Contents

Multiple sequence alignment in R

Preamble

In this practical we will make use of a number of packages that allow us to handle biological sequences in R. We will also make use of core R functions to visualise the results and to prepare data input rather than relying completely on existing functions. The aims are:

1. To to improve your core R skills
2. To expose you to R packages useful for handling sequences in R
3. To allow you to perform multiple sequence alignment and to become aware of some of the problems.
4. To expose you to object orientation and inheritance in R
5. To get used to reading the help pages in R

For this we will use the following packages:

1. `msa`: provides multiple sequence alignment
2. `Biostrings`: provides functions for handling biological sequences, including the output of the msa functions
3. `seqinr`: some format conversion
4. `ape`: phylogenetic tree building

The package `msa` contains compiled code that implements several multiple sequence alignment algorithms (including `clustal` and `muscle`). The msa function returns an object that is of a class derived from a class defined by the `Biostrings` package. That doesn't only sound like a mouthful, it is also (to me anyway) a bit of a handful to deal with. You will see in the exercises later on.

The `Biostrings` package provides classes that represent biological sequences in an efficient manner. This overcomes many of the problems with handling strings in `R`, but means that you cannot simply use core `R` functions on these objects without performing data conversions. Accessing the underlying information can thus be a bit painful as it requires learning functions which are specific to the `Biostrings` classes (from reading the very long manuals).

The `Biostrings` package provides functions for holding and reading multiple sequence alignments. Using these allows you to perform multiple sequence alignment using other programs and to then import the alignments into `R` for further analysis. Here we will use the `msa` package to perform the sequence alignments within R; however, if you have time you can also use the `Biostrings` functions to read in data from a prior sequence alignment and have a look at that.

The `ape` package provides a large number of functions related to phylogenetic tree building and associated analyses. Here we can use it to make and look at trees in various ways.

The sequence data

I made use of the `Ensembl`[1] database to obtain a number of sequences that can be meaningfully aligned to each other. To do this I searched for Sox17, and from the page for the human Sox17 gene I clicked on the `orthologues` option in the left hand control panel. This leads you to a wonderful page where you can download the orthologue sequences, both before and after alignment. I downloaded both coding cDNA (CDS) and protein sequences as well as the aligned protein sequences. You can find these in the files:

1. Human_SOX17_orthologues_cds.fa
2. Human_SOX17_orthologues.fa
3. Human_SOX17_orthologues_al.fa

You should be able to download the sequences from Canvas[2]. I leave it to you to work out what they are. If you want to look at them, I suggest that you open a terminal on your computer (on Windows this means run `cmd.exe`; on Mac the terminal is called 'terminal'). In the terminal navigate[3] to the directory where you saved the sequence files (this **should** be the same, or a strong relative, of the directory where you made a new project in `R`) for this practical.

The sequence data is in the (multiple sequence) fasta format. This means:

```
>identifier optional text
ACATAACGATAVA
ADCATAGAGA
>identifier optional text
ACATAGTAA
ACATAA
```

If you manage to look at the files you will see something like:

[1] http://www.ensembl.org

[2] Remember to download the files to a new directory where you will start a new `R` session by opening this file in `Rstudio`.

[3] On windows use `cd` to change directory (without any argument to print out the current working directory) and use `dir` to list the contents of a directory. On Macs, use `cd` to change directory, `pwd` to print the working directory and `ls` to list contents. `cd <dir_name>` enters a directory, and `cd ..` goes up one level in the hierarchy.

```
>ENSOPRP00000006427
MSSPDAGYASDDQSQPRSALPAVMAGLGPCPWAESLSPLGDVKAKGETAASSGAAAGAAG
RAKGESRIRRPMNAFMVWAKDERKRLAQQNPDLHNAELSKMLGKSWKALTLAEKRPFVEE
AERLRVQHMQDHPNYKYRPRRRKQVKRLKRVEGGFLHGLAEPQAAAMGPETGRVAMDGLG
LPFPEQGFPAGPPLLPHMDGHFDCQLDAPPLDGYPLATPDTPLDSVEHDPTISAAPLHGD
PRAGTYSYAQVSDYAVAAEPPAPVHPRLGPESAGPAMPGLLAPPSALHMYYGGMGSPSVG
GGRSFQMQPPPQQQQPQQHPHPPGPGQPSPPHEALPCRDGADPSQPAELLGEVDRTEFEQ
YLHYVCKPDLGLPYQGHDSNVNLSDSHGALSSVVSDASSAVYYCNYPDV
>ENSPPYP00000020852
MSSPDAGYASDDQSQTRSALPAVMAGLGPCPWAESLSPIGDMKVKGEAPASSGAPAGAAG
RAKGESRIRRPMNAFMVWAKDERKRLAQQNPDLHNAELSKMLGKSWKALTLAEKRPFVEE
AERLRVQHMQDHPNYKYRPRRRKQVKRLKRVEGGFLHGLAEPQAAALGPEGGRVAMDGLG
LQFPEQGFPAGPPLLPPHMGGHYRDCQSLGAPPLDGYPLPTPDTSPLDGVDPDPAFFAAP
MPGDCPAAGTYSYAQVSDYAGPPEPPAGPMHPRLGPEPAGPSIPGLLAPPSALHMYYGAM
GSPGAGGGRGFQMQPQHQHQHQHQHQHHPPGPGQPSPPPEALPCRDGTEPSQPAELLGEV
DRTEFEQYLHFVCKPEMGLPYQGHDSGVNLPDSHGAISSVVSDASSAVYYCNYPDV
>ENSTNIP00000004898
IRRPMNAFMVWAKDERKRLAQNPDLHNAELSKMLGKSWKALPVTEKQPFVEEAERLRVQH
MQDHPNYKYRPRRRKQVKRIKRLDSGFLVHGAADHQSQSIAGEGRVCMESLGLGYHEHGF
QIPSQPLSHYRDAQALGGPSYEAYSLPTPDTSPLDAVEPDSMFFQPHSQEDCHMMPTYPY
```

The identifiers used here are `Ensembl` stable identifiers. To map these to specific species we will use the `ensembl_species_codes.txt` file and a bit of (not very efficient or clear) `R`.

There are a total of 195 different sequences in these files. You probably do not want to align all of them in one large multiple sequence alignment as that is likely to take rather too much time. Instead you can try to align subsets of varying sizes to work out how many are fine. Remember that execution time should scale roughly with the square of the number of sequences.

Starting up in R

1. Download the sequence and the `ensembl_species_codes.txt` files as well as this markdown file to a new directory with a sane name.
2. Start RStudio in this new directory by opening the markdown file. You may also try to get RStudio to make a new project associated with this directory. Look at the `file` menu and see if you can work out how to do that.
3. Create two new `R` script files. Call one of these something like `main_msa.R` and the other `msa_functions.R`. You should only execute code from the first of these directly; code in the second file should be run by calling `source("msa_functions.R")` from the `main_msa.R` file. The `msa_functions.R` file should **only** contain function definitions and associated comments. Code in the `main` file will make use of (i.e. call) the functions defined in the `functions` file *after* this has been sourced.

Reading in the sequence files

Your own function for reading fasta files Although you can find functions for reading in fasta formatted files in at least two of the packages we are going to use I want you to write a function in core `R` that does this. This will not be a fast and efficient function, but that doesn't matter for this exercise.

In your functions file enter the following code:

```
## fn should be a file name that specifies a fasta formatted file
read.fasta <- function(fn){
    data <- readLines(fn)
    id.lines <- grep("^>", data)
    seq.beg <- id.lines + 1
    seq.end <- c(id.lines - 1, length(data))[-1]
    seq <- mapply( function(b,e){
        paste( data[b:e], collapse="" )},
        seq.beg, seq.end )
    names(seq) <- sub("^>", "", data[id.lines])
    seq
}
```

The commands in the function will be explained a bit later on as there are a number of new things introduced here.

Once you have entered this code into your functions file and **saved it**, you can `source` it from your main file[4]:

```
source("msa_functions.R")
```

After you have executed this command from the script, try the following in the console (note that you *do not* need to enter the comments!):

```
## see what happened:
ls()

## and write the name of the function without the ()
read.fasta
```

What `source` does and how `read.fasta` works When you `source` the functions file, R will run (execute) all of the code within the file. When the `function` function is called it returns an object of class "function" and assigns it to whatever is specified (`read.fasta`) here. After this you can *call* the function by putting the arguments to it in brackets after the function name. You can also pass the function object to other functions as an argument. Or see the function

[4]To source the file from your main file, enter the above code and execute it by 'sending' it to the console (in `RStudio`, put the cursor on the line containing the `source` command and hit enter.

name when you list (`ls`) the object present in the current session. And you can see the class of the objects by calling `class` on them;

```
class( read.fasta )
```

The `read.fasta` function introduces some of the fundamental tools for handling data in `R`. The expression beginning on the first line:

```
read.fasta <- function(fn){
    ...
}
```

defines a new function that takes a single argument. That argument is assigned (sort of copied) to the variable `fn` which can be used within the body of the function. The body of the function is enclosed by the *curly brackets* (`{` and `}`). If you call this function:[5]

```
read.fasta("name_of_some_fasta_file.fa")
```

the code within the curly brackets (also referred to as braces) will be executed with the value of `fn` set to `"name_of_some_fasta_file.fa"`. If there is no file of that name in the given location, the function will simply return with an error and nothing will happen (this being one of the best things about `R`).

The first line of the function body:

```
data <- readLines(fn)
```

Calls the `readLines` function with the argument set to `fn`. That is, it will ask `readLines` to read from the file specified as an argument to `read.fasta`. `readLines` reads one line at a time from the file and returns the file content as a `character` vector (essentially a list of strings, but remember in `R` a `list` has a special meaning).

The next lines of the function:

```
id.lines <- grep("^>", data)
```

calls the function `grep` on the `character` vector (`data`) containing the lines of text. `grep` examines each entry of the character vector given as its second argument (`data` in this case) and checks whether it contains the pattern given as its first argument. The pattern specified here (`"^>"`) matches any string that begins with the `">"` character; such lines should contain the sequence identifiers (consider what is specified by the fasta format). `grep` returns the index (the line numbers) of the lines that match the pattern (if any).

You can try the following to get a feeling for how `grep` works and how it is used above to identify the lines containing identifiers. In the console, or the main script (if you want to remember what you did):

[5] Don't call it like above; there is no file called `name_of_some_fasta_file.fa`. That bit should be replaced with the name of an existing file.

```r
## this makes a character vector:
data <- c(">id1", "ACTAGATAGA", "GATAGATAACT", ">id2", "TAATTATATAGCAGAT", "TATATACTA", "ATACTA")

## this uses grep to search for data that start with the > character
## (the ^ indicates that the match must be made at the first character).
grep("^>", data)

## by default, grep returns the indices of the rows that match. But
## we can also ask us to return the values:
grep("^>", data, value=TRUE)

## we can also use the return value of grep to subset data.
## this gives use the same thing as above
data[ grep("^>", data) ]

## we can also look for any character pattern that we want:
grep("ATTA", data)

grep("ATTA", data, value=TRUE)

## look for entries that begin with A
grep("^A", data, value=TRUE)

## or begin with AC
grep("^AC", data, value=TRUE)

## to remember where the identifiers are:
id.lines <- grep("^>", data)
```

Note that if you run the above code in the console that you should delete the `data` and `id.lines` objects after you are finished with them. Otherwise you will have objects in your current session that you don't know the origin of and that is *bad*. For now though you can keep them around for a while to try out some other parts of the function.

The fasta format specifies that sequences begin on lines following the identifier lines; we can obtain these by simply adding 1 to the `id.lines` vector.

We can also infer that sequences will end on the lines immediately before the identifier lines. Except for the last sequence which should end on the last line. And of course there is no sequence above the first identifier, so we need to get rid of that as well.

The next two lines of code use this logic to obtain the line numbers corresponding to the beginning and end of sequences. You can try to run these in the console if you have defined an `id.lines` vector as in the code above. Look at the values of `seq.beg` and `seq.end` and see if you can work out what happened. Compare the output with and without the weird `[-1]` subsetting at the end of the expression.

```
    seq.beg <- id.lines + 1
    seq.end <- c(id.lines - 1, length(data))[-1]
```

The first of these lines is straightforward; it merely adds one to every element of `id.lines`.

The second line looks a bit weird; whenever you see a line like this, simply break it down into pieces and follow the logic. Start from inside the brackets reading from left to right. Again, if you have the `id.lines` vector, try this and following commands:

```
id.lines - 1
```

That part is OK, it just subtracts one from every element of `id.lines`. In order to add the last line to this we use the concatenate `c` function. This will simply merge its arguments into a single vector:

```
c(id.lines - 1, length(data))
```

`length(data)` simply returns the number of entries in `data` (which contains the lines read in by `readLines`). At this point we are almost done; we only need to remove the first element (which will point at the line above the first identifier, which probably doesn't exist). In `R` we can select elements using square brackets; usually we do this on named variables, but we can also do so directly on data returned from a function. Here we use `[-1]` to select a subset of the lines returned by the `c(id.lines - 1, length(data))` call. Negative indices *remove* lines in `R` (this is quite `R` specific):

```
## what does id.lines contain:
id.lines

## and what does id.lines[-1] give you
id.lines[ -1 ]

## is equivalent to:
id.lines[ 2:length(id.lines) ]
## Remember that 2:length(id.lines) will return a vector of elements
## from 2 to the number of lines in id.lines; and that you can also use
## vectors of elements to select subsets of vectors.
```

At this point we have identified the set of lines containing identifiers, sequence beginnings and ends. We now want to merge the lines containing each sequence. We can use `paste` for this. And to do it for every sequence we can use the `mapply` function. The `mapply` function is part of a family of functions in `R` that are used to *apply* a function to every element of a data structure. The `mapply` function is special in that it allows us to specify more than one vector to which the function should be applied.

```
    seq <- mapply( function(b,e){
        paste( data[b:e], collapse="" )},
```

```
        seq.beg, seq.end )
```

The first argument of `mapply` should be either the name of a function, or a function definition. The function should take the number of additional arguments given to `mapply`. In this case we want to select the lines between every pair of `seq.beg` and `seq.end` elements; so the function should take two arguments. Since these represent the beginning and end indices of the lines containing each sequence we call them `b` and `e`. The code will then be executed for:

1. The 1st element of `seq.beg` and 1st element of `seq.end`
2. The 2nd element of `seq.beg` and 2nd element of `seq.end`
3. The 3rd element of `seq.beg` and 3rd element of `seq.end`
4. and so on until we reach the end of the vectors.

The code that is executed is a call to `paste`. When the option, `collapse` is specified to `paste` it will merge elements specified as vectors[6]. This is sufficient to merge the sequences into a simple character vector.

To understand the `mapply` function, try the following:

```
## this will run the sum function on pairs of values:

mapply(sum, 1:10, 1:10)

## this will multiply pairs of values using an anonymous function
mapply(function(a, b){ a * b }, 1:10, 1:10)

## and this will use paste to combine three elements to text:
mapply(paste, 1:10, 10:1, 20:29)
```

In `R`, you can also assign names to vectors that are associated with each element of the vector; these can then be used to access individual elements or to identify the name of a specific element. To assign names to vectors we use a somewhat strange expression:

```
names(seq) <- a_character_vector
## a_character_vector should be a vector of the
## same length as the vector seq
## and all its elements should be unique
```

That is we assign a value to a result of a function call. Normally when we call a function we assign its return value to a variable; here we are doing the opposite thing. That's a bit strange, and the justification for doing so is a bit complicated.

But in general;

[6]The `paste` function behaves very differently depending on whether the arguments to it are specified as a vector or individual arguments; that is, `paste(1, 2)` gives a different result to `paste(c(1, 2))`.

```
a <- f(b)
```

i.e. a call to a function (`f`) with the argument `b` cannot result in the value of `b` being modified. In the above expression the only variable that can be modified is `a` (which will be assigned the return value of the function call).

However, expressions like:

```
f(b) <- a
```

can result in modification of the variable `b`. When we call `names(seq) <- s.names`, we want to modify the vector by setting its names attribute. This kind of expression is very common in R code and you should be familiar with it.

In the above code, the actual code we use is:

```
names(seq) <- sub("^>", "", data[id.lines])
```

That is we assign the names not from a vector directly, but from an expression that evaluates to a vector. In this case the expression is:

```
sub("^>", "", data[id.lines])

## compare to:
data[id.lines]
```

The `sub` function substitutes matches of regular expressions (given as its first argument) in its third argument with the value in the second argument. Here we want to remove all leading `">"` characters as they are not formally part of the sequence identities. You should notice that the regular expression used is the same that we used to identify the lines that contain sequence identifiers. Here we simply replace the character with an empty string.

In the final line of the function we simply write the name of the variable that the function should return.

To summarise the function does the following:

1. Reads all lines from the specified file (`fn`) into the vector `data`.
2. Identifies all lines beginning with `">"` and stores their line numbers in `id.lines`
3. Determines the lines where sequences begin; this is immediately after the identifiers, so is simply equal to `id.lines + 1`. Assigns these values to `seq.beg`.
4. Determines the lines where sequences end; sequences will generally end on the line before the next identifier, `id.lines - 1`. However, the first sequence will end on the line before the second identifier, and the last sequence will end on the last line; so we need to modify the vector to add the number of the last line and to remove the first entry resulting from `id.lines - 1`. Assigns these to the `seq.end`.
5. Merges the lines between and including every pair of elements in `seq.beg` and `seq.end`.

6. Assign names to the resulting sequences.
7. Return the sequences.

Although you can find packages containing functions that do the same as this `read.fasta` function, it is worthwhile to learn how to write it. This is mainly because it should give you the skills to read more or less any kind of data into `R`, but also because it is often less trouble to simply write a function like this than to spend the time finding a package that contains it, and then working out exactly what it does (there are many subtle ways to change the behaviour of this function).

Finally read the sequences into R You should have downloaded the sequence data files mentioned above to the directory associated with your R session. If you have done that, it should be enough for you to run the following from your main R script (i.e. type this into your main script and then execute it:

```
pep <- read.fasta("Human_SOX17_orthologues.fa")

pep.al <- read.fasta("Human_SOX17_orthologues_al.fa")

cdna <- read.fasta("Human_SOX17_orthologues_cds.fa")
```

I have put empty lines between each command here to emphasise that I want you to run each command separately. I believe RStudio has a way of doing this without intervening empty lines; and if it does you can find the answer from Google. From here on, you should try to decide yourself whether you want to run several statements in one go or to limit yourself to single ones.

After reading any kind of data into `R` you should have a look at what actually happened. You should never assume that the operation did what you thought it was supposed to do. In this case you can try the following:

```
### run these for all of the objects that you have read in
### run each command separately as you want to look at the output
summary(pep)
length(pep)
class(pep)

head(pep)
head(cdna)
head(pep.al)

tail(pep)

names(pep)
head( names(pep) )
tail( names(pep) )
```

```
names(pep) == names(pep.al)
names(cdna) == names(pep)

all(names(pep) == names(cdna))
any(names(pep) == names(cdna))
```

Note that you would normally run these commands in the console rather than from your script. This is because you are only making sure that things are as you think they should be. In this case you *might* want to save the commands to your script if you think it will help you to remember them in the future. If you do, you should also add some comments to explain what they do and perhaps what they return. For example you could write:

```
## this will tell me the class and length of the pep object
summary(pep)
## this returned:
##   Length     Class      Mode
##      195 character character
## indicating that I have 195 sequences
```

Exactly what you write is up to you. In general it is better to write too much rather than too little. If you are wondering what the above commands do, the remember that you can always use the help command:

```
?all
```

(Again, this you would not normally enter into your script).

Finally to get a better idea of what you have read in, lets see how long the sequences are:

```
## run as a separate command for each.
nchar(pep)
nchar(pep.al)
nchar(cdna)
```

You should have noticed a pattern there; if you haven't seen it, you can try the following:

```
nchar(cdna) / nchar(pep)

## table returns the number of instances of each value
table( nchar(cdna) / nchar(pep) )

## that's a bit weird; you can try
## ( %/% does integer division )
table( nchar(cdna) %/% nchar(pep) )

## and maybe
```

```
table( nchar(cdna) %% 3 )

## and finally
table(nchar(pep.al))
```

Here the `nchar` command returns a named vector of integral values. It's quite difficult to get some sense of the total set of values. You could of course calculate some summary statistics for these:

```
mean( nchar(pep) )
sd( nchar(pep) )
median( nchar(pep) )
mad( nchar(pep) )

mean( nchar(pep.al) )
sd( nchar(pep.al) )
median( nchar(pep.al) )
mad( nchar(pep.al) )

## what about the summary function?
summary( nchar(pep) )
summary( nchar(pep.al) )

## same as:
quantile( nchar(pep) )
quantile( nchar(pep.al) )
```

Although it's difficult to get a good feeling about the nature of the data from the above statistics you should have noticed something significant. To get a better idea of what a load of numbers look like we normally try to visualise them in some way. Here you can try:

```
## the distributions
hist( nchar(pep) )
hist( nchar(pep.al) )
hist( nchar(cdna) )

## quantile plots
plot( sort(nchar(pep)) )
plot( sort(nchar(cdna)) )
plot( sort(nchar(pep.al)) )

plot( sort(nchar(pep)), type='l' )
plot( sort(nchar(pep)), type='b' )

## or even
plot( nchar(pep), nchar(cdna) )
## we have a certain expectation that we can check:
```

```
abline( 0, 3, col='red' )
```

If you are happy with the sequences we can now try to run some multiple sequence alignments on these. But before you do, you should feel comfortable with the above commands and the sequence data that you have read.

Run some MSAs

Install the MSA package (and its dependencies) To do multiple sequence alignment we will make use of the `msa` package. This package is not a `CRAN` package, but is a `Bioconductor` package. To use it, you first have to install it. This requires that you have the appropriate functions available; these can be obtained either by installing a package (newer method) or by sourcing a file available on the Bioconductor web site. Here we will use the more modern method:

```
if(!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")

## pay attention to what happens when you run the next line:
BiocManager::install("msa")
```

The first part of this code essentially asks whether or not `BiocManager` package is installed. If not then it installs it. The next expression uses the `install` function of the `BiocManager` package to install the `msa` package from the Bioconductor archive. In general this works; sometimes you may get errors related to the R version installed.

Here you need to call `BiocManager::install()` for all packages that you are going to use.

The installation of packages can be quite time consuming; probably more so on Linux than on Windows as installation on Linux defaults to compiling all `C` code included whereas Windows uses precompiled packages[7].

When you install `msa` you should note that other packages will also be installed. This is because the `msa` package depends on the `Biostrings` package and in fact extends one of the `Biostrings` packages.

After installing a package you still need to tell `R` that you want to use it. (Note that you need only install a package once; once it is installed you should be able to use it from any R session on your computer).

```
## To tell R that you want to use the msa (and other) package there are two options:
library(msa)

## or
require(msa)
```

[7] Alright, I admit it; one case where things may be more convenient on Windows. There are a few more, but not many…

```
## it doesn't generally matter which you use.
## but if you are interested, as usual do:
?library
?require
## to get the answer.
```

Do your first multiple sequence alignments Multiple sequence alignments are expensive to compute and I've provided you with 195 sequences. It might not be a good idea to try to align all of those sequences at the same time (this will require approximately $\frac{200^2}{2} \approx 20000$ individual alignments. Hence we will instead use subsets of sequences and determine the amount of time required.

```
## to do a multiple alignment of 10 peptide sequences we use
## a call to the msa function.
## to determine how long this takes we wrap the call inside a call
## to the system.time function:

system.time( msa(pep[1:10], type="protein") )
## on my machine this prints out:
## use default substitution matrix
##   user  system elapsed
##  0.236   0.004   0.239

system.time( msa(pep[1:20], type="protein") )
##  user  system elapsed
## 0.641   0.000   0.640

## how about the cdna sequences?
system.time( msa(cdna[1:10], type="dna") )
  ##  user  system elapsed
  ## 1.447   0.000   1.447

system.time( msa(cdna[1:20], type="dna") )
  ##  user  system elapsed
  ## 4.609   0.000   4.609
```

Try to determine how many sequences that you can align in a reasonable amount of time, and whether you want to make use of protein or cDNA sequences. Note that if you want to use a random selection of sequences rather than just the first `n`, then you can use the `sample` function. See `?sample` to work out how.

In the above code we have not assigned the return value of the call to `msa` to any variable. So the calls are kind of useless apart from giving us an idea of how much time it will take to align the sequences. In the following code replace the definition of `n` with the number of sequences that you wish to align.

```
## replace the 30 with a value that is suitable for your computer
```

```
n <- 30
## and choose whether you want to align cDNA or peptide sequences
## here I align cDNA sequences
cdna.msa.1 <- msa(cdna[1:n], type="dna")

## now try to look at what you got:
cdna.msa.1
summary(cdna.msa.1)
length(cdna.msa.1)
names(cdna.msa.1)
```

Having obtained the alignment we can see that it is an alignment, but there is no obvious way to access the alignment details. Welcome to the wonderful world of `S4` objects. Unfortunately the only way to work out what to do is to read the manual. And that is unfortunately rather long. And you will need to read more than one manual. I will try to show the process for how to do that here.

First check the help for the `msa` function:

```
?msa

msa                     package:msa                     R Documentation

Unified interface to multiple sequence alignment algorithms

Description:

     The 'msa' function provides a unified interface to the three
     multiple sequence alignment algorithms in this package:
     'ClustalW', 'ClustalOmega', and 'MUSCLE'.

Usage:

          msa(inputSeqs, method=c("ClustalW", "ClustalOmega", "Muscle"),
              cluster="default", gapOpening="default",
              gapExtension="default", maxiters="default",
              substitutionMatrix="default", type="default",
              order=c("aligned", "input"), verbose=FALSE, help=FALSE,
              ...)

Arguments:

inputSeqs: input sequences; this argument can be a character vector, an
           object of class 'XStringSet' (includes the classes
           'AAStringSet', 'DNAStringSet', and 'RNAStringSet'), or a
           single character string with a file name. In the latter case,
```

...

The help page goes on for a long time. In this case we are interested in what was returned by the function, and so we scroll down until we reach the `Value` section:

```
Value:

     Depending on the type of sequences for which it was called, 'msa'
     returns a 'MsaAAMultipleAlignment', 'MsaDNAMultipleAlignment', or
     'MsaRNAMultipleAlignment' object.  If called with 'help=TRUE',
     'msa' returns an invisible 'NULL'.
```

This tells us that `msa` returns an object of the class `MsaAAMultipleAlignment`, `MsaDNAMultipleAlignment` or `MsaRNAMultipleAlignment`.

To find out which we use the `class` function:

```
## in my case:
class(cdna.msa.1)
## which prints:
[1] "MsaDNAMultipleAlignment"
attr(,"package")
[1] "msa"
```

To find out how to handle this class we again use the help function:

```
?MsaDNAMultipleAlignment
```

Again, you get a long document. The part that you are interested in is the `Methods` section. Here we find only one method, `print`, and that isn't that interesting. Instead reading in the `Details` section we find that:

```
Details:

     The class 'MsaAAMultipleAlignment' extends the
     'AAMultipleAlignment' class, the class 'MsaDNAMultipleAlignment'
     extends the 'DNAMultipleAlignment' class, and the class
     'MsaRNAMultipleAlignment' extends the 'RNAMultipleAlignment'
     class. All three classes extend their parent classes by the slots
     contained in the 'MsaMetaData', i.e. all three classes are class
     unions of the aforementioned parent classes and the class
     'MsaMetaData'.
```

the classes defined by `msa` actually extend some other classes. To extend a class essentially means to add extra functions or data parts to the class. The functions defined by the parent class should all be available. So here I want to look at the help for the `DNAMultipleAlignment` class. You may wish to look at some of the others.

Again, we use the help function, `?DNAMultipleAlignment`. At the top you will

find that the class is actually provided by the `Biostrings` package; not the `msa` package. This explains why `Biostrings` got installed when we installed `msa`. You can also see that you can create these kind of objects by reading alignments from files (which means that you can use any external program you like as long as it can export alignments to a supported format).

Further down you can find a section entitled `Accessor methods`. This will describe functions which allow you to obtain the data contained within the object. You can also see that there is a section called `Coercion`; this gives information on how to convert the data to other types. You can try some of these out:

```
## simple dimensions
nrow(cdna.msa.1)
ncol(cdna.msa.1)

## coerce (convert) to a matrix or a character vector
cdna.msa.1.ch <- as.character(cdna.msa.1)
cdna.msa.1.m <- as.matrix(cdna.msa.1)

## have a careful look:
length(cdna.msa.1.ch)
head(cdna.msa.1.ch)

dim(cdna.msa.1.m)
```

Looking at the documentation you can also find that there are a number of utility functions, including things like: `consensusMatrix`, `consensusString`, `alphabetFrequency`, `detail`. I would recommend that you have a look at some of these to see what they do.

We can also use some core R functions to get an idea about how the sequences were aligned (starting with the matrix object returned by `as.matrix`).

```
## we can convert the cdna.msa.1.m to a logical matrix
## containing TRUE values where we have gaps and FALSE values
## everywhere else. This can then be viewed using the image function

cdna.msa.1.m.l <- cdna.msa.1.m == "-"
image(cdna.msa.1.m.l)

## maybe try:
image(t(cdna.msa.1.m.l))

## or better yet:
image(x=1:ncol(cdna.msa.1.m), y=1:nrow(cdna.msa.1),
      t(cdna.msa.1.m.l), xlab="position", ylab="sequence")
```

You should be able to work out what the image tells you; if not discuss with your neighbours or ask me. Interpreting this image is actually one of the more

important things in the whole exercise.

The rows in the above image represent the individual sequences. It would be nicer if we had some information as to where these sequences come from (i.e. what species). We can do so by mapping the codes in the file `ensembl_species_codes.txt` to the identifiers. This is a bit of a pain to do. I will give you some code that will do this and show you how to use it for the figure; but I will leave it to you to work out how it works. Don't be too concerned if you can't; but do try to look at what each expression and command returns and see if it makes sense.

Note that this code is something that might be used more than once and that it might be better to write it as a function and put it in the functions file.

```
ens.codes <- read.table(("ensembl_species_codes.txt"),
                        sep="\t", stringsAsFactors=FALSE, quote="" )

sp.i <- sapply( ens.codes[,1], function(x){
    which(sub("^([^0-9]+)?P[0-9]+", "\\1",  names(cdna)) == x) })

cdna.sp <- rep(NA, length(cdna))
for(i in 1:length(sp.i)){
    cdna.sp[ sp.i[[i]] ] <- ens.codes[i, 2]
}

cdna.sp[is.na(cdna.sp)] <- "unknown"
names(cdna.sp) <- names(cdna)
```

And with that we can make a more informative figure:

```
par(mar=c(5.1, 20.1, 5.1, 2.1))
image(x=1:ncol(cdna.msa.1.m), y=1:nrow(cdna.msa.1),
      t(cdna.msa.1.m.l), xlab="position", ylab="", yaxt='n')
axis(side=2, at=1:nrow(cdna.msa.1), labels=cdna.sp[ rownames(cdna.msa.1)], las=2 )
```

From this picture it should be obvious that you probably would not want to include all of the sequence data when attempting to construct a phylogeny. A large part of the sequence data is only present in a subset of the species and for these sequence there *are no* homologous sequences in the other species to align to.

Non-default options to `msa` From the `msa` help you should have noticed that there are many different ways in which you can do the alignments. You should have a look at some of the alternatives. In particular try using the `Muscle` algorithm instead of `ClustalW` and see how this affects speed and performance. For protein sequences you may want to consider if you can change the substitution matrix as well. I will provide a simple example, but it is better if you try to work out other alternatives.

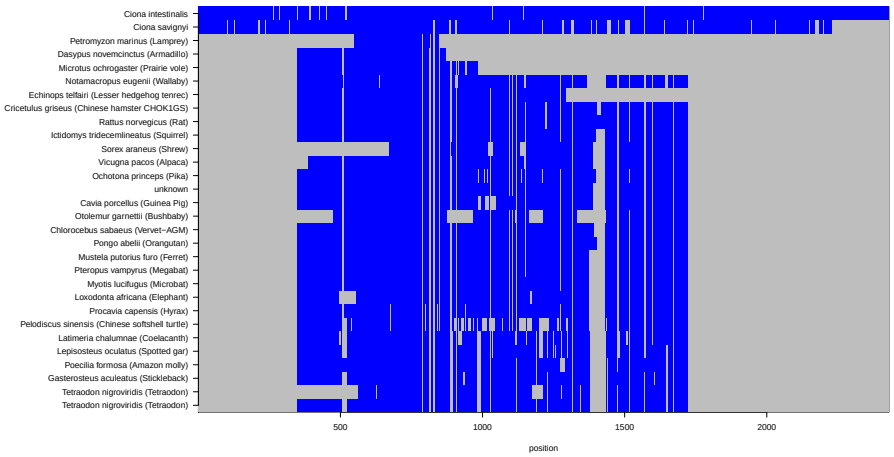

Figure 1: Multiple sequence alignment of a set of Sox17 cDNA sequences (gaps=grey, residues=blue)

```
## first lets try simply changing to the muscle
## method seeing how it impacts the speed.
system.time( msa(cdna[1:10], method="Muscle", type="dna"))
##   user  system elapsed
##  1.852   0.000   1.852

system.time( msa(cdna[1:12], method="Muscle", type="dna"))

## *** ERROR ***  MSA::SetIdCount: cannot increase count

## Fatal error, exception caught.
## Error in msaFun(inputSeqs = inputSeqs, cluster = cluster, gapOpening = gapOpening,  :
##   MUSCLE finished by an unknown reason
## Timing stopped at: 0.001 0 0.001
```

Hmm, that's a bit disappointing; with the version that I am using anyway, it seems that the Muscle implementation doesn't work for more than 10 sequences. That should be a bug, and it might be that it works for you since I don't have the latest version of R.

Distances from the alignments

We can obtain distances between the sequences in a number of different ways and there are a number of packages that provide functions for this. However, we can also simply use the distance definition used by `clustal` and implement a function that calculates this.

Put the following code into your functions file:

```
## msal should be a character matrix as returned by
## as.matrix( <msa_alignment> )
seq.distances <- function(msal){
    ## apply works on rows or columns. The second argument
    ## defines which. Here we specify 1, which means rows
    ## r1 and r2 are variables that represent a row of the matrix
    apply( msal, 1, function(r1){
        apply( msal, 1, function(r2){
            ## define the columns that are not gaps in both rows:
            ## I use a variable with b in it, for Boolean
            b.ng <- (r1 != "-") & (r2 != "-")
            ## and define which of these are the same
            b.id <- b.ng & (r1 == r2)
            ## the sum of a logical vector containging TRUE / FALSE
            ## values, is equal to the number of TRUE values
            ## here all that are b.id are not gaps and identical in the
            ## the two rows. To return a distance we can thus do:
            1 - sum(b.id) / sum(b.ng)
        })
    })
}
```

Then save and `source` the functions file. And run the following from your main file:

```
## but note that you may need to change cdna.msa.1.m to
## whatever you called the variable that you obtained by
## as.matrix( <your alignment> )
cdna.msa.1.d <- seq.distances( cdna.msa.1.m )

## then have a look at the return value
dim(cdna.msa.1.d)

par(mar=c(15.1, 15.1, 4.1, 2.1))
image(x=1:30, y=1:30, cdna.msa.1.d, yaxt='n', xaxt='n', ylab="", xlab="", asp=1)
axis( 1, at=1:30, labels=cdna.sp[1:30], las=2, cex.lab=0.25 )
axis( 2, at=1:30, labels=cdna.sp[1:30], las=1, cex.lab=0.25 )

cdna.msa.1.d
```

To get a tree from these distances we can run the `nj` function from the `ape` package:

```
## note that you may as before need to replace the
## 'cdna.msa.1.d' with whatever you called your distance
## matrix
```

```r
## You may need to:
BiocManager::install("ape")
## and then
require("ape")

cdna.msa.1.nj <- nj(cdna.msa.1.d)

## we can look at that using plot:
plot( cdna.msa.1.nj )

## To get better labels for the nodes you can replace the row
## and column names, or you can modify the tip labels directly
## in the nj object. Look at:

names(cdna.msa.1.nj)

## and
cdna.msa.1.nj$tip.label

tmp  <- cdna.msa.1.nj
tmp$tip.label  <- cdna.sp[ cdna.msa.1.nj$tip.label ]

plot(tmp)

## to get an idea of how to do this.
```

You can see the trees that I got in figure 2. They look very bad indeed. There are many reasons for why that might be the case ranging from bugs in my code to bad choices of sequence ranges (eg. global alignment everywhere).

Additional stuff

To get distances we probably want to only consider part of the alignment. We can use the colmask attribute of the multiple alignment for this.

```r
## we can also use the automask option.
cdna.msa.1.masked <- maskGaps(cdna.msa.1, min.fraction=0.5, min.block.width=12)
colmask(cdna.msa.1.masked)

colmask(cdna.msa.1) <- IRanges(start=680, end=850)

dim(as.matrix(cdna.msa.1)) ## not affected by the mask
```

If you are interested in this you can read the documentation to `colmask` and related functions (`?colmask`). However, these are specific to `Biostrings` and will not be covered by here.

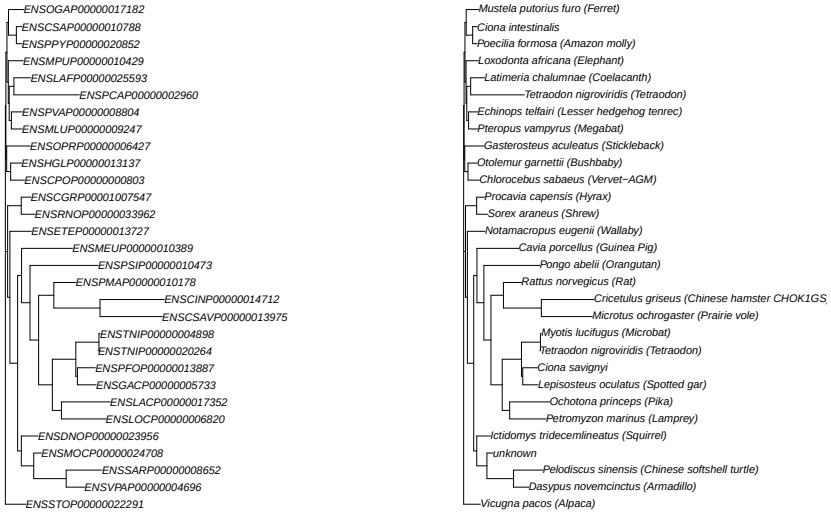

Figure 2: Rather bad looking multiple sequence alignments