about **rooted phylogenetic trees** in a compact, written format. In its standard format, Newick does not store information about branch lengths, though it can be augmented to include lengths.

## 50.3   Trees with three taxa

A simple phylogenetic tree with three species can be represented in Newick as:

$$((Human, Chimp), Gorilla)$$

In Newick format, **clades** are enclosed in parentheses. So, in the example above, "(Human, Chimp)" means that humans and chimps are a **clade** and are each other's closest neighbors on the tree.

*Gorilla*
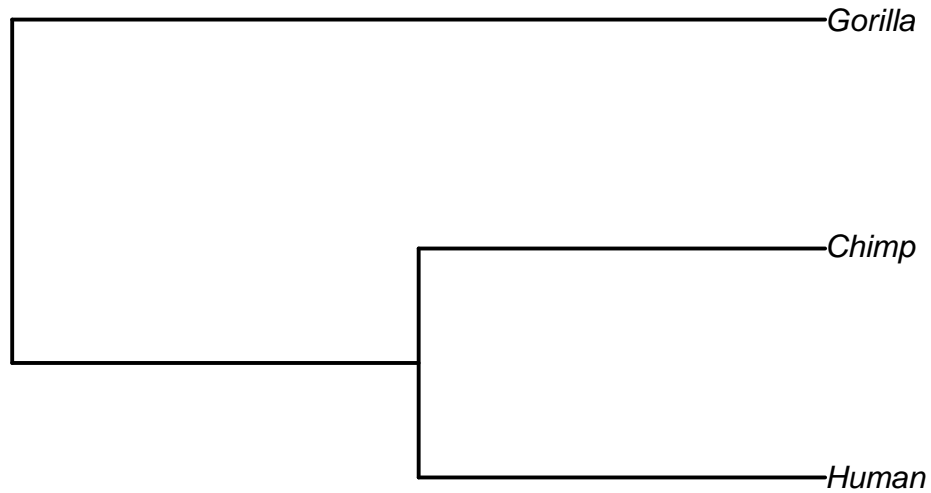
*Chimp*

*Human*

The general **mathematical rules of parentheses** apply to Newick format. Therefore, the *order* within the parentheses doesn't matter: "(Human, Chimp)" is the same as "(Chimp, Human)".

The two version of the trees result in differences in the order the taxa appear from top to bottom, but do not change its **toplogy** or interpretation.

**(Human, Chimp)**                    **(Chimp, Human)**

*Gorilla*                              *Gorilla*

*Chimp*                                *Human*
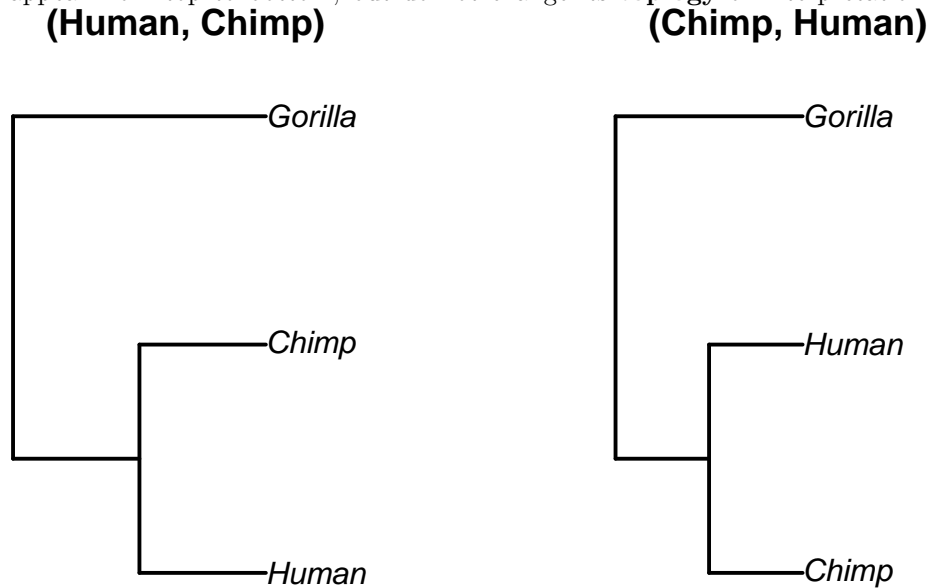
*Human*                                *Chimp*

Similarly, "((Human, Chimp), Gorilla)" is the same as "(Gorilla, (Human, Chimp))".

**((Human, Chimp), Gorilla)**    **(Gorilla, (Chimp, Human))**

*Gorilla*
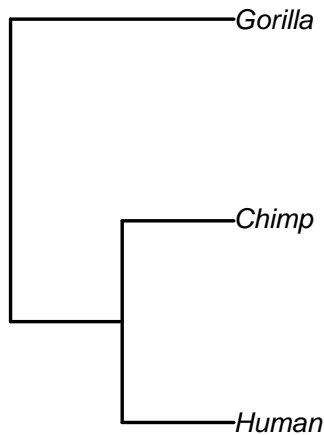
*Human*

*Chimp*

*Chimp*

*Human*
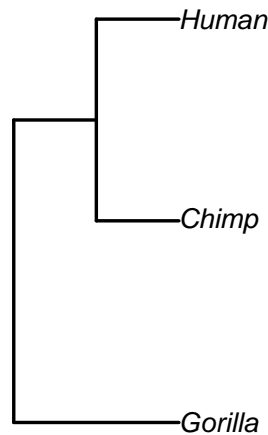
*Gorilla*

These rules for Newick are analogous to the rules for using parentheses in math: (1+2) is the same as (2+1), and ((2+1) * 2) is the same as (2*(2+1)).

## 50.4 Trees with four taxa

If we split Gorillas into the two major groups, western Gorillas in Gabon and eastern Gorillas in the Democratic Republic of Congo, we would put them in a clade: "(Gorilla-W, Gorilla-E)". We'll now have two separate clades of two taxa. We would then build up the full tree as

$$((Human, Chimp), (Gorilla - W, Gorilla - E))$$

This has two clades of two taxa each

1. Humans-Chimps

**((Chimp, Human), (Gorilla–W, Gorilla**



2. Western gorillas-Eastern Gorillas
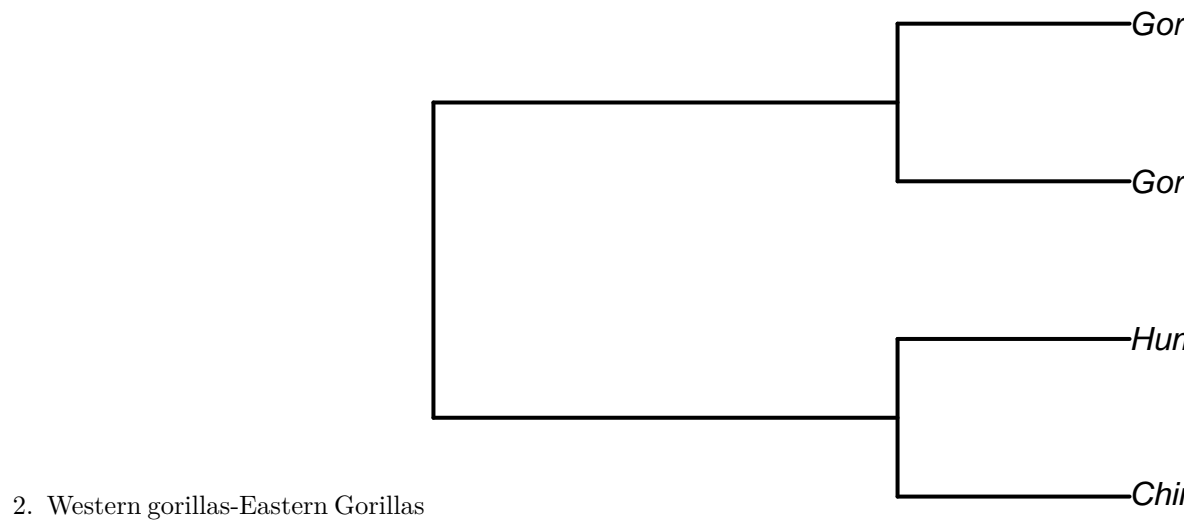
If wanted to represent a tree with humans, chimps, gorillas and orangutan we would represent it like this:

$$(((Human, Chimp), Gorilla), Orangutan)$$

The parenthesis to the right of Gorilla indicates that Gorilla falls within the human-chimp clade.

# (((Human, Chimp),Gorilla), Orangutan)
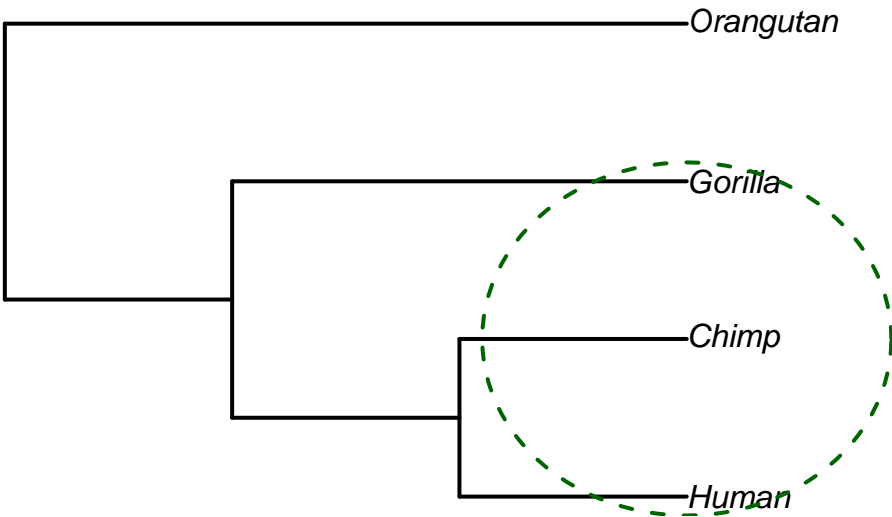


Just as in math, the number of parentheses needs to balance out. In this case, there are three parentheses open to the right, all next to human. There therefore needs to be 3 parentheses open to the right.

```
# 321           1         2            3
# (((Human, Chimp),Gorilla), Orangutan)
```

# Chapter 51

# Plotting trees from Newick notation in R

In R, a phylogenetic tree in Newick format can be input as a character string and plotted using the ape package.

## 51.1 Preliminaries

```
library(ape)
library(phangorn)
```

## 51.2 Note

Throughout this lesson I use the `par()` command to set up plots. You can ignore this command because its pretty tricky to use.

## 51.3 3 taxa tree in Newick

We input the tree as quoted text; note the semi colon at the end!

```
# two taxa in clade
"(H,C)"
```

```
## [1] "(H,C)"
```

```
# third taxa
"( (H,C) , G )"
```

```
## [1] "( (H,C) , G )"
```

```
# full string
str1 <- "((H, C), G );" # semi colon!
```

We then convert the tree using `read.tree()`

```
tree1 <-ape::read.tree(text=str1)
```

We plot with `plot()`

```
plot(tree1, main = "3-taxa tree")
```

## 3–taxa tree
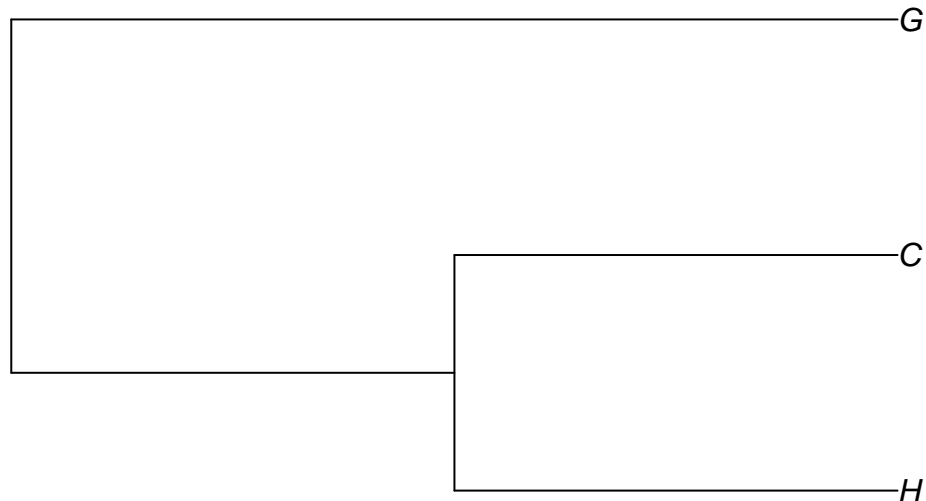


We can change the orientation of the tree by changing the order within the parentheses. For example, if we want Gorillas to appear at the bottom of the graph

```
str2<- "(G,(H, C));"            # semi colon!
tree2 <-read.tree(text=str2)
plot(tree2, main = "3-taxa tree")
```

# **3–taxa tree**
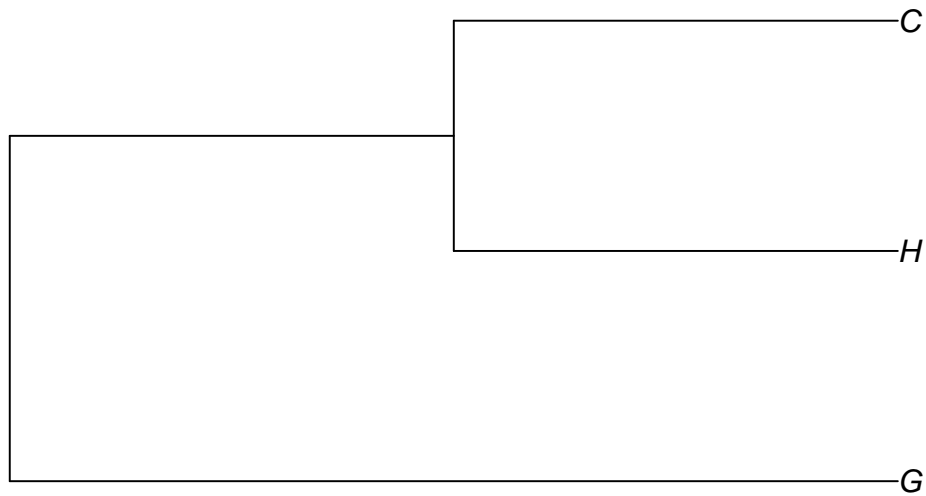


Perhaps we want humans at the top

```
str3<- "(G,(C, H));"              # semi colon!
tree3 <-read.tree(text=str3)
plot(tree3, main = "3-taxa tree")
```

# **3–taxa tree**
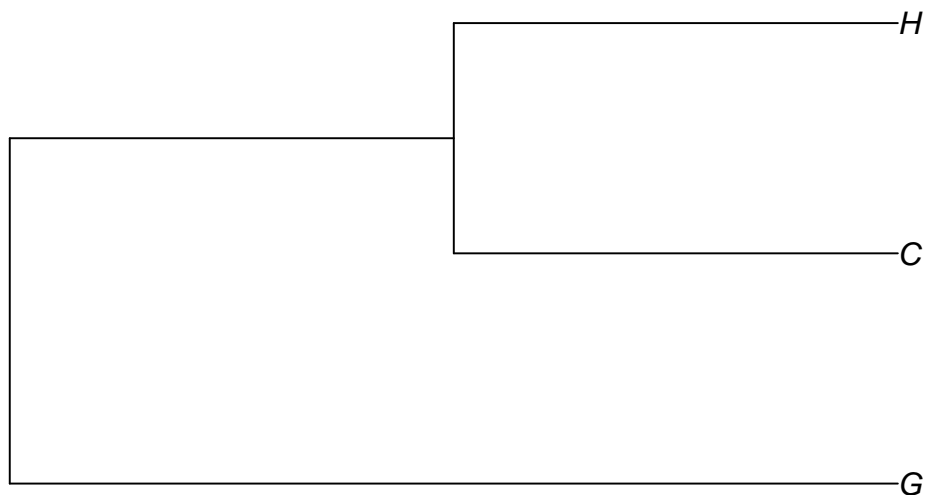


Note that all of these trees have the same **branching order** and therefore the exact same **topology**.

**Challenge:** Try to add the forth possible format of the tree that has the same topology.

TODO doesnt render

```
par(mfrow = c(2,2))
plot(tree1, main = "3-taxa tree 1")
plot(tree2, main = "3-taxa tree 2")
plot(tree3, main = "3-taxa tree 3")
```

## 51.4   4 taxa tree in Newick

When there are four taxa, there are two main topologies. The first is a simple **ladder branching** typology. This is represented like this

```
str4<- "(((Human, Chimp),Gorilla), Orangutan);"
tree4 <-read.tree(text=str4)

par(mfrow = c(1,1), mar = c(2,2,2,2))
plot(tree4, main = "4-taxa ladder tree")
```
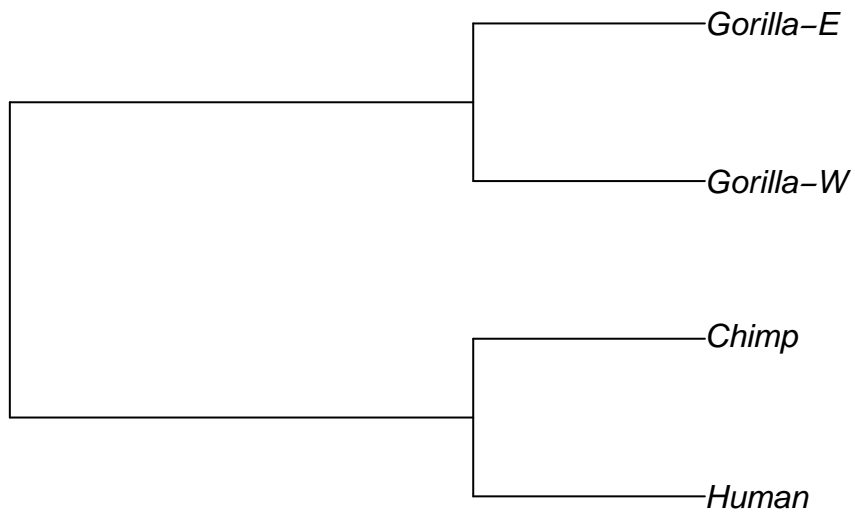
### 4–taxa ladder tree
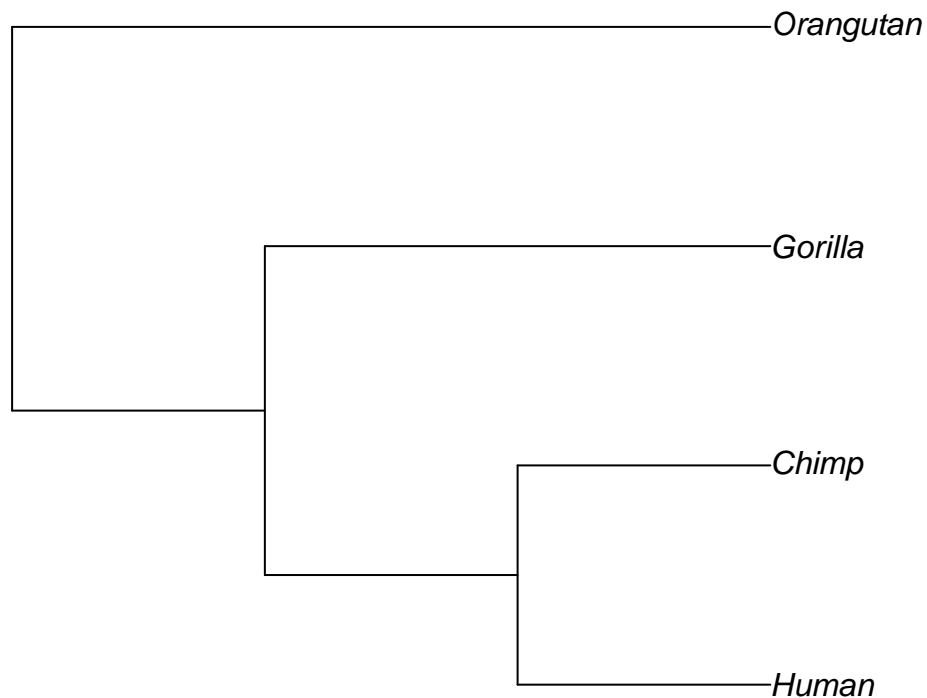


The second major topology is when there are *two* major clades, each with two

taxa

```
str5<- "((Human, Chimp), (Gorilla-W, Gorilla-E));"
tree5 <-read.tree(text=str5)
plot(tree5, main = "4-taxa, 2 clade tree")
```

**4–taxa, 2 clade tree**

*Gorilla–E*

*Gorilla–W*

*Chimp*

*Human*

Within each of the topologies, the grouping of species can be switched depending on what the evidence suggests. For example, some researchers have proposed the humans are more closely related to Orangutans. Their hypothesis has a **ladder branching** typology with the order of the species switched.

```
str6<- "(((Human, Orangutan),Chimp), Gorilla);"
tree6 <-read.tree(text=str6)
plot(tree6, main = "")
```
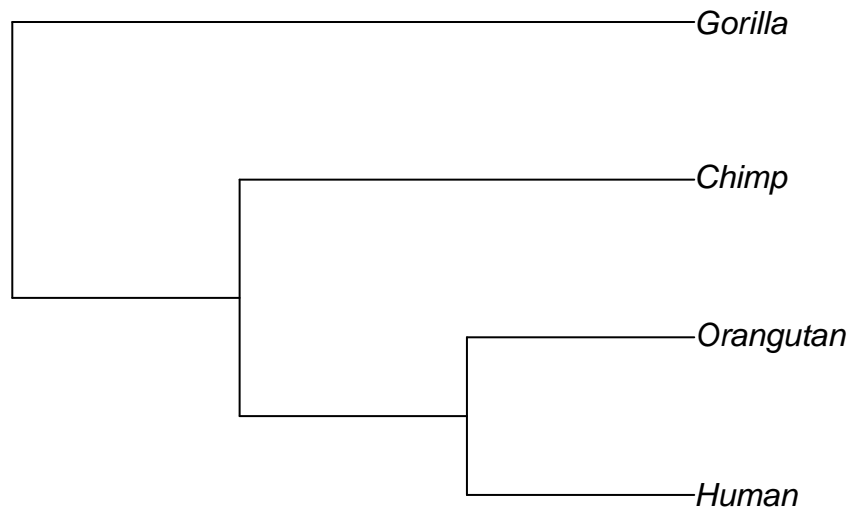
## 51.5  5 taxa in Newick Format

Five taxa can occur in a number of general typologies. The most basic is a continuous ladder. After Orangutans, humans next-most distantly related "cousin" are the Gibbons.
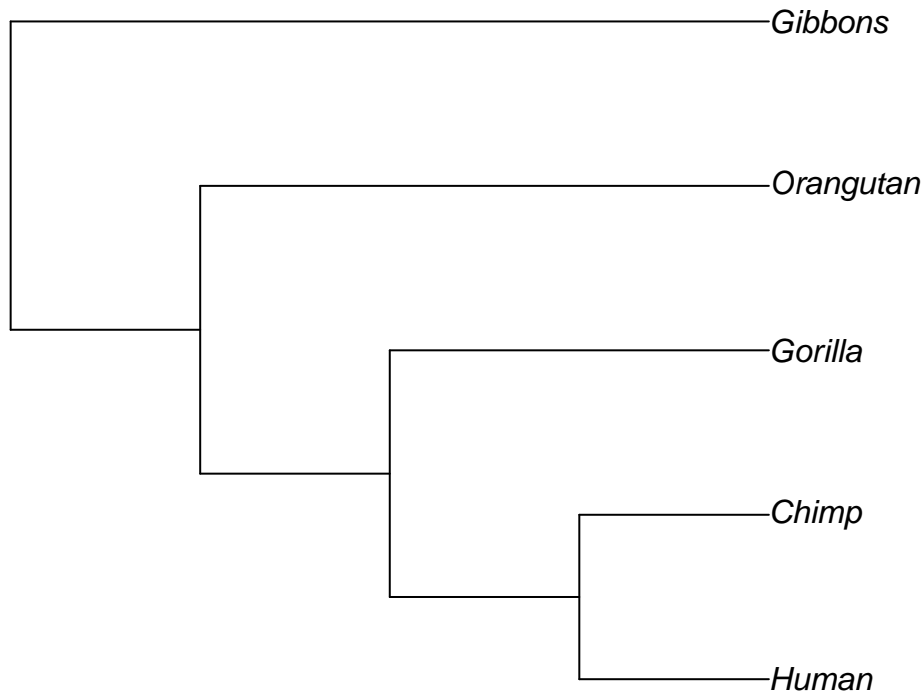
```
par(mfrow = c(1,1), mar = c(2,2,2,2))
str7<- "((((Human, Chimp),Gorilla), Orangutan),Gibbons);"
tree7 <-read.tree(text=str7)
plot(tree7, main = "5-taxa ladder tree")
```
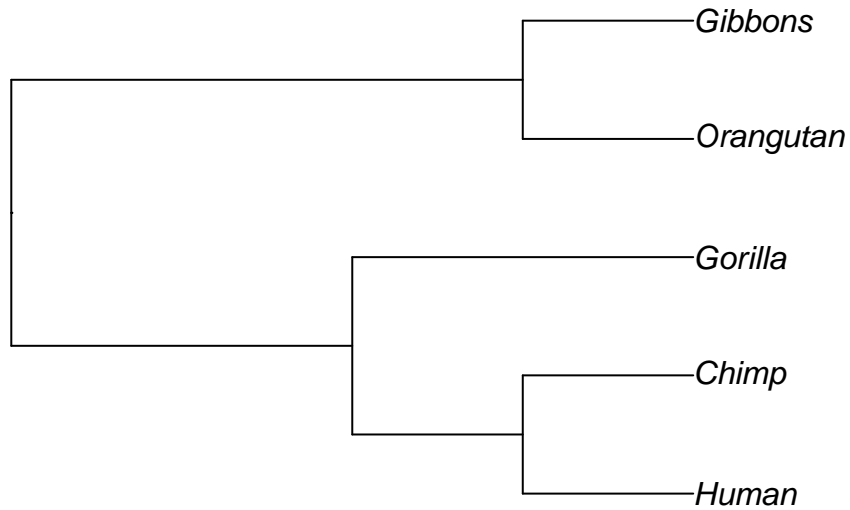
# 5–taxa ladder tree



Let's say there's a hypothesis that Gibbons and Orangutans form a clade (there's no actual evidence of this). This typology would create a tree with two main clades, one with 3 species and one with two.

```
str8 <- "((((Human, Chimp),Gorilla), (Orangutan,Gibbons)));"
tree8 <-read.tree(text=str8)
plot(tree8, main = "5-taxa, 2 big clades tree")
```

## 5–taxa, 2 big clades tree



A third major typology occurs if there are four taxa in a clade on a branch. Let's say there's a hypothesis that Orangutans and Gorillas are sister species (they aren't), and Gibbons are there more distantly related cousins.

```
str9 <- "(((Human, Chimp), (Orangutan, Gorilla)), Gibbons);"
tree9 <-read.tree(text=str9)
plot(tree9, main = "5-taxa, with 4-taxa clade ")
```

## 5–taxa, with 4–taxa clade
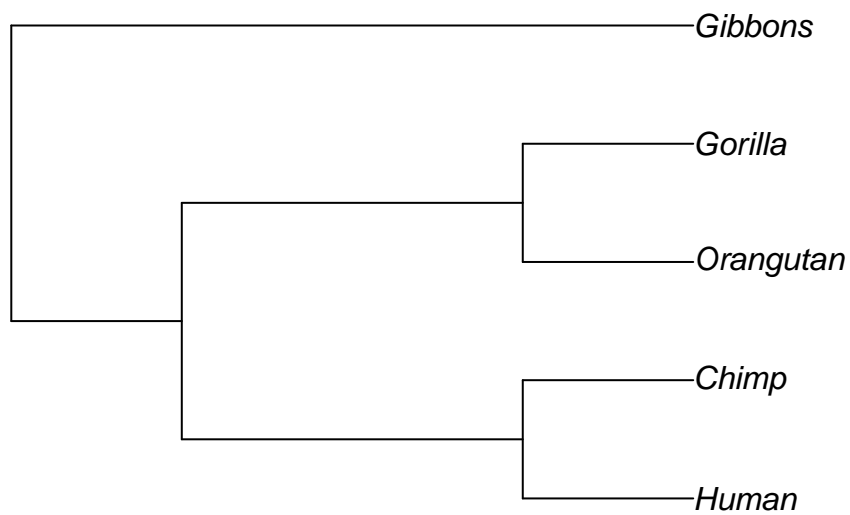
Here's another example of this format, dropping Gibbons and adding the two Gorilla subspecies (unlike many of the previous examples, this one is true).

```
str10<- "(((Human, Chimp),(Gorilla-W, Gorilla-E)), Orangutan);"
tree10 <-read.tree(text=str10)
plot(tree9, main = "")
```
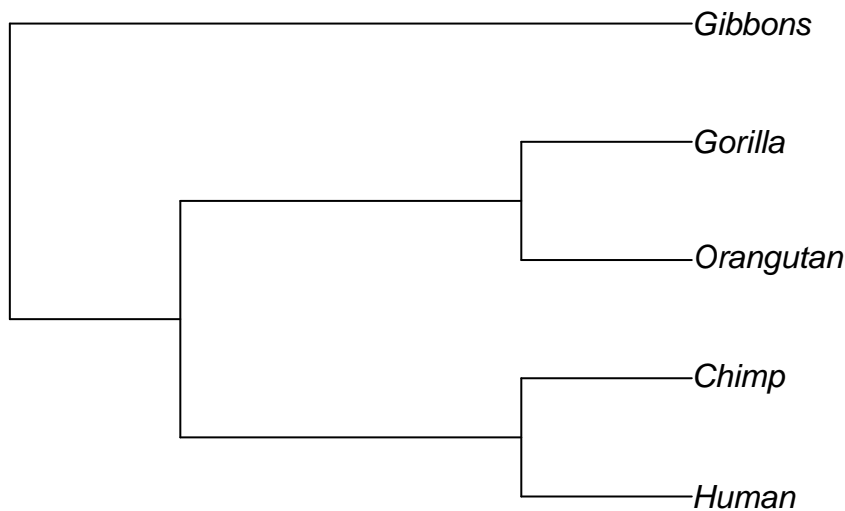


## 51.6  Many-taxa tree

Newick format can be used to represent any kind of tree, though in practice this can be very difficult to write out by hand. Here is an example of a number of vertebrates

```
par(mfrow = c(1,1), mar = c(2,2,2,2))
str.vert<-    "(((((((cow, pig),whale),(bat,(lemur,human))),(robin,iguana)),coelacanth),gold_fish
vert.tree<-read.tree(text=str.vert)
plot(vert.tree,no.margin=TRUE,edge.width=2)
```
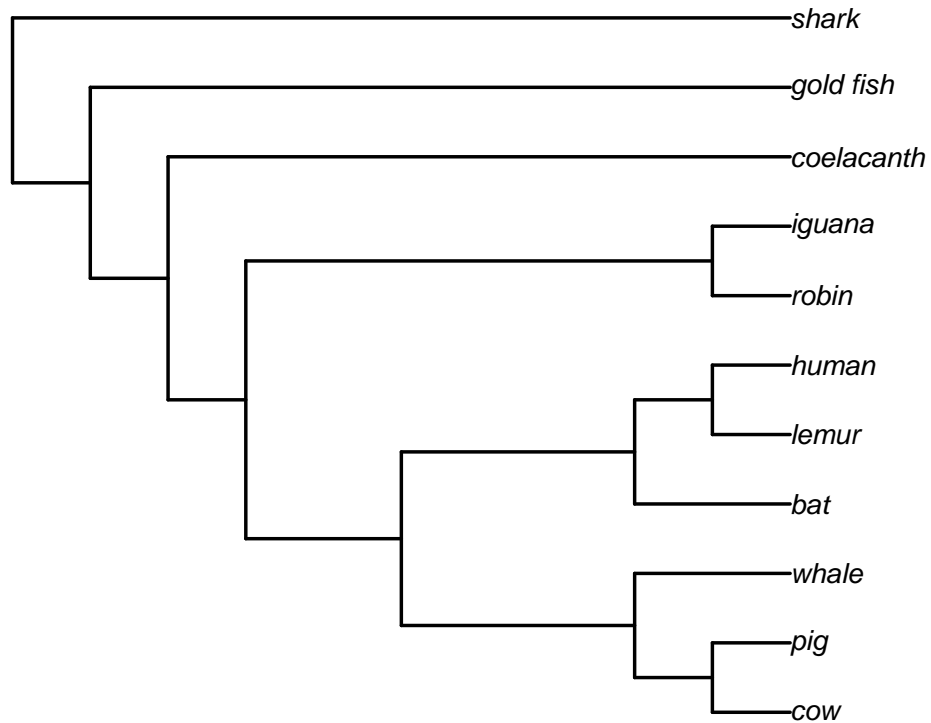
## 51.7 Newick and Branch lengths

Below is an example of a BIG tree which also has branch lengths represented. The original analysis was in Miyamoto and Goodman (1986) "Biomolecular Systematics of Eutherian Mammals: Phylogenetic Patterns and Classification."

```
str.vert<-  "(Edentata:55, (((Orycteropus:12, Trichechus:43):1, (Procavia:29, (Elephas
vert.tree<-read.tree(text=str.vert)
plot(vert.tree,no.margin=TRUE,edge.width=2)
```

```
## Warning in plot.phylo(vert.tree, no.margin = TRUE, edge.width = 2): 1 branch
## length(s) NA(s): branch lengths ignored in the plot
```

## 51.8 Additional Reading

The Wikipedia article on Newick format shows how it can be extended for more complex scenarios, including annotation of branch lengths. https://en.wikiped ia.org/wiki/Newick_format

Newick format is briefly mentioned on page 47 of Baum & Smith Tree Thinking.

# Chapter 52

# Phylogenetic trees using the UPGMA clustering algoirthm

**By**: Nathan Brouwer

## 52.1 Introduction

**UPGMA** is a method for constructing **phylogenetic trees** using **genetic distances**. It is an old, outmoded method that is rarely ever used any more for building final versions of trees. Instead of UPGMA, the **neighbor-joining** (NJ) algorithm is used when a distance-based phylogeny is made. UPGMA's relative simplicity, however, makes it a useful starting point for thinking about how to construct phylogenetic trees and code up algorithms to build them. UPGMA has also been used in the past in statistics, machine learning, transcriptomics, and community ecology as a general **clustering algorithm** and so is a useful starting point when understanding current algorithms in those fields.

## 52.2 Preliminaries

```r
# Potentially new packages
## install.packages("phangorn")
## install.packages("ape")

library(ape)
library(phangorn)
```

### 52.2.1   Vocab

- matrix, matrices
- diagonal of matrix
- symmetrical matrix
- lower triangle of matrix
- Newick format
- character string
- function, argument
- branch, edge, clade

### 52.2.2   General R functions

- `c()`
- `matrix()`
- colnames, rownames
- cbind, rbind
- class
- as.dist()
- par
- min
- str

### 52.2.3   Specific functions

- phangorn::phangorn()
- ape::read.tree
- ape::as.DNAbin
- ape::dist.dna()

## 52.3   Sequence comparisons

Let's consider three short, hypothetical DNA sequences. We'll put each into a **vector** using `c(...)`:

```r
human   <- c("a","t","c","g","a","t","c","g")
chimp   <- c("a","t","c","a","a","t","c","a")
gorilla <- c("a","a","a","a","a","a","a","a")
```

The sequences are each 8 bases long.

```r
length(gorilla)
```

```
## [1] 8
```

This means we'll have 8 **loci**, or unique **homologous positions**, to consider.

We can examine each pair of sequences and count up the number of differences between each one. This gives us a preliminary estimate of how **diverged** the sequences are.

Humans and chimps have 2 differences, at the 4th and 8th loci

```
rbind(human, chimp)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## human "a"  "t"  "c"  "g"  "a"  "t"  "c"  "g"
## chimp "a"  "t"  "c"  "a"  "a"  "t"  "c"  "a"
```

Chimps and gorillas have 4 differences (2nd, 3rd, 6th, and 7th loci):

```
rbind(chimp, gorilla)
```

```
##         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## chimp   "a"  "t"  "c"  "a"  "a"  "t"  "c"  "a"
## gorilla "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"
```

Humans and gorillas have 6 differences (all but 1st and 6th different)

```
rbind(human, gorilla)
```

```
##         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## human   "a"  "t"  "c"  "g"  "a"  "t"  "c"  "g"
## gorilla "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"
```

## 52.4 Distance matrices

Having counted up the number **pairwise differences** between each pair of sequences, we can represent this data as a genetic **distance matrix** like this.

First, make the matrix using the `matrix()` command.

```r
# make the matrix
dist_mat <- matrix(data = c(NA, NA, NA,
                            2, NA,NA,
                            6, 4, NA),
                   nrow = 3,
                   byrow = T)
```

Now, label the rows and columns.

```r
# label the matrix
colnames(dist_mat) <- c("H","C","G")
rownames(dist_mat) <- c("H","C","G")
```

The final matrix looks like this

```r
# look at the output
dist_mat
```

```
##    H  C  G
## H NA NA NA
## C  2 NA NA
## G  6  4 NA
```

**ASIDE: Building matrices**:

Alternatively, we could make each row of the matrix into a vector and then stack them with `rbind()`.

```
# make vectors
human.row <- c(NA,NA,NA)
chimp.row <- c( 2,NA,NA)
gorilla.row <- c(6,4,NA)

# stack vectors by row using rbind()
dist_mat <- rbind(human.row,
                  chimp.row,
                  gorilla.row)

# add names
spp.names <- c("human","chimp","gorilla")
colnames(dist_mat) <- spp.names
rownames(dist_mat) <- spp.names

# look at output
dist_mat
```

```
##          human chimp gorilla
## human       NA    NA      NA
## chimp        2    NA      NA
## gorilla      6     4      NA
```

**End ASIDE**

Note that we leave the values on the **diagonal** of the matrix as `NA` because the genetic distance between humans and humans, or chimps and chimps, is 0 and isn't helpful. A matrix like this is also **symmetrical**, where the **lower triangle** portion is the same as the upper portion We could build an **upper triangular matrix** like the one below if we wanted, but the convention is to use the lower.

```
# An upper triangular matrix
## make data in vectors
human.row <- c(NA,2,6)
chimp.row <- c( NA,NA,4)
gorilla.row <- c(NA,NA,NA)

# stack with ribind()
dist_mat_upper <- rbind(human.row,
```

```
                         chimp.row,
                         gorilla.row)

# add names
colnames(dist_mat_upper) <- spp.names
rownames(dist_mat_upper) <- spp.names
```

Compare lower triangular matrix form of these data to the upper triangular for:

```
# lower triangular matrix:
dist_mat
```

```
##         human chimp gorilla
## human     NA    NA      NA
## chimp      2    NA      NA
## gorilla    6     4      NA
# upper triangular matrix
dist_mat_upper
```

```
##         human chimp gorilla
## human     NA     2       6
## chimp     NA    NA       4
## gorilla   NA    NA      NA
```

In both matrices the distance from humans to gorillas is 6 differences, humans to chimps is 2, and chimps to gorillas is 4. Make sure you understand why this is true.

Again, the lower triangular form is the form usually used. Because `NA` occurs in the entire top row and last column, people often omit it by dropping the row and column of all NAs.

```
dist_mat[-1, -3]
```

```
##         human chimp
## chimp      2    NA
## gorilla    6     4
```

## 52.5   From a distance matrix to phylogenetic tree

These calculations imply that humans and chimps are more closely related (the sequences are made up, but this is true). Using data like this, [**phylogenetic algorithms** therefore cluster humans and chimps into a **clade**. In **Newick Format** this would be (H,C).
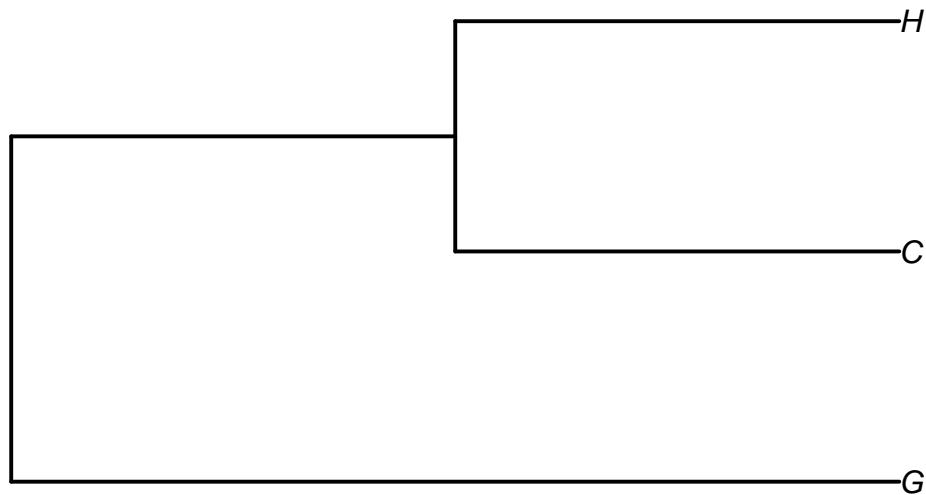
Since we have just three taxa, we add G outside the first clade: ((H,C),G).

We can plot a tree from Newick like this

```
# define tree
## NOTE: the string ends with a semi-colon!
str1<- c("(G,(C, H) );")

## convert character string to tree
tree1<-ape::read.tree(text=str1)

## plot tree
plot.phylo(tree1,edge.width=2, main = "")
```



Newick format does not represent **genetic distances** and therefore the **branch length** of phylogenetic, only the clade structure and branching pattern of the tree. We can calculate branch lengths using the UPGMA algorithm.

## 52.6 Distance matrix in R from raw sequences

To make a distance matrix in R from the original sequences we first need to put our original sequences into a matrix with the `rbind()` command.

```
my_seqs <- rbind(human,
                 chimp,
                 gorilla)
```

This is what it looks like

```
my_seqs

##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## human    "a"  "t"  "c"  "g"  "a"  "t"  "c"  "g"
## chimp    "a"  "t"  "c"  "a"  "a"  "t"  "c"  "a"
```

```
## gorilla "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"
```

We then need to do some fancy processing (this has to do with how the packages we're going to use works, which is picky about formats because its optimized to accommodate real sequences which can be very very long). We'll use the `as.DNAbin()` function.

```
my_seqs_bin <- ape::as.DNAbin(my_seqs)
```

We then can calculate the **distance** between each sequence in terms of the number (N) of mutations. This uses the `dist.dna()` function. We set the argument `model = "N"`. This is the most basic and naive model of mutation and ignores the possibility of back mutations.

```
seq_dist0 <- ape::dist.dna(my_seqs_bin, model = "N")
```

This gives us a distance matrix

```
seq_dist0
```

```
##         human chimp
## chimp      2
## gorilla    6     4
```

Note that there is a subtle difference in how R looks at the matrix we made earlier with the matrix() command and this matrix.

Them matrices have the same data, but slightly different formats.

```
dist_mat
```

```
##         human chimp gorilla
## human      NA    NA      NA
## chimp       2    NA      NA
## gorilla     6     4      NA
```

```
seq_dist0
```

```
##         human chimp
## chimp      2
## gorilla    6     4
```

They also each have a different class

```
class(dist_mat)
```

```
## [1] "matrix" "array"
```

```
class(seq_dist0)
```

```
## [1] "dist"
```

## 52.7   Distance matrix in R from normal matrix

Sometimes we already have a distance matrix computed that we want to enter directly into R. For example, perhaps it was reported in a paper and we want to explore the matrix on our own. In this case we can convert our normal matrix we made initially into a properly formatted distance matrix. This requires the base R `as.dist()` function. I'll call this object "seq_dist_from_mat" to indicate that it came "from a matrix". Note that I'm going to give it the full 3 x 3 matrix, `dist_mat`.

```
# input matrix
dist_mat
```

```
##         human chimp gorilla
## human      NA    NA      NA
## chimp       2    NA      NA
## gorilla     6     4      NA
```

```
# covert to distance matrix
seq_dist_from_mat <- as.dist(dist_mat)

# new format
seq_dist_from_mat
```

```
##         human chimp
## chimp       2
## gorilla     6     4
```

I'll now compare the two ways of making the matrix: `seq_dist0`, which was build from the original sequences, and `seq_dist_from_mat`, which was built from the hand-entered matrix.

First, let's compare their structure

```
seq_dist0
```

```
##         human chimp
## chimp       2
## gorilla     6     4
```

```
seq_dist_from_mat
```

```
##         human chimp
## chimp       2
## gorilla     6     4
```

Then check them out with is() and class()

```
is(seq_dist0) == is(seq_dist_from_mat)
```

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

```
class(seq_dist0) == class(seq_dist_from_mat)
```

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

Now I'll confirm that that have the same content (Note that distance matrices are not indexed internally by rows and columns, but by the order that the values appear. The following code therefore returns only 3 responses, not 4.)

```
seq_dist0 == seq_dist_from_mat
```

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

Cool, so we can make a matrix either from the raw sequences or from a hand-entered matrix.
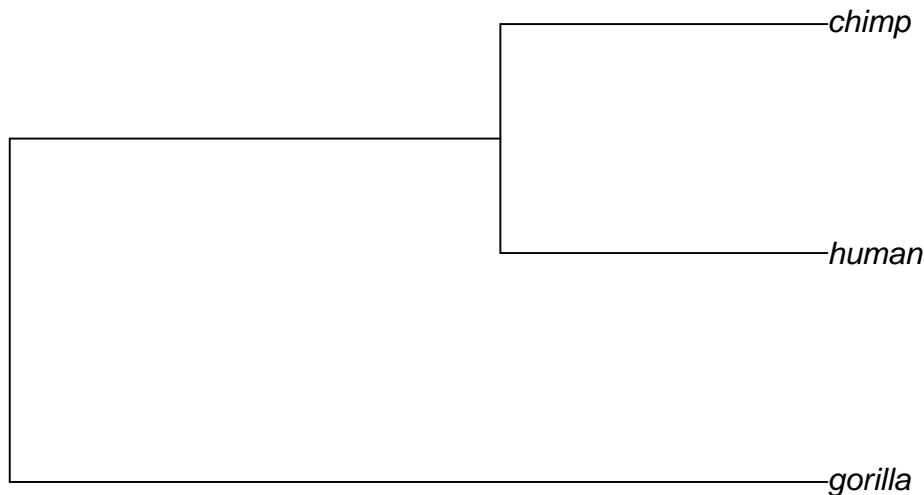
## 52.8   UPGMA in R

Once you have a distance matrix, UPGMA is easy in R with the `upgma()` function

```
seq_upgma0 <- upgma(seq_dist0)
```

Now plot it with `plot()`

```
plot(seq_upgma0, main = "Original data")
```

**Original data**



This doesn't look too different than the phylogeny produced from Newick. Let's play with the distance matrix to see what happens.

First, make some more copies of the distance matrix

```r
seq.dist1 <- seq_dist0
seq.dist2 <- seq_dist0
seq.dist3 <- seq_dist0
```

The relationship between humans and chimps in in the upper left hand cell of the matrix. Internally, $R$ is calling this the first cell of the matrix

```r
seq.dist1[1]
```

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

Let's increase and decrease this value

```r
seq.dist1[1] <- 20    #change from 2 to 20
seq.dist2[1] <- 0.5   #change from 2 to 0.5
seq.dist3[1] <- 0.05  #change from 2 to 0.05
```

Now we'll build trees with our original matrix and the news ones to compare them.

Note: We'll make a grid of plots using some fancy-looking code `par(mfrow = c(2,2), mar = c(2,2,2,2))`. Don't worry about what exactly this means; it just sets up some graphical parameters to make things look nice. I've also added some code to make the plot of the original data look a little different

```r
# set up graphical parameters
par(mfrow = c(2,2), mar = c(2,2,2,2))

# build 1st tree and plot
## build tree
seq_upgma0 <- upgma(seq_dist0)

## plot, in upper left-hand corner
plot.phylo(seq_upgma0,
           main = "Original data",
           edge.color = "darkgreen",
           edge.width = 4)

# build 2nd tree and plot
seq.upgma1 <- upgma(seq.dist1)
plot(seq.upgma1, main = "Very long distance humans vs. chimps!")

seq.upgma2 <- upgma(seq.dist2)
plot(seq.upgma2, main = "Shorter distance humans vs. chimps")


seq.upgma3 <- upgma(seq.dist3)
plot(seq.upgma3, main = "Much shorter distance humans vs. chimps!")
```
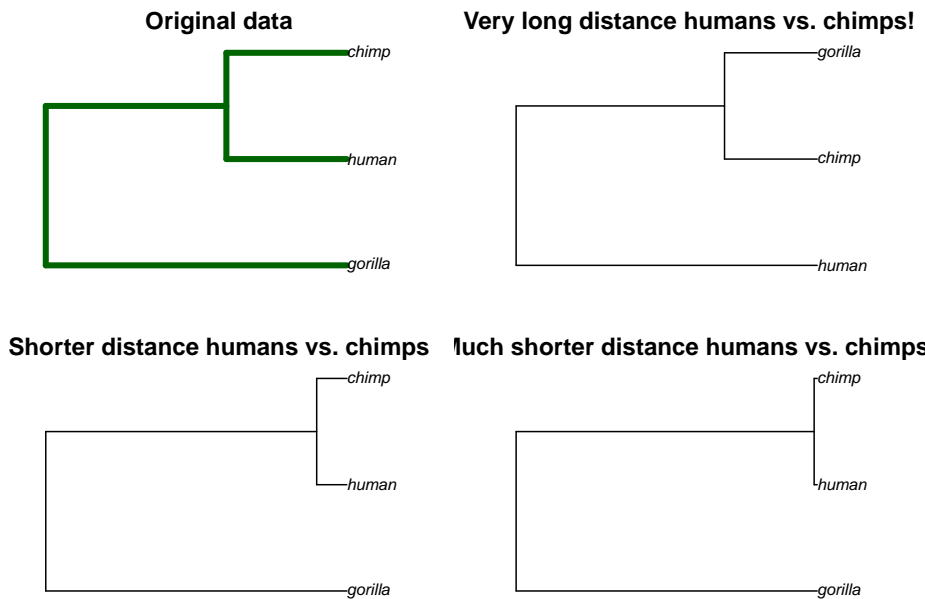
**Original data**

**Very long distance humans vs. chimps!**

**Shorter distance humans vs. chimps**  **Much shorter distance humans vs. chimps**

## 52.9 How UPGMA clustering works

UPGMA and other **clustering algorithms** work by finding looking at a distance matrix and finding the two things that are closet together.

Human and chimps are closest

```
seq_dist0
```

```
##        human chimp
## chimp     2
## gorilla   6     4
```

UPGAM therefore extracts the this minimum distance. We'll extract it by hand using the `min()` function.

```
d.min <- min(seq_dist0)
```

When two individual taxa are grouped into a clade the branch lengths are set at 1/2 the distance between them. So, the total distance is 2, so the branch lengths will be 1.

```
d.min/2
```

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

We can confirm that this is correct by checking what R calculated via the `upgma()` function. We can access *R's* calculations by looking at the object we saved the UPGMA output too

```
seq_upgma0
```

```
##
## Phylogenetic tree with 3 tips and 2 internal nodes.
##
## Tip labels:
##   human, chimp, gorilla
##
## Rooted; includes branch lengths.
```

On its own this isn't helpful. But what we can do is use the `str()` command to look under the hood of the object

```
str(seq_upgma0)
```

```
## List of 4
##  $ edge       : int [1:4, 1:2] 5 5 4 4 1 2 3 5
##  $ edge.length: num [1:4] 1 1 2.5 1.5
##  $ tip.label  : chr [1:3] "human" "chimp" "gorilla"
##  $ Nnode      : int 2
##  - attr(*, "class")= chr "phylo"
##  - attr(*, "order")= chr "postorder"
```

The output tells us there is something called `edge.length` associated with the `seq_upgma0` object, and the smallest two values are 1. This looks promising, since **edge** is the math term for branches in a tree.

Next, UPGMA builds a new matrix that replace the Human and Chimp columns with a combined Humans-chimp column. The algorithm then re-calculates the distance from this human-chimp clade to the remaining gorilla group The distance from this human-chimp group to gorillas is the average distance from humans to gorillas and chimps to gorillas.

The original distance from humans to gorillas was 6, and the distance from chimps to gorillas was 4, as shown by the distance matrix

```
seq_dist0
```

```
##         human chimp
## chimp      2
## gorilla    6     4
```

Let's store these values in R objects.

```
#distance humans to gorillas
Dh_g <- 6
```

```
#distance chimps to gorillas
Dc_g <- 4
```

Now, we'll take the average of these two distance. The math written out in explicit detail is:

```
Dhc_g <- (Dh_g*1 + Dc_g*1)/(1+1)
Dhc_g
```

```
## [1] 5
```

The result is total distance between our new human-chimp clade and gorillas.

### 52.9.1 Challenge

This new distance allows us to calculate the branch lengths for the rest of the tree. The branch length from the human-chimp (h-c) clade to the gorillas (g) is 1/2 half the distance we just calculated:

Dhc_g/2 = 5/2 = 2.5

The remaining branch length is 2.5 - 1 = 1.5.

See if you can figure out why this is done by sketching out the tree.

These values match what is in the upgma object

```
str(seq_upgma0)
```

```
## List of 4
##  $ edge       : int [1:4, 1:2] 5 5 4 4 1 2 3 5
##  $ edge.length: num [1:4] 1 1 2.5 1.5
##  $ tip.label  : chr [1:3] "human" "chimp" "gorilla"
##  $ Nnode      : int 2
##  - attr(*, "class")= chr "phylo"
##  - attr(*, "order")= chr "postorder"
```

## 52.10 Further information

### 52.10.1 General info

A good, no-math overview of the general ideas of UPGMA is at this blog post

Wikipedia provides a good overview with math that isn't too daunting (though it could be improved). https://en.wikipedia.org/wiki/UPGMA

### 52.10.2 Worked examples:

An excellent worked example is at this site. http://www.slimsuite.unsw.edu.au/teaching/upgma/ This example breaks down the math into basic steps, but there's a lot of taxa so its still complicated.

A worked example showing the limits of UPGMA is here: https://www.icp.ucl.ac.be/~opperd/private/upgma.html

Another, dense-looking example is here: http://www.nmsr.org/upgma.htm

Another worked example is https://www.mun.ca/biology/scarr/2900_UPGMA.htm

### 52.10.3 Additional notes:

Some useful but terse notes are here: https://www.sequentix.de/gelquest/help/upgma_method.htm

If you want to understand what the **ultrametricity** assumption is, check out this site (however, I don't get hung up on this): https://www.sequentix.de/gelquest/help/explanation_of_the_term_ultrametric.htm

Explanation of the convoluted meaning of "Un-weighted" is here https://www.mun.ca/biology/scarr/UPGMA_vs_WPGMA.html

# Chapter 53

# Tutorial: Writing functions to calculate the number of phylogenetic trees

**By** Nathan Brouwer

## 53.1   Repliminaries

### 53.1.1   Functions

- factorial()
- seq()
- print()
- c()
- if()
- function()
- args()

## 53.2   Concepts

#### 53.2.0.1   Biology Concepts:

- Number of possible phylogenetics trees

#### 53.2.0.2   Programming concepts

- vectorized inputs to functions
- conditional statements

- Role of arguments in writing function
- Role of { and } in writing functions
- function source code
- functions with defaults
- functions with 1 arguments
- functions with 2 arguments

### 53.2.1 Vocab:

#### 53.2.1.1 Programming vocab

- function
- function argument
- function default
- conditional statement

### 53.2.2 Packages:

None

## 53.3 Introduction

## 53.4 Number of rooted trees

The number of possible rooted phylogenetic trees is calculated using the equation below, where n is the number of taxa:

$\frac{(2*n-3)!}{2^{n-2}*(n-2)!}$

## 53.5 Number of rooted trees in R

In R we take factorials using the `factorial()` function

```
factorial(3)
```

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

```
3*2*1
```

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

```
factorial(4)
```

```
## [1] 24
```

```
4*3*2*1
```

```
## [1] 24
```

We can re-write our text equation above as: factorial(2n-3)/(2^(n-2))*(factorial(n-2))

In R would be for n = 3

```r
n<-3
factorial(2*n-3)/((2^(n-2))*factorial(n-2))
```

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

For n = 4

```r
n<-4
factorial(2*n-3)/((2^(n-2))*factorial(n-2))
```

```
## [1] 15
```

All of the parentheses make this a bit nutty. Let me write this out as a separate numerator and denominator

```r
#numerator
numerator   <- factorial(2*n-3)

#denominator
denominator <- 2^(n-2)*factorial(n-2)

#division
numerator/denominator
```

```
## [1] 15
```

## 53.6   Functions in R

Functions in R have the general format

```r
function_name <- function(arguments = ...){
 output <- ...  # math etc saved to object
 print(output)
}
```

The `function_name` can be any valid R object name. The function `function()` creates the function.

A function can have any number of arguments. Note that the arguments are enclosed in parentheses, and after the last ) there is a }.

After all the stuff inside the function (e.g. the math we want the function to do) its finished with a }.

## 53.7 Function to calculate the number of possible phylogenetic trees

We can set up a function to encapsulate this. We'll call the function
`tree_count()`. It has one **argument** (`n`), the number of taxa on the tree.
We'll set the **default value** of the argument to be `n = 3` taxa.

I use the function `print()` at the end so that the function actually returns
output. If this isn't included then nothing will be returned by the function.

In the code below note the locations of the following things

- function name
- function-making command
- The opening parenthesis (
- the argument
- the default
- the closing parenthesis )
- the opening curly bracket {
- the math
- the `print()` command
- the closing curly bracket }

```
#               [ ]function-making command
#              |
# [ ]Function  |          [ ] argument
# name         |          |
# |            |          |    [ ]closing ")"
# |            |          |    |
# |            | [ ] "("  |    |  [ ]"{"
# |            |      \   |    | /
tree_count <- function(  n = 3 ){     # note the "}"

  # [ ] math
  numerator    <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator

  # [ ] print output
  print(trees)

}  # end with the "}"
#\
# [ ] "}"
```

Does it work?

```
tree_count()
```

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

```
tree_count(n = 3)
```

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

When working on a function its also good to check your output against against a source with the correct results. Tables with the number of trees per taxa are in many books, and also available here: https://en.wikipedia.org/wiki/Phylog enetic_tree

We can view the **function source code** by just running the name of the function without any parentheses (or arguments)

```
tree_count
```

```
## function(  n = 3 ){     # note the "}"
##
##    # [ ] math
##    numerator   <- factorial(2*n-3)
##    denominator <- 2^(n-2)*factorial(n-2)
##    trees  <- numerator / denominator
##
##    # [ ] print output
##    print(trees)
##
## }
## <bytecode: 0x7f82078c6098>
```

We can see just the arguments, and any defaults, for a function using `args()`

```
args(tree_count)
```

```
## function (n = 3)
## NULL
```

**ASIDE**: The `print()` function is key. Make and run this function and see what happens.

```
                                    #/ don't forget this!
tree_count_bad <- function(n = 3){

  # math
  numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator

}
```

```
#\ don't forget this!
```

```
tree_count_bad()
```

**END ASIDE**

How rapidly does the number of possible trees grow?

```
tree_count(n = 3)
```

```
## [1] 3
```
```
tree_count(n = 4)
```

```
## [1] 15
```
```
tree_count(n = 5)
```

```
## [1] 105
```
```
tree_count(n = 6)
```

```
## [1] 945
```

One of my favorite example data sets has 11 taxa.

```
tree_count(n = 11)
```

```
## [1] 654729075
```

This number grows very rapidly. According to Bianconi et al (2012) there are $3.72*10^{13}$ cells in the body (37,200,000,000,000, or 3.72e13).

```
### with all the zeros
37200000000000
```

```
## [1] 3.72e+13
```
```
### "e" scientific notation
3.72e13
```

```
## [1] 3.72e+13
```
```
3.72e+13
```

```
## [1] 3.72e+13
```

If we put 14 taxa on a tree there are this many possible trees

```
tree_count(n = 14)
```

```
## [1] 7.905854e+12
```

If we put 15 taxa on the tree

```r
tree_count(n = 15)
```

```
## [1] 2.13458e+14
```

## 53.8  Vectorized inputs to functions

One of R's most powerful features is that it works on vectors. I want to make a plot of the the number of possible trees for 3 to 11 taxa. I can make a vector any way I chose

```r
#typing out all numbers
n.taxa <- c(3,4,5,6,7,8,9,11)

# using seq with arguments spelled out
n.taxa <- seq(from = 3,to = 11,by = 1)

#using seq w/o arguments
n.taxa <- seq(3, 11, 1)

#shortcuts
n.taxa <- c(3:11)
n.taxa <- 3:11
```

I can then put the vector as an input into my function

```r
tree_count(n = n.taxa)
```

```
## [1]        3       15      105      945    10395   135135  2027025
## [8] 34459425 654729075
```
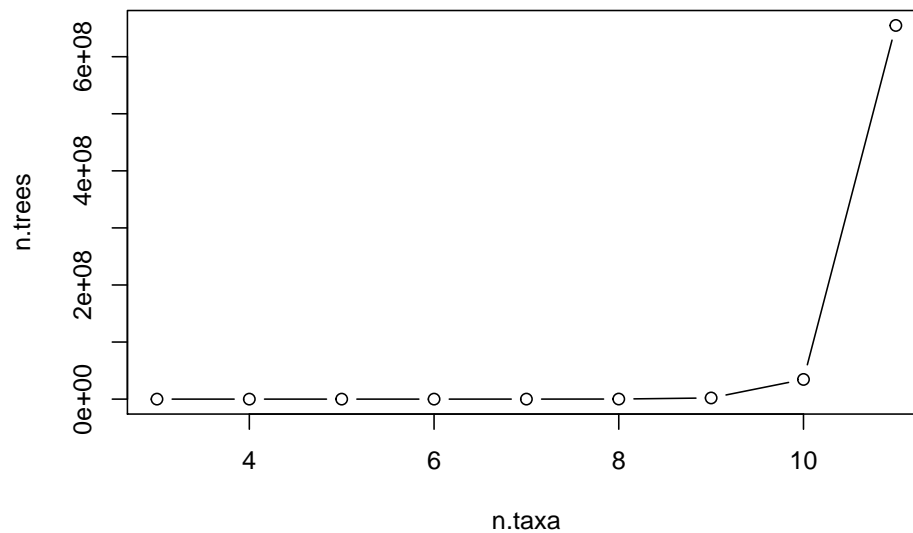
I can save this output to a vector

```r
n.trees <- tree_count(n = n.taxa)
```

```
## [1]        3       15      105      945    10395   135135  2027025
## [8] 34459425 654729075
```

And plot things. type = "b" plots points and a line.

```r
plot(n.trees ~ n.taxa, type = "b")
```

## 53.9 Adding conditional statements

The equation above is only valid for 2 or more taxa. If we put in a lower number the result doesn't make sense

```
tree_count(n = 0)
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## [1] NaN
```

```
tree_count(n = 1)
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## [1] NaN
```

Only when we put in 2 or more taxa does it work

```
tree_count(n = 2)
```

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

```
tree_count(n = 3)
```

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

Additionally, we can input non-integers and get a result.

```
tree_count(n = 3.5)
```

```
## [1] 6.383076
```

We can add a **conditional statement** so that if 0 or 1 are input then an
warning is given. Conditional statements test a logical condition which, if false,
can be used to throw a warning, error, etc.

I'll use the conditional statement if(n < 3) to test if the number of taxa entered
into my tree_count() function is going to be valid. The warning() function
allows me to remind the user of the function of what they should enter.

Note that the conditional statement has the logical operation in parentheses, eg
(n <2), and then what to do if n < 2 is true in curly brackets

```
tree_count2 <- function(n = 3){

  # conditional statement
  ## is n a valid number for using in this equation?
  ## if it is NOT, throw a warning
  ## if it is a valid number, skip everything in the
  ## { } and go to the math
  if(n < 3){

    # warning if test is TRUE
    warning("This function is only valid for 2 or more taxa.")
  }

  # If test is FALSE (n = 3 or n > 3)
  ## continue with the math
  numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator

  # print the results
  print(trees)
}
```

Now we can test this

```
tree_count2(n = 0)
```

```
## Warning in tree_count2(n = 0): This function is only valid for 2 or more taxa.
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## Warning in gamma(x + 1): NaNs produced
```

```
## [1] NaN
```

```
tree_count2(n = 1)
```

```
## Warning in tree_count2(n = 1): This function is only valid for 2 or more taxa.
```

```
## Warning in tree_count2(n = 1): NaNs produced
```

```
## Warning in tree_count2(n = 1): NaNs produced
```

```
## [1] NaN
```

## 53.10   Adding multiple conditional statements

The tree_count2() function above is still throwing error messages because even though it doesn't work with n <2, and even though its now giving us a warning, its still dong the math. We can add a second conditional statement around the math to remedy this.

```
tree_count2b <- function(n = 3){

  # conditional statement with if(){...}
  ## the condition: if n < 2,
  if(n < 2){

    # the result
    warning("This function is only valid for 2 or more taxa.")

  }

  #do the math
  if(n > 2){
    numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator

  #return the result
  return(trees)
  }

}
```

Now we just get the warning we wrote but nothing else

```
tree_count2b(0)
```

```
## Warning in tree_count2b(0): This function is only valid for 2 or more taxa.
```

```
tree_count2b(1)
```

```
## Warning in tree_count2b(1): This function is only valid for 2 or more taxa.
```

## 53.11  Adding additional arguments

Perhaps we want provide the option of always printing the results in scientific notation. We can another argument to our tree_count() function called "format". If the argument is set to "sci", "scientific" or "e", the function will change the options for how numbers are printed.

```r
tree_count <- function(n = 3,                    # 1st argument: number of taxa
                      format = "standard"){ # 2nd argument: format

  # conditional statement with if(){...}
  ## the condition: if n < 2,
  if(n < 2){

    # the result
    warning("This function is only valid for 2 or more taxa.")
  }

  # the math
  numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator

  # another conditional statement:
  ## set formatting
  if(format %in% c("sci","scientific","e")){
    options(scipen = -2,digits = 3)
  }

  # print the output
  print(trees)

  # re-set the formatting
  options(scipen = 0,digits = 7)
}
```

Check out the source code

```
tree_count
```

```
## function(n = 3,                    # 1st argument: number of taxa
##                       format = "standard"){ # 2nd argument: format
##
```

```
##    # conditional statement with if(){...}
##    ## the condition: if n < 2,
##    if(n < 2){
##
##      # the result
##      warning("This function is only valid for 2 or more taxa.")
##    }
##
##    # the math
##    numerator   <- factorial(2*n-3)
##    denominator <- 2^(n-2)*factorial(n-2)
##    trees  <- numerator / denominator
##
##    # another conditional statement:
##    ## set formatting
##    if(format %in% c("sci","scientific","e")){
##      options(scipen = -2,digits = 3)
##    }
##
##    # print the output
##    print(trees)
##
##    # re-set the formatting
##    options(scipen = 0,digits = 7)
## }
```

Check out the agruements

```
args(tree_count)
```

```
## function (n = 3, format = "standard")
## NULL
```

Check the output

```
tree_count(n = 10)
```

```
## [1] 34459425
```

```
tree_count(n = 10, format = "sci")
```

```
## [1] 3.45e+07
```

```
tree_count(n = 10, format = "scientific")
```

```
## [1] 3.45e+07
```

```
tree_count(n = 10, format = "e")
```

```
## [1] 3.45e+07
```

## 53.12 Assignment: __un__rooted trees

We often don't root phylogenetic trees. This reduces the number of possible trees and is described by the equation

Text (note: this had a typo in previous version of assignmen!) (2n-5)!/(2^(n-3))(n-3)!

Rendered:

$$\frac{(2*n-5)!}{2^{n-3}*(n-3)!}$$

## 53.13 Assignment part 1

Modify the equation used above to work for unrooted trees. Call the function tree_count_unrooted() Compare your results to http://carrot.mcb.uconn.edu /mcb396_41/tree_number.html

You can use the simplest form of the function which doesn't have any additional argument, eg

```
#NOTE: this is for a ROOTED TREE
## change math to be for UN-ROOTED TREE
## change name of function to t()ree_count_unrooted
tree_count_rooted <- function(n = 3){
  numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
  trees  <- numerator / denominator
  return(trees)
}


#Always test!
tree_count_rooted(4)
```

## [1] 15

### 53.13.1 Part 1 answer

```
tree_count_unrooted <- function(n = 3){
  numerator   <- factorial(2*n-5)
  denominator <- 2^(n-3)*factorial(n-3)
  trees  <- numerator / denominator
  return(trees)
}
```

CHeck against

```
tree_count_unrooted(n = 3)
```

## [1] 1

```
tree_count_unrooted(n = 4)
```

## [1] 3

```
tree_count_unrooted(n = 5)
```

## [1] 15

Compare rooted and unrooted

```
tree_count_unrooted(n = 5)
```

## [1] 15

```
tree_count_rooted(n = 5)
```

## [1] 105

## 53.14 Assignment part 2

Create a function that will work for rooted OR unrooted trees. Do this by adding an additional argument like

type = "rooted"

and conditional statements like

if(type == "rooted"){ #do this }

if(type == "unrooted"){ #do something else }

Again, you can use the simplest form of the argument.

(Note: there was a typo in the original version of this where there were missing quotation marks around "rooted" and "unrooted" eg "rooted, so if you copy and pasted the code - which is what I would've done! - it wouldn't have worked).

### 53.14.1 Answer

here is one way this could be written.

```
tree_count3 <- function(n = 3,
                        type = "rooted"){

if(type == "rooted"){
  #run rooted vs. of the calculation
  numerator   <- factorial(2*n-3)
  denominator <- 2^(n-2)*factorial(n-2)
```

```r
  trees  <- numerator / denominator
  }

if(type == "unrooted"){
  #unrooted version of the equation
  numerator   <- factorial(2*n-5)
  denominator <- 2^(n-3)*factorial(n-3)
  trees  <- numerator / denominator
  return(trees)
 }

return(trees)

}
```

Check the answer

```r
tree_count3(n= 5, type = "rooted")
```

```
## [1] 105
```

```r
tree_count3(n= 5, type = "unrooted")
```

```
## [1] 15
```

We can plot rooted vs unrooted

```r
n.taxa <- seq(3,9,1)
y.rooted <- tree_count3(n= n.taxa, type = "rooted")
y.unrooted <- tree_count3(n= n.taxa, type = "unrooted")

plot(y.rooted ~ n.taxa, type = "b")
points(y.unrooted ~ n.taxa, type = "b", col = 3, lty = 2)
```