A Little Book of R For Bioinformatics

Release 0.1

Avril Coghlan

Aug 18, 2017

Contents

By Avril Coghlan, Wellcome Trust Sanger Institute, Cambridge, U.K. Email: alc@sanger.ac.uk

This is a simple introduction to bioinformatics, with a focus on genome analysis, using the R statistics software.

To encourage research into neglected tropical diseases such as leprosy, Chagas disease, trachoma, schistosomiasis etc., most of the examples in this booklet are for analysis of the genomes of the organisms that cause these diseases.

There is a pdf version of this booklet available at: https://github.com/avrilcoghlan/LittleBookofRBioinformatics/raw/master/_build/latex/Bioinformatics.pdf.

If you like this booklet, you may also like to check out my booklet on using R for biomedical statistics, http://a-little-book-of-r-for-biomedical-statistics.readthedocs.org/, and my booklet on using R for time series analysis, http://a-little-book-of-r-for-time-series.readthedocs.org/.

Chapters in this Book

How to install R and a Brief Introduction to R

Introduction to R

This little booklet has some information on how to use R for bioinformatics.

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 simple programming.

Installing R

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

How to check if R is installed on a Windows PC

Before you install R on your computer, the first thing to do is to check whether R is already installed on your computer (for example, by a previous user).

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).

If you are using a Windows PC, there are two ways you can check whether R is already isntalled on your computer:

1. 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 step 2 instead.

2. Click on the "Start" menu at the bottom left of your Windows desktop, and then move your mouse over "All Programs" in the menu that pops up. See if "R" appears in the list of programs that pops up. If it does, it means that R is already installed on your computer, and you can start R by selecting "R" (or R X.X.X, where X.X.X gives the version of R, eg. R 2.10.0) from the list.

If either (1) or (2) above does succeed in starting R, it means that R is already installed on the computer that you are using. (If neither succeeds, R is not installed yet). If there is an old version of R installed on the Windows PC that you are using, it is worth installing the latest version of R, to make sure that you have all the latest R functions available to you to use.

Finding out what is the latest version of R

To find out what is the latest version of R, you can look at the CRAN (Comprehensive R Network) website, http://cran.r-project.org/.

Beside "The latest release" (about half way down the page), it will say something like "R-X.X.X.tar.gz" (eg. "R-2.12.1.tar.gz"). This means that the latest release of R is X.X.X (for example, 2.12.1).

New releases of R are made very regularly (approximately once a month), as R is actively being improved all the time. It is worthwhile installing new versions of R regularly, 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).

Installing R on a Windows PC

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

1. Go to http://ftp.heanet.ie/mirrors/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 2.10.1 for Windows" (or R X.X.X, where X.X.X gives the version of R, eg. R 2.11.1). Click on this link.

5. You may be asked if you want to save or run a file "R-2.10.1-win32.exe". Choose "Save" and save the file on the Desktop. Then double-click on the icon for the file to run it.

6. You will be asked what language to install it in - choose English.

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 "Information" at the top. Click "Next" again.

10. The next page says "Select Destination Location" at the top. By default, it will suggest to install R in "C:\Program Files" on your computer.

11. Click "Next" at the bottom of the R Setup wizard window.

12. The next page says "Select components" at the top. Click "Next" again.

13. The next page says "Startup options" at the top. Click "Next" again.

14. The next page says "Select start menu folder" at the top. Click "Next" again.

15. The next page says "Select additional tasks" at the top. Click "Next" again.

16. R should now be installed. This will take about a minute. When R has finished, you will see "Completing the R for Windows Setup Wizard" appear. Click "Finish".

17. To start R, you can either follow step 18, or 19:

18. 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 step 19 instead.

19. 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, eg. R 2.10.0) from the menu of programs.

20. The R console (a rectangle) should pop up:
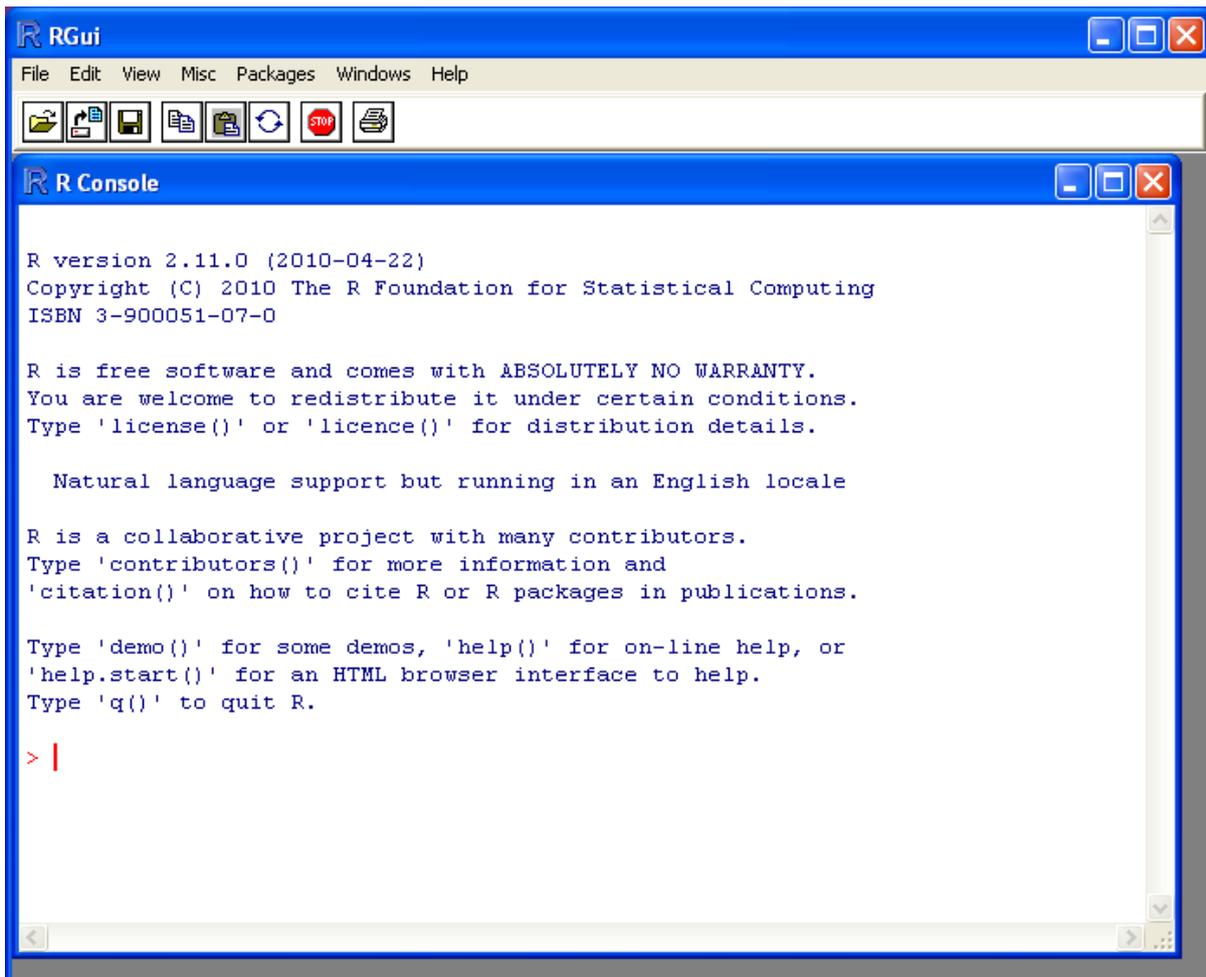
How to install R on non-Windows computers (eg. Macintosh or Linux computers)

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 http://ftp.heanet.ie/mirrors/cran.r-project.org and follow the R installation instructions for the appropriate operating system at http://ftp.heanet.ie/mirrors/cran.r-project.org/doc/FAQ/R-FAQ.html#How-can-R-be-installed_003f).

Installing R packages

R comes with some standard packages that are installed when you install R. However, in this booklet I will also tell you how to use some additional R packages that are useful, for example, the "rmeta" package. These additional packages do not come with the standard installation of R, so you need to install them yourself.

How to install an R package

Once you have installed R on a Windows computer (following the steps above), you can install an additional package by following the steps below:

1. To start R, follow either step 2 or 3:

2. 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 step 3 instead.

3. 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, eg. R 2.10.0) from the menu of programs.

4. The R console (a rectangle) should pop up.

5. Once you have started R, you can now install an R package (eg. the "rmeta" package) by choosing "Install package(s)" from the "Packages" menu at the top of the R console. This will ask you what website you want to download the package from, you should choose "Ireland" (or another country, if you prefer). It will also bring up a list of available packages that you can install, and you should choose the package that you want to install from that list (eg. "rmeta").

6. This will install the "rmeta" package.

7. The "rmeta" package is now installed. Whenever you want to use the "rmeta" package after this, after starting R, you first have to load the package by typing into the R console:

```
> library("rmeta")
```

Note that there are some additional R packages for bioinformatics that are part of a special set of R packages called Bioconductor (www.bioconductor.org) such as the "yeastExpData" R package, the "Biostrings" R package, etc.). These Bioconductor packages need to be installed using a different, Bioconductor-specific procedure (see *How to install a Bioconductor R package* below).

How to install a Bioconductor R package

The procedure above can be used to install the majority of R packages. However, the Bioconductor set of bioinformatics R packages need to be installed by a special procedure. Bioconductor (www.bioconductor.org) is a group of R packages that have been developed for bioinformatics. This includes R packages such as "yeastExpData", "Biostrings", etc.

To install the Bioconductor packages, follow these steps:

1. To start R, follow either step 2 or 3:

2. 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 step 3 instead.

3. 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, eg. R 2.10.0) from the menu of programs.

4. The R console (a rectangle) should pop up.

5. Once you have started R, now type in the R console:

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

6. This will install a core set of Bioconductor packages ("affy", "affydata", "affyPLM", "annaffy", "annotate", "Biobase", "Biostrings", "DynDoc", "gcrma", "genefilter", "geneplotter", "hgu95av2.db", "limma", "marray", "matchprobes", "multtest", "ROC", "vsn", "xtable", "affyQCReport"). This takes a few minutes (eg. 10 minutes).

7. At a later date, you may wish to install some extra Bioconductor packages that do not belong to the core set of Bioconductor packages. For example, to install the Bioconductor package called "yeastExpData", start R and type in the R console:

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

8. Whenever you want to use a package after installing it, you need to load it into R by typing:

```
> library("yeastExpData")
```

Running R

To use R, you first need to start the R program on your computer. You should have already installed R on your computer (see above).

To start R, you can either follow step 1 or 2: 1. 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 step 2 instead.

2. 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, eg. R 2.10.0) from the menu of programs.

This should bring up a new window, which is the *R console*.

A brief introduction to R

You will type R commands into the R console in order to carry out analyses in R. In the R console you will see:

```
>
```

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

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

```
> 2*3
[1] 6
> 10-3
[1] 7
```

All variables (scalars, vectors, matrices, etc.) created by R are called *objects*. In R, we assign values to variables using an arrow. 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, and the contents of that R object will be displayed:

```
> x
[1] 6
```

There are several possible different types of objects in R, including scalars, vectors, matrices, arrays, data frames, 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 (eg. numeric or characters), while lists may include elements such as characters as well as numeric quantities.

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)
```

To see the contents of the variable *myvector*, we can just type its name:

```
> myvector
[1]  8  6  9 10  5
```

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
```

In contrast to a vector, a list can contain elements of different types, for example, both numeric and character elements. A list can also 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:

```
> mylist <- list(name="Fred", wife="Mary", myvector)
```

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

```
> mylist
$name
[1] "Fred"

$wife
[1] "Mary"

[[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). Thus, we can extract the second and third elements from *mylist* by typing:

```
> mylist[[2]]
[1] "Mary"
> mylist[[3]]
[1]  8  6  9 10  5
```

Elements of lists may also be named, and in this case the elements may 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] "Mary"
```

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.

Another type of object that you will encounter in R is a *table* variable. For example, if we made a vector variable *mynames* containing the names of children in a class, we can use the table() function to produce a table variable that contains the number of children with each possible name:

```
> mynames <- c("Mary", "John", "Ann", "Sinead", "Joe", "Mary", "Jim", "John",
↪"Simon")
> table(mynames)
mynames
   Ann    Jim    Joe   John   Mary  Simon Sinead
     1      1      1      2      2      1      1
```

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

```
> mytable <- table(mynames)
```

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 *mytable* (the number of children called "John"), we type:

```
> mytable[[4]]
[1] 2
```

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

```
> mytable[["John"]]
[1] 2
```

Functions in R usually require *arguments*, which are input variables (ie. 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)
2
```

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 webpage will pop up with information about the function that you asked for help with.

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. The RSiteSearch() function 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 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")
Help files with alias or concept or title matching
'deviation' using fuzzy matching:

genefilter::rowSds
                Row variance and standard deviation of
                a numeric array
nlme::pooledSD      Extract Pooled Standard Deviation
stats::mad          Median Absolute Deviation
stats::sd           Standard Deviation
vsn::meanSdPlot     Plot row standard deviations versus row
```

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

In the example above, the help.search() function found a relevant function (sd() here). However, if you did not find what you were looking for with help.search(), you could then use the RSiteSearch() function to see if a search of all functions described on the R website may find something relevant to the topic that you're interested in:

```
> RSiteSearch("deviation")
```

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.

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* (ie. the average of 8, 6, 9, 10 and 5), we can use the mean() function:

```
> mean(myvector)
[1] 7.6
```

We have been using built-in R 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 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 (eg. 10, 25):

```
> myfunction(10)
[1] 120
> myfunction(25)
[1] 645
```

To quit R, type:

```
> q()
```

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.

Acknowledgements

For very helpful comments and suggestions for improvements on the installation instructions, thank you very much to Friedrich Leisch and Phil Spector.

Contact

I will be very grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Neglected Tropical diseases

Neglected tropical diseases are serious diseases that affect many people in tropical countries and which have been relatively little studied. The World Health Organisation lists the following as neglected tropical diseases:

trachoma, leprosy, schistosomiasis, soil transmitted helminths, lymphatic filariasis, onchocerciasis, Buruli ulcer, yaws, Chagas disease, African trypanosomiasis, leishmaniasis, Dengue fever, rabies, Dracunculiasis (guinea-worm disease), and Fascioliasis (see http://www.who.int/neglected_diseases/diseases/en/).

The genomes of many of the organisms that cause neglected tropical diseases have been fully sequenced, or are currently being sequenced, including:

- the bacterium Chlamydia trachomatis, which causes trachoma
- the bacterium Mycobacterium leprae, which causes leprosy
- the bacterium Mycobacterium ulcerans, which causes Buruli ulcer
- the bacterium Treponema pallidum subsp. pertenue, which causes yaws
- the protist Trypanosoma cruzi, which causes Chagas disease
- the protist Trypanosoma brucei, which causes African trypanosomiasis
- the protist Leishmania major, and related *Leishmania* species, which cause leishmaniasis
- the schistosome worm Schistosoma mansoni, which causes schistosomiasis
- the nematode worms Brugia malayi and Wuchereria bancrofti, which cause lymphatic filariasis
- the nematode worm Loa loa, which causes subcutaneous filariasis
- the nematode worm Onchocerca volvulus, which causes onchocerciasis
- the nematode worm Necator americanus, which causes soil-transmitted helminthiasis
- the Dengue virus, which causes Dengue fever
- the Rabies virus, which causes Rabies

To encourage research into these organisms, many of the examples in this booklet are based on analysing these genomes.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

DNA Sequence Statistics (1)

Using R for Bioinformatics

This booklet tells you how to use the R software to carry out some simple analyses that are common in bioinformatics. In particular, the focus is on computational analysis of biological sequence data such as genome sequences and protein sequences.

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

To use R, you first need to start the R program on your computer. You should have already installed R on your computer (if not, for instructions on how to install R, see How to install R).

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 several specialised *packages* for R. In this practical, you will learn to use the SeqinR package to retrieve sequences from a DNA sequence database, and to carry out simple analyses of DNA sequences.

Some well known bioinformatics packages for R are the Bioconductor set of R packages (www.bioconductor.org), which contains several packages with many R functions for analysing biological data sets such as microarray data; and the SeqinR package (pbil.univ-lyon1.fr/software/seqinr/home.php?lang=eng), which contains R functions for obtaining sequences from DNA and protein sequence databases, and for analysing DNA and protein sequences.

To use function from the SeqinR package, we first need to install the SeqinR package (for instructions on how to install an R package, see How to install an R package). Once you have installed the "SeqinR" R package, you can load the "SeqinR" R package by typing:

```
> library("seqinr")
```

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() function, you can type:

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

FASTA format

The FASTA format is a simple and widely used format for storing biological (DNA or protein) sequences. It was first used by the FASTA program for sequence alignment. It begins with a single-line description starting with a ">" character, followed by lines of sequences. Here is an example of a FASTA file:

```
> A06852 183 residues
MPRLFSYLLGVWLLLSQLPREIPGQSTNDFIKACGRELVRLWVEICGSVSWGRTALSLEE
PQLETGPPAETMPSSITKDAEILKMMLEFVPNLPQELKATLSERQPSLRELQQSASKDSN
LNFEEFKKIILNRQNEAEDKSLLELKNLGLDKHSRKKRLFRMTLSEKCCQVGCIRKDIAR
LC
```

The NCBI sequence database

The National Centre for Biotechnology Information (NCBI) (www.ncbi.nlm.nih.gov) in the US maintains a huge database of all the DNA and protein sequence data that has been collected, the NCBI Sequence Database. This also a similar database in Europe, the European Molecular Biology Laboratory (EMBL) Sequence Database (www.ebi.ac.uk/embl), and also a similar database in Japan, the DNA Data Bank of Japan (DDBJ; www.ddbj.nig.ac.jp). These three databases exchange data every night, so at any one point in time, they contain almost identical data.
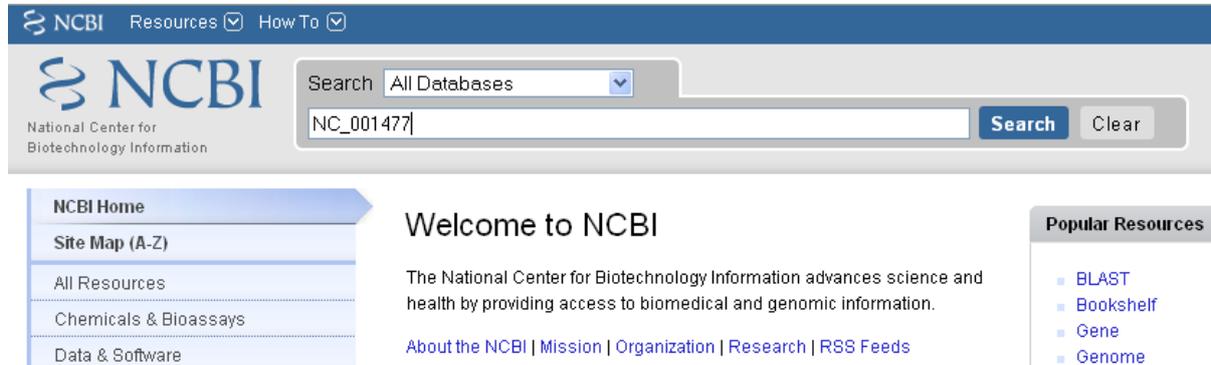
Each sequence in the NCBI Sequence Database is stored in a separate *record*, and is assigned a unique identifier that can be used to refer to that sequence record. The identifier is known as an *accession*, and consists of a mixture of numbers and letters. For example, Dengue virus causes Dengue fever, which is classified as a neglected tropical disease by the WHO. 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 and NC_002640, respectively.

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 will be present in all three databases, but it will have different accessions in each database, as they each use their own numbering systems for referring to their own sequence records.

Retrieving genome sequence data via the NCBI website

You can easily retrieve DNA or protein sequence data from the NCBI Sequence Database via its website www.ncbi.nlm.nih.gov.

The Dengue DEN-1 DNA sequence is a viral DNA sequence, and as mentioned above, its NCBI accession is NC_001477. To retrieve the DNA sequence for the Dengue DEN-1 virus from NCBI, go to the NCBI website, type "NC_001477" in the Search box at the top of the webpage, and press the "Search" button beside the Search box:

On the results page you will see the number of hits to "NC_001477" in each of the NCBI databases on the NCBI website. There are many databases on the NCBI website, for example, the "PubMed" data contains abstracts from scientific papers, the "Nucleotide" database contains DNA and RNA sequence data, the "Protein" data contains protein sequence data, and so on. The picture below shows what the results page should look like for your NC_001477 search. As you are looking for the DNA sequence of the Dengue DEN-1 virus genome, you expect to see a hit in the NCBI Nucleotide database, and indeed there is hit in the Nucleotide database (indicated by the "1" beside the icon for the Nucleotide database):

To look at the one sequence found in the Nucleotide database, you need to click on the icon for the NCBI Nucleotide database on the results page for the search:

When you click on the icon for the NCBI Nucleotide database, it will bring you to the record for NC_001477 in the NCBI Nucleotide database. This will contain the name and NCBI accession of the sequence, as well as other details such as any papers describing the sequence:

To retrieve the DNA sequence for the DEN-1 Dengue virus genome sequence as a FASTA format sequence file, click on "Send" at the top right of the NC_001477 sequence record webpage, and then choose "File" in the pop-up menu that appears, and then choose FASTA from the "Format" menu that appears, and click on "Create file".

A box will pop up asking you what to name the file, and where to save it. You should give it a sensible name (eg. "den1.fasta") and save it in a place where you will remember (eg. in the "My Documents" folder is a good idea):

You can now open the FASTA file containing the DEN-1 Dengue virus genome sequence using WordPad on your computer. To open WordPad, click on "Start" on the bottom left of your screen, click on "All Programs" in the menu that appears, and then select "Accessories" from the menu that appears next, and then select "WordPad" from the menu that appears next. WordPad should start up. In Wordpad, choose "Open" from the "File" menu. The WordPad "Open" dialog will appear. Set "Files of type" to "All Documents" at the bottom of the WordPad "Open" dialog. You should see a list of files, now select the file that contains the DEN-1 Dengue virus sequence

(eg. "den1.fasta"). The contents of the FASTA format file containing the Dengue DEN-1 sequence should now be displayed in WordPad:

Retrieving genome sequence data using SeqinR

Instead of going to the NCBI website to retrieve sequence data from the NCBI database, you can retrieve sequence data from NCBI directly from R, by using the SeqinR R package.

For example, you learnt above how to retrieve the DEN-1 Dengue virus genome sequence, which has NCBI accession NC_001477, from the NCBI website. To retrieve a sequence with a particular NCBI accession, you can use R function "getncbiseq()" below, which you will first need to copy and paste into R:

```
> getncbiseq <- function(accession)
  {
     require("seqinr") # this function requires the SeqinR R package
     # first find which ACNUC database the accession is stored in:
     dbs <- c("genbank","refseq","refseqViruses","bacterial")
     numdbs <- length(dbs)
     for (i in 1:numdbs)
     {
        db <- dbs[i]
        choosebank(db)
        # check if the sequence is in ACNUC database 'db':
        resquery <- try(query(".tmpquery", paste("AC=", accession)), silent = TRUE)
        if (!(inherits(resquery, "try-error")))
        {
           queryname <- "query2"
           thequery <- paste("AC=",accession,sep="")
           query(`queryname`,`thequery`)
```

```
            # see if a sequence was retrieved:
            seq <- getSequence(query2$req[[1]])
            closebank()
            return(seq)
        }
        closebank()
    }
    print(paste("ERROR: accession",accession,"was not found"))
}
```

Once you have copied and pasted the function getncbiseq() into R, you can use it to retrieve a sequence from the NCBI Nucleotide database, such as the sequence for the DEN-1 Dengue virus (accession NC_001477):

```
> dengueseq <- getncbiseq("NC_001477")
```

The variable *dengueseq* is a vector containing the nucleotide sequence. Each element of the vector contains one nucleotide of the sequence. Therefore, to print out a certain subsequence of the sequence, we just need to type the name of the vector *dengueseq* followed by the square brackets containing the indices for those nucleotides. For example, the following command prints out the first 50 nucleotides of the DEN-1 Dengue virus genome sequence:

```
> dengueseq[1:50]
[1] "a" "g" "t" "t" "g" "t" "t" "a" "g" "t" "c" "t" "a" "c" "g" "t" "g" "g" "a"
[20] "c" "c" "g" "a" "c" "a" "a" "g" "a" "a" "c" "a" "g" "t" "t" "t" "c" "g" "a"
[39] "a" "t" "c" "g" "g" "a" "a" "g" "c" "t" "t" "g"
```

Note that *dengueseq[1:50]* refers to the elements of the vector *dengueseq* with indices from 1-50. These elements contain the first 50 nucleotides of the DEN-1 Dengue virus sequence.

Writing sequence data out as a FASTA file

If you have retrieved a sequence from the NCBI database using the "getncbiseq()" function, you may want to save the sequence to a FASTA-format file on your computer, in case you need the sequence for further analyses (either in R or in other software).

You can write out a sequence to a FASTA-format file in R by using the "write.fasta()" function from the SeqinR R package. The write.fasta() function requires that you tell it the name of the output file using the "file.out" argument (input). You also need to specify the R variable that contains the sequence using the "sequences" argument, and the name that you want to give to the sequence using the "names" argument.

For example, if you have stored the DEN-1 Dengue virus sequence in a vector *dengueseq*, to write out the sequence to a FASTA-format file called "den1.fasta" that contains the sequence labelled as "DEN-1", you can type:

```
> write.fasta(names="DEN-1", sequences=dengueseq, file.out="den1.fasta")
```

Reading sequence data into R

Using the SeqinR package in R, you can easily read a DNA sequence from a FASTA file into R. For example, we described above how to retrieve the DEN-1 Dengue virus genome sequence from the NCBI database, or from R using the getncbiseq() function, and save it in a FASTA format file (eg. "den1.fasta").

You can read this FASTA format file into R using the read.fasta() function from the SeqinR R package:

```
> library("seqinr")
> dengue <- read.fasta(file = "den1.fasta")
```

Note that R expects the files that you read in (eg. "den1.fasta") to be in the "My Documents" folder on your computer, so if you stored "den1.fasta" somewhere else, you will have to move or copy it into "My Documents".

The command above reads the contents of the fasta format file den1.fasta into an R object called *dengue*. The variable *dengue* is an R list object. As explained above, a list is an R object that is like a vector, but can contain

elements that are numeric and/or contain characters. In this case, the list *dengue* contains information from the FASTA file that you have read in (ie. the name given to the dengue sequence in the FASTA file, and the DNA sequence itself). In fact, the first element of the list object *dengue* contains the the DNA sequence. As described above, we can access the elements of an R list object using double square brackets. Thus, we can store the DNA sequence for DEN-1 Dengue virus in a variable *dengueseq* by typing:

```
> dengueseq <- dengue[[1]]
```

Now the variable *dengueseq* is a vector containing the nucleotide sequence.

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*. To subsequently obtain the length of the genome sequence, we would use the length() function, typing:

```
> length(dengueseq)
[1] 10735
```

The length() function will give you back the length of the sequence stored in variable *dengueseq*, in nucleotides. The length() function actually gives the number of elements in the input vector that you pass to it, which in this case in the number of elements in the vector *dengueseq*. Since each element of the vector *dengueseq* 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).

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*, using the commands above), you would type:

```
> table(dengueseq)
dengueseq
   a    c    g    t
3426 2240 2770 2299
```

This means that the DEN-1 Dengue virus genome sequence has 3426 As, 2240 Cs, 2770 Gs and 2299 Ts.

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.

```
> GC(dengueseq)
[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%.

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". 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. For example, to find the number of occurrences of DNA words that are 1 nucleotide long in the sequence *dengueseq*, we type:

```
> count(dengueseq, 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:

```
> count(dengueseq, 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 <- count(dengueseq,1)
> denguetable[[3]]
 [1] 2770
```

The command above extracts the third element of the table produced by count(dengueseq,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[["g"]]
 [1] 2770
```

Summary

In this practical, you will have learnt to use the following R functions:

1. length() for finding the length of a vector or list
2. 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:

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

Links and Further Reading

Some links are included here for 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-lyon1.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).

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.

Acknowledgements

Thank you to Noel O'Boyle for helping in using Sphinx, http://sphinx.pocoo.org, to create this document, and github, https://github.com/, to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, to build and distribute this document.

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/).

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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).

Q1. What are the last twenty nucleotides of the Dengue virus genome sequence?

Q2. What is the length in nucleotides of the genome sequence for the bacterium *Mycobacterium leprae* strain TN (accession
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.

Q3. How many of each of the four nucleotides A, C, T and G, and any other symbols, are there in the *Mycobacterium leprae*
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.

Q4. What is the GC content of the *Mycobacterium leprae* TN genome sequence, when (i) all non-A/C/T/G nucleotides are inc
Hint: look at the help page for the GC() function to find out how it deals with non-A/C/T/G nucleotides.

Q5. How many of each of the four nucleotides A, C, T and G are there in the complement of the *Mycobacterium leprae* TN g
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?

Q6. How many occurrences of the DNA words CC, CG and GC occur in the *Mycobacterium leprae* TN genome sequence?

Q7. How many occurrences of the DNA words CC, CG and GC occur in the (i) first 1000 and (ii) last 1000 nucleotides of the
How can you check that the subsequence that you have looked at is 1000 nucleotides long?

DNA Sequence Statistics (2)

A little more introduction to R

In the chapter on How to install R, you learnt about variables in R, such as scalars, vectors, and lists. You also learnt how to use functions to 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*:

```
> x <- 100
> log10(x)
[1] 2
> myvector <- c(30,16,303,99,11,111)
> mean(myvector)
[1] 95
```

You also learnt that 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
```

A useful function in R is the seq() function, which can be used to create a sequence of numbers that run from a particular number to another particular number. For example, if we want to create the sequence of numbers from 1-100 in steps of 1 (ie. 1, 2, 3, 4, ... 97, 98, 99, 100), we can type:

```
> seq(1, 100, by = 1)
    1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

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 2 (ie. 1, 3, 5, 7, ... 97, 99), we can type:

```
> seq(1, 100, by = 2)
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
[26] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

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, if we want to print out the square of each number between 1 and 10, we can write the following for loop:

```
> for (i in 1:10) { print (i*i) }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
```

In the *for loop* above, the variable i is a counter 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 10. In the tenth cycle through the loop, the value of i is 10, and so $i * i$ =100 is printed out.

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 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 *avector* containing the numbers 2, 9, 100, and 133, and write a *for loop* to print out the square of each number in vector *avector*:

```
> avector <- c(2, 9, 100, 133)
> for (i in avector) { print (i*i) }
[1] 4
[1] 81
[1] 10000
[1] 17689
```

How can we use a *for loop* to print out the square of every second number between 1 and 10? The answer is to use the seq() function 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.

R allows the production of a variety of plots, including scatterplots, histograms, piecharts, and boxplots. For example, if you have two vectors of numbers *myvector1* and *myvector2*, you can plot a scatterplot of the values in *myvector1* against the values in *myvector2* using the plot() function. 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*:

```
> myvector1 <- c(10, 15, 22, 35, 43)
> myvector2 <- c(3, 3.2, 3.9, 4.1, 5.2)
> plot(myvector1, myvector2, xlab="myvector1", ylab="myvector2")
```
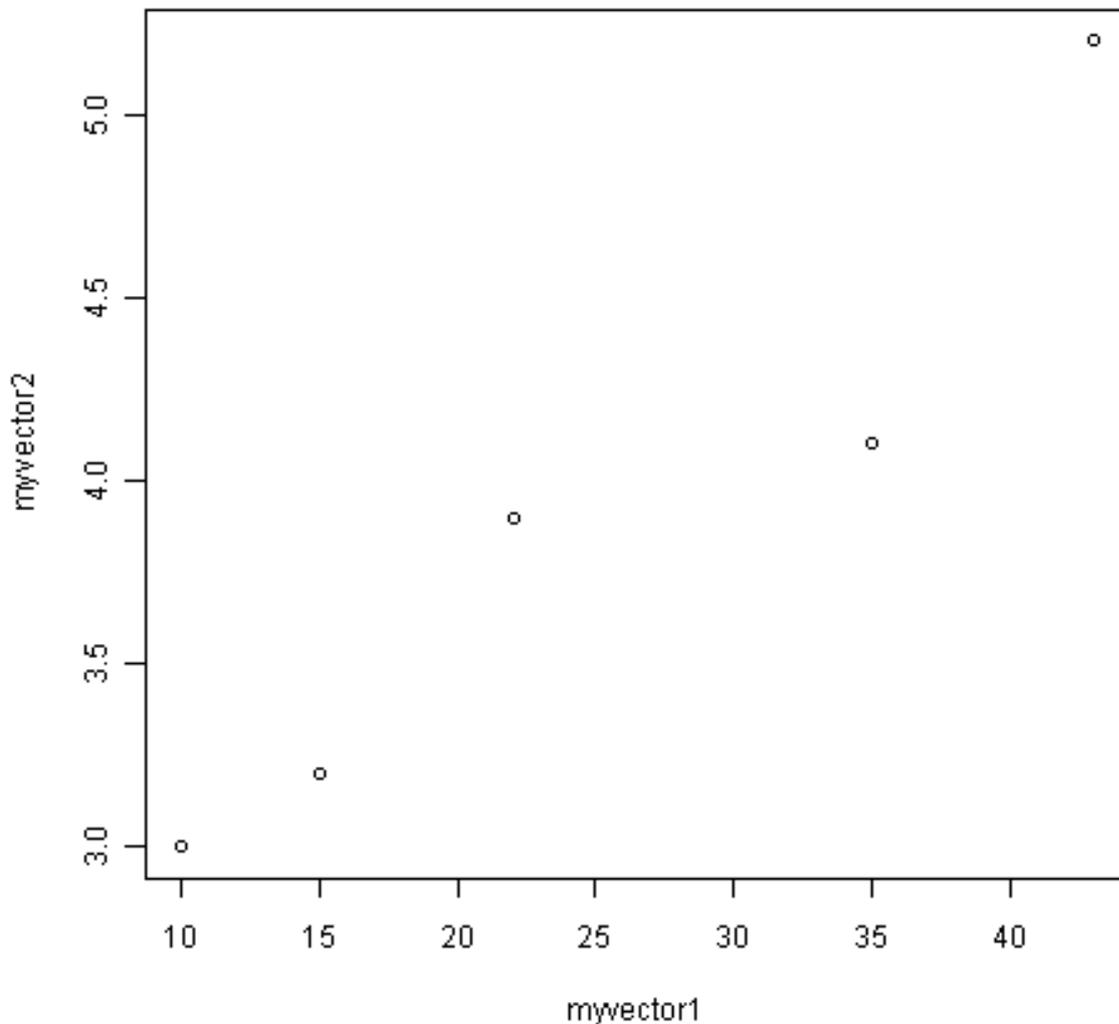

If you look at the help page for the plot() function, you will see that there are lots of optional arguments (inputs) that it can take that. For example, one optional argument is the *type* argument, that determines the type of the plot. By default, plot() will draw a dot at each data point, but if we set *type* to be "b", then it will also draw a line between each subsequent data point:

```
> plot(myvector1, myvector2, xlab="myvector1", ylab="myvector2", type="b")
```
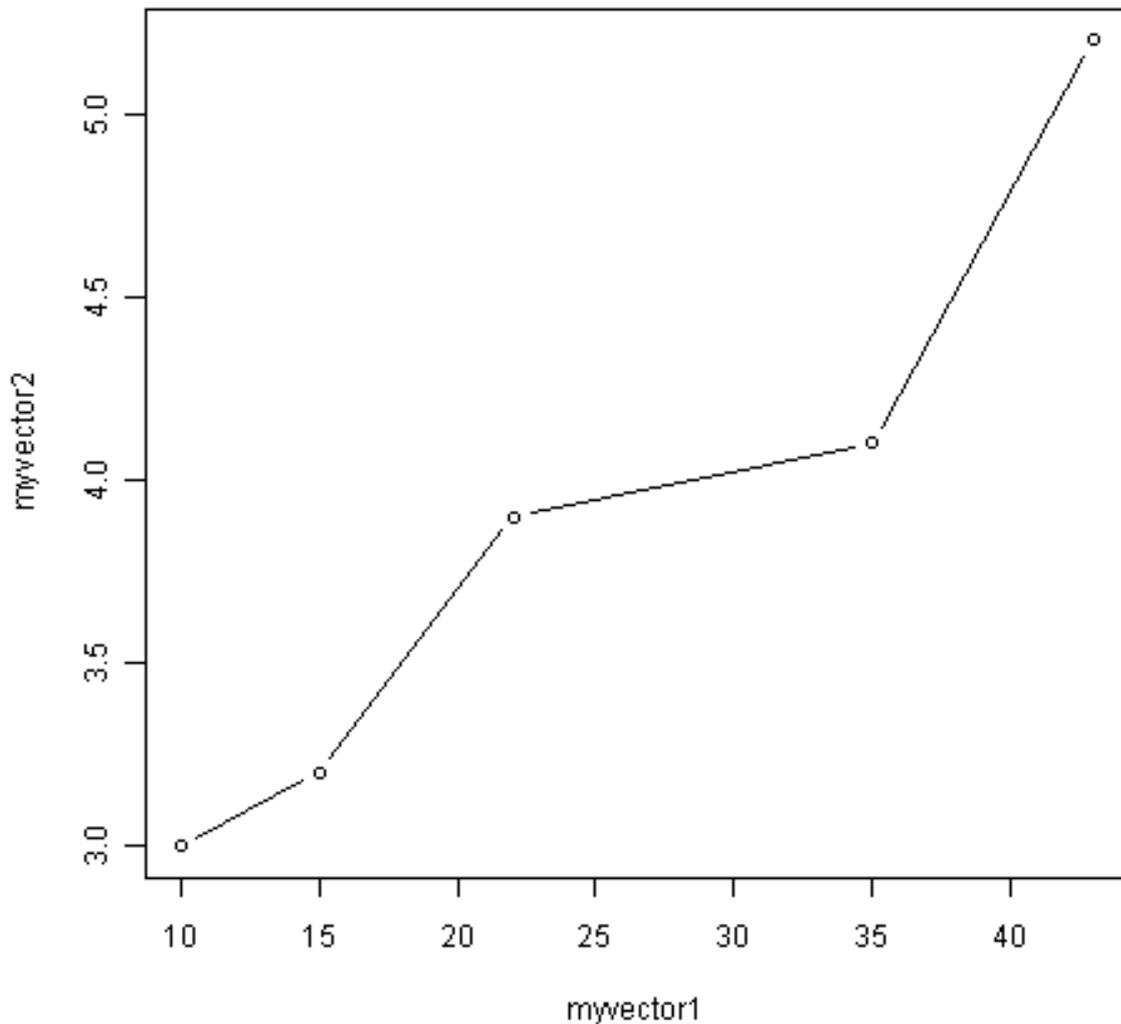
We have been using built-in R functions such as mean(), length(), print(), plot(), etc. We can also create our own functions in R to do calculations that you want to carry out very often on different input data sets. For example, we can create a function to calculate the value of 20 plus the square of some input number:

```
> 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 (eg. 10, 25):

```
> myfunction(10)
[1] 120
> 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":

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

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:

```
> x <- 100
> log10(x) # Finds the log to the base 10 of variable x.
[1] 2
```

Reading sequence data with SeqinR

In the previous chapter you learnt 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 or using the getncbiseq() function in R.

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. "den1.fasta").

Once you have downloaded the sequence data for a particular NCBI accession, and stored it on a file on your computer, you can then read it into R by using read.fasta function from the SeqinR R package. For example, if you have stored the DEN-1 Dengue virus sequence in a file "den1.fasta", you can type:

```
> library("seqinr")                        # Load the SeqinR package.
> dengue <- read.fasta(file = "den1.fasta")  # Read in the file "den1.fasta".
> dengueseq <- dengue[[1]]                 # Put the sequence in a vector
↪called "dengueseq".
```

Once you have retrieved a sequence from the NCBI Sequence Database and stored it in a vector variable such as *dengueseq* in the example above, it is possible to extract subsequence of the sequence by type the name of the vector (eg. *dengueseq*) 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[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"
```

Local variation in GC content

In the previous chapter, you learnt 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*, we can type:

```
> GC(dengueseq)
[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. 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.

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:

```
> GC(dengueseq[1:2000])        # Calculate the GC content of nucleotides 1-2000 of
↪the Dengue genome
[1] 0.465
> GC(dengueseq[2001:4000])     # Calculate the GC content of nucleotides 2001-4000
↪of the Dengue genome
[1] 0.4525
> GC(dengueseq[4001:6000])     # Calculate the GC content of nucleotides 4001-6000
↪of the Dengue genome
[1] 0.4705
> GC(dengueseq[6001:8000])     # Calculate the GC content of nucleotides 6001-8000
↪of the Dengue genome
[1] 0.479
> GC(dengueseq[8001:10000])    # Calculate the GC content of nucleotides 8001-10000
↪of the Dengue genome
[1] 0.4545
> GC(dengueseq[10001:10735])   # Calculate the GC content of nucleotides 10001-10735
↪of the Dengue genome
[1] 0.4993197
```

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:

```
> starts <- seq(1, length(dengueseq)-2000, by = 2000)
> starts
[1]    1 2001 4001 6001 8001
> n <- length(starts)    # Find the length of the vector "starts"
> for (i in 1:n) {
        chunk <- dengueseq[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)-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 examiniation 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.

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)-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[starts[i]:(starts[i]+1999)]
        chunkGC <- GC(chunk)
        print(chunkGC)
        chunkGCs[i] <- chunkGC
    }
> plot(starts,chunkGCs,type="b",xlab="Nucleotide start position",ylab="GC content")
```
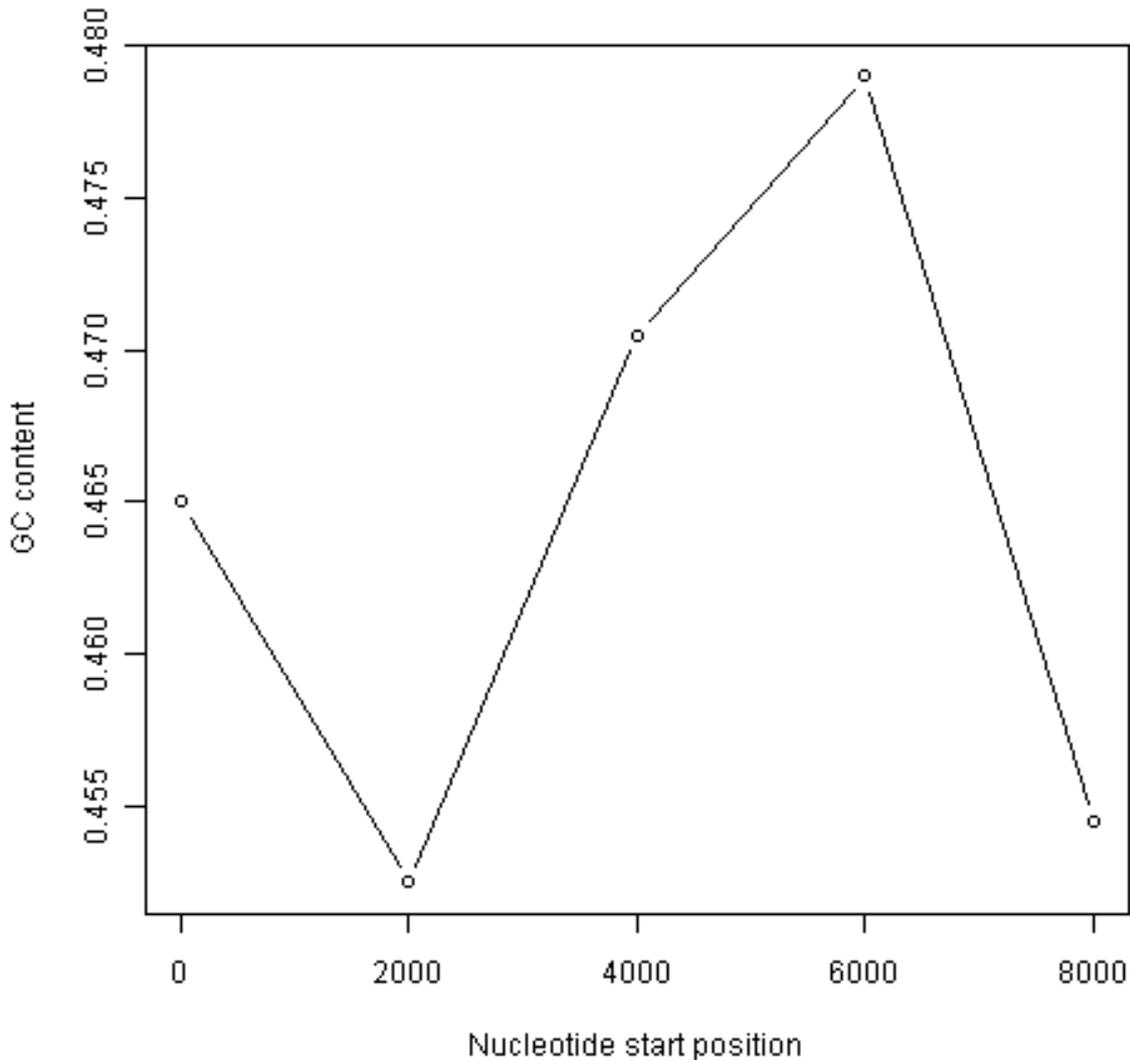
In the code above, the line "chunkGCs <- numeric(n)" makes a new vector *chunkGCs* which has the same number of elements as the vector *starts* (5 elements here). This vector *chunkGCs* is then used within the *for loop* for storing the GC content of each chunk of DNA.

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 windowsizes. Therefore, it makes sense to write a function to do the sliding window plot, that can take the windowsize that the user wants to use and the sequence that the user wants to study as arguments (inputs):

```
> slidingwindowplot <- function(windowsize, inputseq)
{
   starts <- seq(1, length(inputseq)-windowsize, by = windowsize)
   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 <- inputseq[starts[i]:(starts[i]+windowsize-1)]
        chunkGC <- GC(chunk)
        print(chunkGC)
        chunkGCs[i] <- chunkGC
```

```
   }
   plot(starts,chunkGCs,type="b",xlab="Nucleotide start position",ylab="GC content
↪")
}
```

This function will make a sliding window plot of GC content for a particular input sequence *inputseq* specified by the user, using a particular windowsize *windowsize* 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 windowsizes. For example, you could create two different sliding window plots of the DEN-1 Dengue virus genome sequence, using windowsizes of 3000 and 300 nucleotides, respectively:

```
> slidingwindowplot(3000, dengueseq)
```
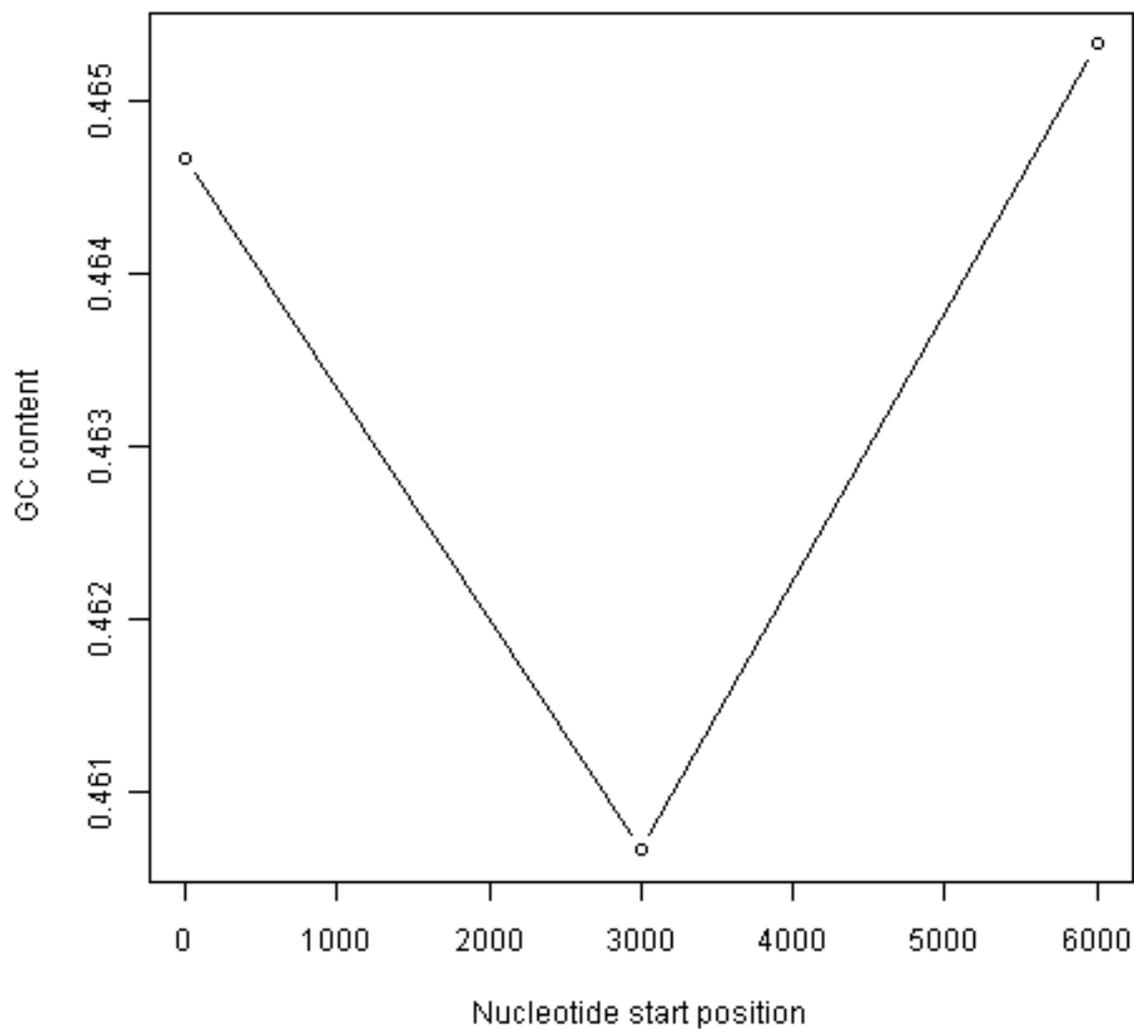


```
> slidingwindowplot(300, dengueseq)
```
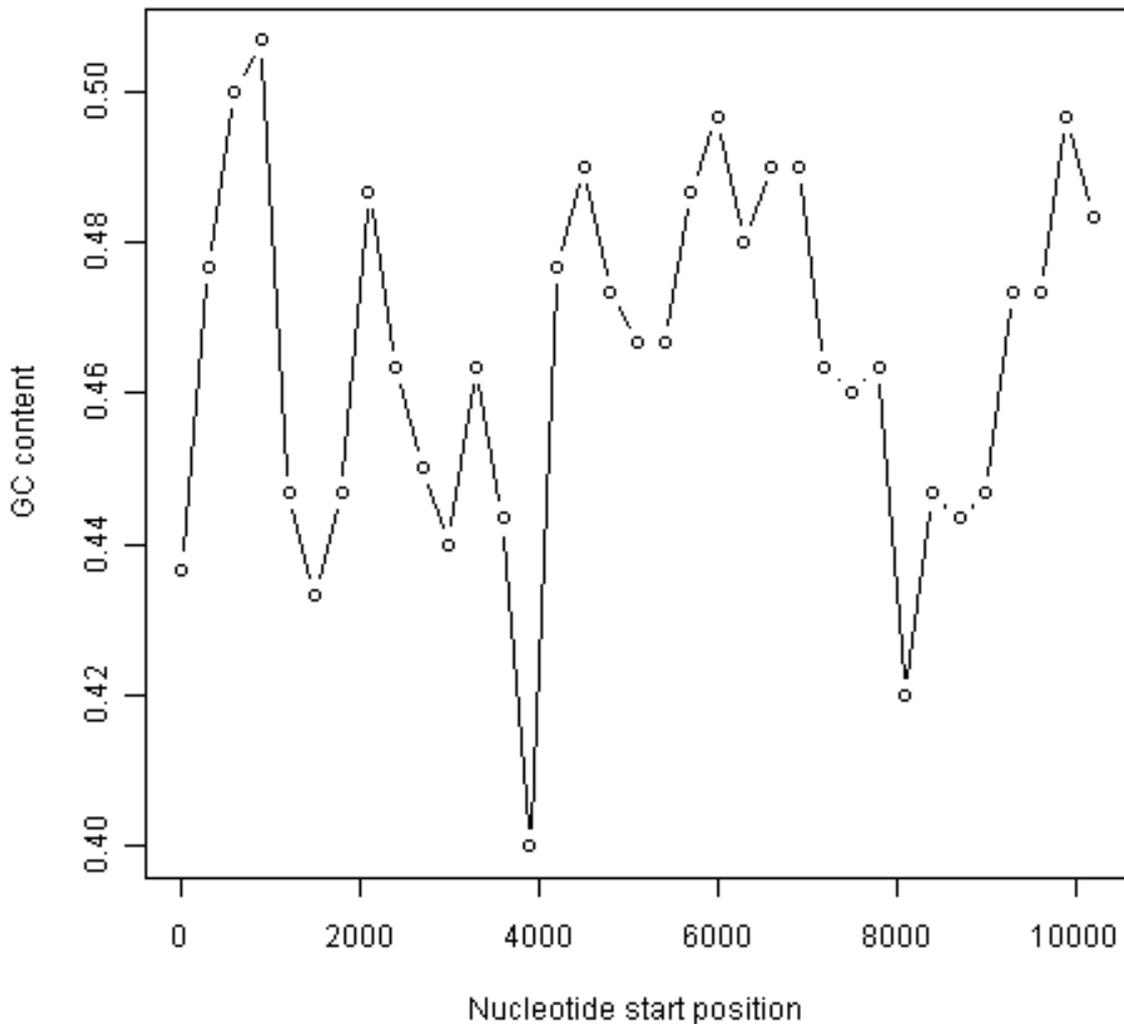
Over-represented and under-represented DNA words

In the previous chapter, you learnt 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:

```
> count(dengueseq, 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", ie. "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 ρ (Rho) is used to measure how over- or under-represented a particular DNA word is. For a 2-nucleotide (dinucleotide) DNA word ρ is calculated as:

$\rho(xy) = f_{xy}/(f_x*f_y),$

---

where f_{xy} and f_x are the frequencies of the DNA words xy and x in the DNA sequence under study. For example, the value of ρ for the DNA word "TA" can be calculated as: $\rho(TA) = f_{TA}/(f_T * f_A)$, where f_{TA}, f_T and f_A are the frequencies of the DNA words "TA", "T" and "A" in the DNA sequence.

The idea behind the ρ statistic is that, if a DNA sequence had a frequency f_x of a 1-nucleotide DNA word x, and a frequency f_y of a 1-nucleotide DNA word y, then we expect the frequency of the 2-nucleotide DNA word xy to be $f_x * f_y$. 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 ρ would be equal to 1. If we find that ρ 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 (ie. it is *over-represented*).

For example, say that your input sequence has only 5% Ts (ie. $f_T = 0.05$). 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 (ie. we expect $f_{TT} = f_T * f_T$). 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. $f_{TT} = 0.3$, so $\rho = 0.3/0.0025 = 120$), 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 ρ 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 ρ for the 2-nucleotide DNA word "GC", using the formula $\rho(GC) = f_{GC}/(f_G * f_C)$, where f_{GC}, f_G and f_C are the frequencies of the DNA words "GC", "G" and "C" in the DNA sequence:

```
> count(dengueseq, 1) # Get the number of occurrences of 1-nucleotide DNA words
    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, 2) # Get the number of occurrences of 2-nucleotide DNA words
   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 $\rho(GC)$ 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.

Summary

In this chapter, you will have learnt to use the following R functions:

1. seq() for creating a sequence of numbers

2. print() for printing out the value of a variable

3. plot() for making a plot (eg. a scatterplot)

4. numeric() for making a numeric vector of a particular length

5. function() for making a function

All of these functions belong to the standard installation of R. You also learnt how to use *for loops* to carry out the same operation again and again, each time on different inputs.

Links and Further Reading

Some links are included here for 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-lyon1.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).

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.

Acknowledgements

Thank you to Noel O'Boyle for helping in using Sphinx, http://sphinx.pocoo.org, to create this document, and github, https://github.com/, to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, to build and distribute this document.

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/).

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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 (2).

Q1. Draw a sliding window plot of GC content in the DEN-1 Dengue virus genome, using a window size of 200 nucleotides.
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?

Q2. Draw a sliding window plot of GC content in the genome sequence for the bacterium *Mycobacterium leprae* **strain TN (a**
Make a sketch of each plot that you draw. 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 windowsize (eg. 200 nucleotides) or a much larger windowsize (eg. 200,000 nucleotides)?

Q3. Write a function to calculate the AT content of a DNA sequence (ie. the fraction of the nucleotides in the sequence that a
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?

Q4. Write a function to draw a sliding window plot of AT content. Use it to make a sliding window plot of AT content along
Make a sketch of the plot that you draw.

Q5. Is the 3-nucleotide word GAC GC over-represented or under-represented in the *Mycobacterium leprae* **TN genome sequ**
What is the frequency of this word in the sequence? What is the expected frequency of this word in the sequence? What is the ρ (Rho) value for this word? How would you figure out whether there is already an R function to calculate ρ (Rho)? Is there one that you could use?

Sequence Databases

The NCBI Sequence Database

All published genome sequences are available over the internet, as it is a requirement of every scientific journal that any published DNA or RNA or protein sequence must be deposited in a public database. The main resources for storing and distributing sequence data are three large databases: the NCBI database (www.ncbi.nlm.nih.gov/), the European Molecular Biology Laboratory (EMBL) database (www.ebi.ac.uk/embl/, and the DNA Database of Japan (DDBJ) database (www.ddbj.nig.ac.jp/). These databases collect all publicly available DNA, RNA and protein sequence data and make it available for free. They exchange data nightly, so contain essentially the same data.

In this chapter we will discuss the NCBI database. Note however that it contains essentially the same data as in the EMBL/DDBJ databases.

Sequences in the NCBI Sequence Database (or EMBL/DDBJ) are identified by an accession number. This is a unique number that is only associated with one sequence. For example, the accession number NC_001477 is for the DEN-1 Dengue virus genome sequence. The accession number is what identifies the sequence. It is reported in scientific papers describing that sequence.

As well as the sequence itself, for each sequence the NCBI database (or EMBL/DDBJ databases) also stores some additional *annotation* data, such as the name of the species it comes from, references to publications describing that sequence, etc. Some of this annotation data was added by the person who sequenced a sequence and submitted it to the NCBI database, while some may have been added later by a human curator working for NCBI.

The NCBI database contains several sub-databases, the most important of which are:

- the NCBI Nucleotide database: contains DNA and RNA sequences
- the NCBI Protein database: contains protein sequences
- EST: contains ESTs (expressed sequence tags), which are short sequences derived from mRNAs
- the NCBI Genome database: contains DNA sequences for whole genomes
- PubMed: contains data on scientific publications

Searching for an accession number in the NCBI database

In the DNA Sequence Statistics chapter (1), you learnt how to obtain a FASTA file containing the DNA sequence corresponding to a particular accession number, eg. accession number NC_001477 (the DEN-1 Dengue virus genome sequence), either via the NCBI website or using the getncbiseq() function in R.

As explained in the DNA Sequence Statistics (1) chapter, the FASTA format is a file format commonly used to store sequence information. The first line starts with the character '>' followed by a name and/or description for the sequence. Subsequent lines contain the sequence itself.

```
>mysequence1
ACATGAGACAGACAGACCCCCAGAGACAGACCCCTAGACACAGAGAGAG
TATGCAGGACAGGGTTTTTGCCCAGGGTGGCAGTATG
```

A FASTA file can contain more than one sequence. If a FASTA file contains many sequences, then for each sequence it will have a header line starting with '>' followed by the sequence itself.

```
>mysequence1
ACATGAGACAGACAGACCCCCAGAGACAGACCCCTAGACACAGAGAGAG
TATGCAGGACAGGGTTTTTGCCCAGGGTGGCAGTATG
>mysequence2
AGGATTGAGGTATGGGTATGTTCCCGATTGAGTAGCCAGTATGAGCCAG
AGTTTTTTACAAGTATTTTTCCCAGTAGCCAGAGAGAGAGTCACCCAGT
ACAGAGAGC
```

NCBI Sequence Format (NCBI Format)

As mentioned above, for each sequence the NCBI database stores some extra information such as the species that it came from, publications describing the sequence, etc. This information is stored in the NCBI entry or NCBI record for the sequence. The NCBI entry for a sequence can be viewed by searching the NCBI database for the accession number for that sequence. The NCBI entries for sequences are stored in a particular format, known as NCBI format.

To view the NCBI entry for the DEN-1 Dengue virus (which has accession NC_001477), follow these steps:

1. Go to the NCBI website (www.ncbi.nlm.nih.gov).

2. Search for the accession number.

3. On the results page, if your sequence corresponds to a nucleotide (DNA or RNA) sequence, you should see a hit in the Nucleotide database, and you should click on the word 'Nucleotide' to view the NCBI entry for the hit. Likewise, if your sequence corresponds to a protein sequence, you should see a hit in the Protein database, and you should click on the word 'Protein' to view the NCBI entry for the hit.

4. After you click on 'Nucleotide' or 'Protein' in the previous step, the NCBI entry for the accession will appear.

For example, the NCBI entry for the DEN-1 Dengue virus genome sequence (NCBI accession NC_001477) looks like this:

The NCBI entry for an accession contains a lot of information about the sequence, such as papers describing it, features in the sequence, etc. The 'DEFINITION' field gives a short description for the sequence. The 'ORGAN-ISM' field in the NCBI entry identifies the species that the sequence came from. The 'REFERENCE' field contains scientific publications describing the sequence. The 'FEATURES' field contains information about the location of features of interest inside the sequence, such as regulatory sequences or genes that lie inside the sequence. The 'ORIGIN' field gives the sequence itself.

RefSeq

When carrying out searches of the NCBI database, it is important to bear in mind that the database may contain redundant sequences for the same gene that were sequenced by different laboratories (because many different labs have sequenced the gene, and submitted their sequences to the NCBI database).

There are also many different types of nucleotide sequences and protein sequences in the NCBI database. With respect to nucleotide sequences, some many be entire genomic DNA sequences, some may be mRNAs, and some may be lower quality sequences such as expressed sequence tags (ESTs, which are derived from parts of mRNAs), or DNA sequences of contigs from genome projects.

Furthermore, some sequences may be manually curated so that the associated entries contain extra information, but the majority of sequences are uncurated.

As mentioned above, the NCBI database often contains redundant information for a gene, contains sequences of varying quality, and contains both uncurated and curated data.

As a result, NCBI has made a special database called RefSeq (reference sequence database), which is a subset of the NCBI database. The data in RefSeq is manually curated, is high quality sequence data, and is non-redundant;

this means that each gene (or splice-form of a gene, in the case of eukaryotes), protein, or genome sequence is only represented once.

The data in RefSeq is curated and is of much higher quality than the rest of the NCBI Sequence Database. However, unfortunately, because of the high level of manual curation required, RefSeq does not cover all species, and is not comprehensive for the species that are covered so far.

You can easily tell that a sequence comes from RefSeq because its accession number starts with particular sequence of letters. That is, accessions of RefSeq sequences corresponding to protein records usually start with 'NP_', and accessions of RefSeq curated complete genome sequences usually start with 'NC_' or 'NS_'.

Querying the NCBI Database

You may need to interrogate the NCBI Database to find particular sequences or a set of sequences matching given criteria, such as:

- The sequence with accession NC_001477

- The sequences published in *Nature* **460**:352-358

- All sequences from *Chlamydia trachomatis*

- Sequences submitted by Matthew Berriman

- Flagellin or fibrinogen sequences

- The glutamine synthetase gene from *Mycobacteriuma leprae*

- The upstream control region of the *Mycobacterium leprae dnaA* gene

- The sequence of the *Mycobacterium leprae* DnaA protein

- The genome sequence of *Trypanosoma cruzi*

- All human nucleotide sequences associated with malaria

There are two main ways that you can query the NCBI database to find these sets of sequences. The first possibility is to carry out searches on the NCBI website. The second possiblity is to carry out searches from R.

Below, we will explain how to use both methods to carry out queries on the NCBI database. In general, the two methods should give the same result, but in some cases they do not, for various reasons, as shall be explained below.

Querying the NCBI Database via the NCBI Website

If you are carrying out searches on the NCBI website, to narrow down your searches to specific types of sequences or to specific organisms, you will need to use "search tags".

For example, the search tags "[PROP]" and "[ORGN]" let you restrict your search to a specific subset of the NCBI Sequence Database, or to sequences from a particular taxon, respectively. Here is a list of useful search tags, which we will explain how to use below:

Search tag	Example	Restricts your search to sequences:
[AC]	NC_001477[AC]	With a particular accession number
[ORGN]	Fungi[ORGN]	From a particular organism or taxon
[PROP]	biomol_mRNA[PROP]	Of a specific type (eg. mRNA) or from a specific database (eg. RefSeq)
[JOUR]	Nature[JOUR]	Described in a paper published in a particular journal
[VOL]	531[VOL]	Described in a paper published in a particular journal volume
[PAGE]	27[PAGE]	Described in a paper with a particular start-page in a journal
[AU]	"Smith J"[AU]	Described in a paper, or submitted to NCBI, by a particular author

To carry out searches of the NCBI database, you first need to go to the NCBI website, and type your search query into the search box at the top. For example, to search for all sequences from Fungi, you would type "Fungi[ORGN]" into the search box on the NCBI website.

You can combine the search tags above by using "AND", to make more complex searches. For example, to find all mRNA sequences from Fungi, you could type "Fungi[ORGN] AND biomol_mRNA[PROP]" in the search box on the NCBI website.

Likewise, you can also combine search tags by using "OR", for example, to search for all mRNA sequences from Fungi or Bacteria, you would type "(Fungi[ORGN] OR Bacteria[ORGN]) AND biomol_mRNA[PROP]" in the search box. Note that you need to put brackets around "Fungi[ORGN] OR Bacteria[ORGN]" to specify that the word "OR" refers to these two search tags.

Here are some examples of searches, some of them made by combining search terms using "AND":

Typed in the search box	Searches for sequences:
NC_001477[AC]	With accession number NC_001477
Nature[JOUR] AND 460[VOL] AND 352[PAGE]	Published in *Nature* **460**:352-358
"Chlamydia trachomatis"[ORGN]	From the bacterium *Chlamydia trachomatis*
"Berriman M"[AU]	Published in a paper, or submitted to NCBI, by M. Berriman
flagellin OR fibrinogen	Which contain the word 'flagellin' or 'fibrinogen' in their NCBI record
"Mycobacterium leprae"[ORGN] AND dnaA	Which are from *M. leprae*, and contain "dnaA" in their NCBI record
"Homo sapiens"[ORGN] AND "colon cancer"	Which are from human, and contain "colon cancer" in their NCBI record
"Homo sapiens"[ORGN] AND malaria	Which are from human, and contain "malaria" in their NCBI record
"Homo sapiens"[ORGN] AND biomol_mrna[PROP]	Which are mRNA sequences from human
"Bacteria"[ORGN] AND srcdb_refseq[PROP]	Which are RefSeq sequences from Bacteria
"colon cancer" AND srcdb_refseq[PROP]	From RefSeq, which contain "colon cancer" in their NCBI record

Note that if you are searching for a phrase such as "colon cancer" or "Chlamydia trachomatis, you need to put the phrase in inverted commas when typing it into the search box. This is because if you type the phrase in the search box without using inverted commas, the search will be for NCBI records that contain either of the two words 'colon' or 'cancer' (or either of the two words 'Chlamydia' or 'trachomatis'), not necessarily both words.

As mentioned above, the NCBI database contains several sub-databases, including the NCBI Nucleotide database and the NCBI Protein database. If you go to the NCBI website, and type one of the search queries above in the search box at the top of the page, the results page will tell you how many matching NCBI records were found in each of the NCBI sub-databases.

For example, if you search for "Chlamydia trachomatis[ORGN]", you will get matches to proteins from *C. trachomatis* in the NCBI Protein database, matches to DNA and RNA sequences from *C. trachomatis* in the NCBI Nucleotide database, matches to whole genome sequences for *C. trachomatis* strains in the NCBI Genome database, and so on:
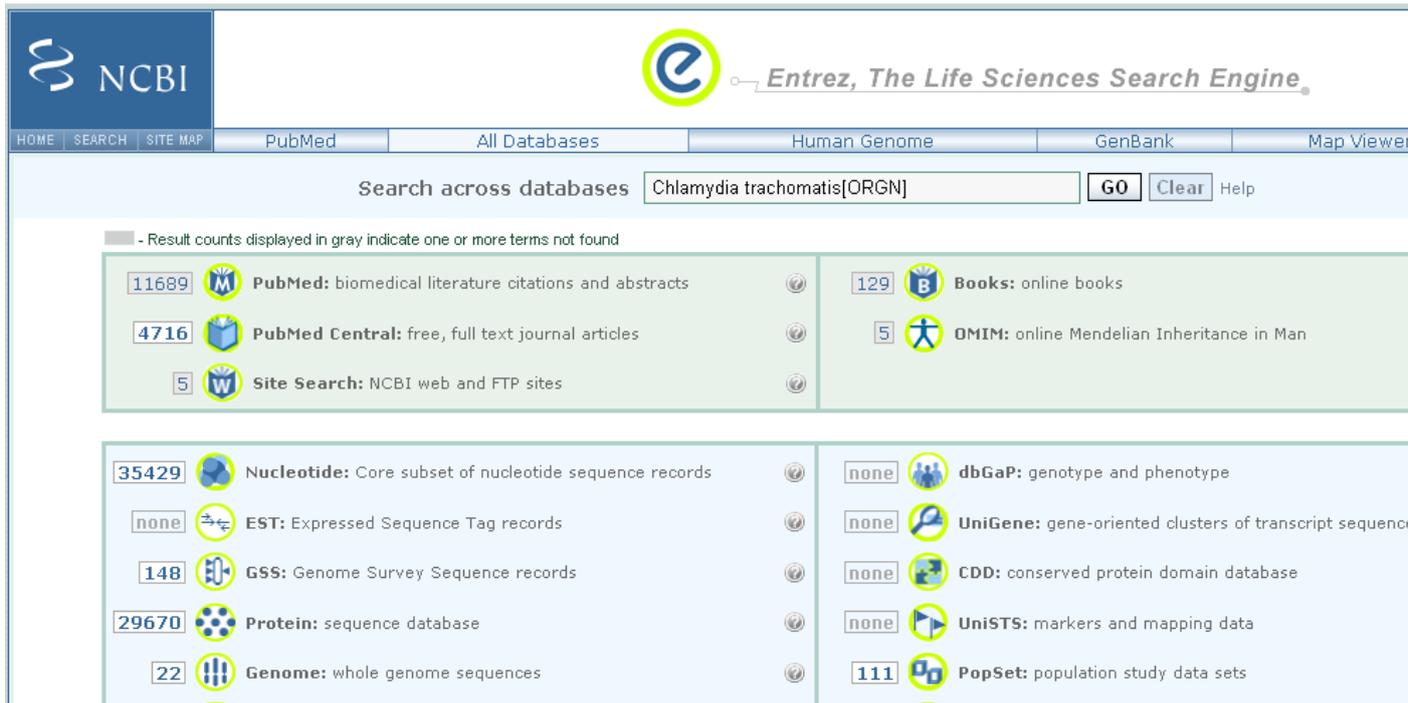
Alternatively, if you know in advance that you want to search a particular sub-database, for example, the NCBI Protein database, when you go to the NCBI website, you can select that sub-database from the drop-down list above the search box, so that you will search that sub-database:
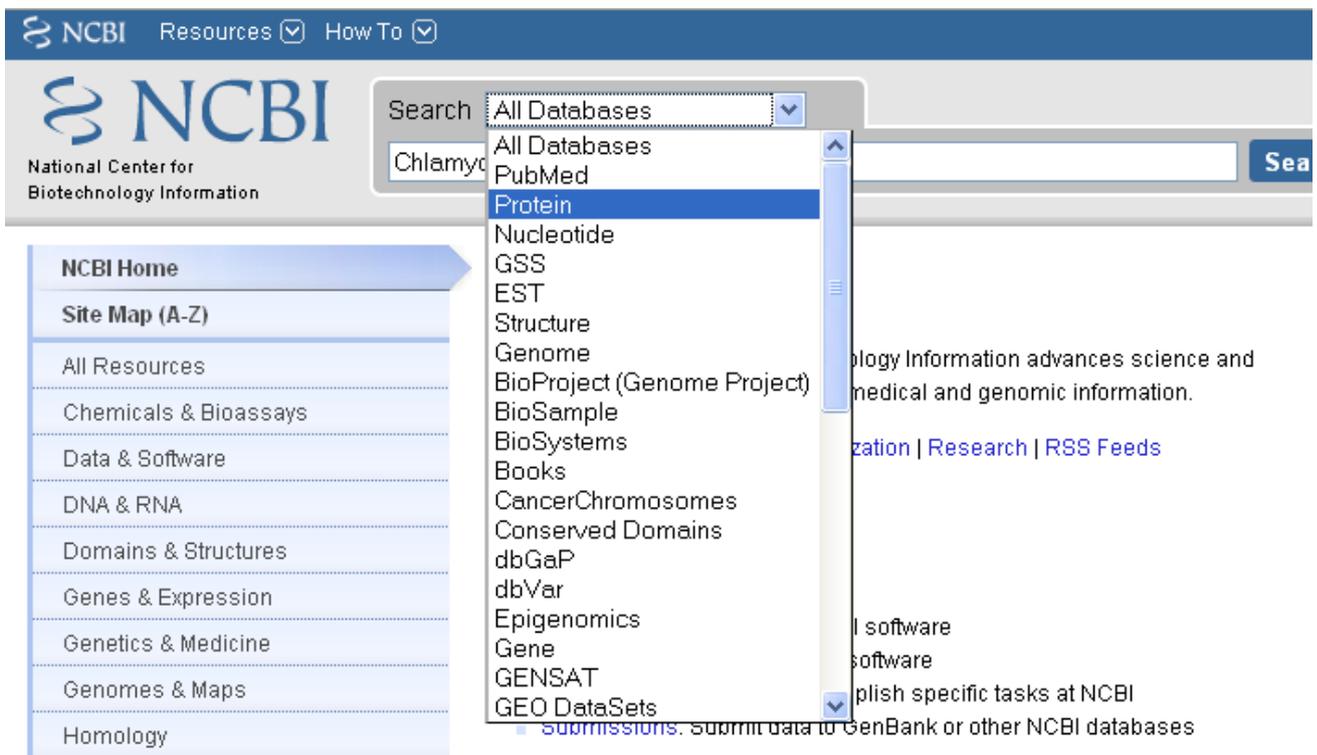
Example: finding the sequences published in *Nature* 460:352-358

For example, if you want to find sequences published in *Nature* **460**:352-358, you can use the "[JOUR]", "[VOL]" and "[PAGE]" search terms. That is, you would go to the NCBI website and type in the search box on the top: "Nature"[JOUR] AND 460[VOL] AND 352[PAGE], where [JOUR] specifies the journal name, [VOL] the volume of the journal the paper is in, and [PAGE] the page number.
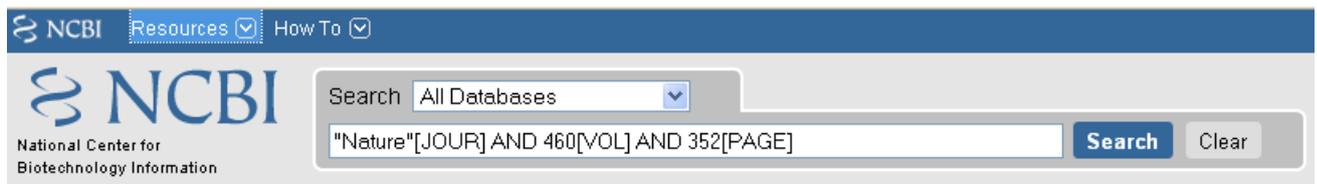
This should bring up a results page with "50890" beside the word "Nucleotide", and "1" beside the word "Genome", and "25701" beside the word "Protein", indicating that there were 50890 hits to sequence records in the Nucleotide database, which contains DNA and RNA sequences, and 1 hit to the Genome database, which contains genome sequences, and 25701 hits to the Protein database, which contains protein sequences:

If you click on the word "Nucleotide", it will bring up a webpage with a list of links to the NCBI sequence records for those 50890 hits. The 50890 hits are all contigs from the schistosome worm *Schistosoma mansoni*.

Likewise, if you click on the word "Protein", it will bring up a webpage with a list of links to the NCBI sequence records for the 25701 hits, and you will see that the hits are all predicted proteins for *Schistosoma mansoni*.

If you click on the word "Genome", it will bring you to the NCBI record for the *Schistosoma mansoni* genome sequence, which has NCBI accession NS_00200. Note that the accession starts with "NS_", which indicates that it is a RefSeq accession.

Therefore, in *Nature* volume 460, page 352, the *Schistosoma mansoni* genome sequence was published, along with all the DNA sequence contigs that were sequenced for the genome project, and all the predicted proteins for the gene predictions made in the genome sequence. You can view the original paper on the *Nature* website at http://www.nature.com/nature/journal/v460/n7253/abs/nature08160.html.

Note: *Schistmosoma mansoni* is a parasitic worm that is responsible for causing schistosomiasis, which is classified by the WHO as a neglected tropical disease.

Querying the NCBI Database via R

Instead of carrying out searches of the NCBI database on the NCBI website, you can carry out searches directly from R by using the SeqinR R package.

It is possible to use the SeqinR R package to retrieve sequences from these databases. The SeqinR package was written by the group that created the ACNUC database in Lyon, France (http://pbil.univ-lyon1.fr/databases/acnuc/acnuc.html). The ACNUC database is a database that contains most of the data from the NCBI Sequence Database, as well as data from other sequence databases such as UniProt and Ensembl.

An advantage of the ACNUC database is that it brings together data from various different sources, and makes it easy to search, for example, by using the SeqinR R package.

As will be explained below, the ACNUC database is organised into various different ACNUC (sub)-databases, which contain different parts of the NCBI database, and when you want to search the NCBI database via R, you will need to specify which ACNUC sub-database the NCBI data that you want to query is stored in.

To obtain a full list of the ACNUC sub-databases that you can access using SeqinR, you can use the "choosebank()" function from SeqinR:

```
> library("seqinr") # Load the SeqinR R package
> choosebank()      # List all the sub-databases in ACNUC
  [1] "genbank"      "embl"         "emblwgs"      "swissprot"
  [5] "ensembl"      "hogenom"      "hogenomdna"   "hovergendna"
  [9] "hovergen"     "hogenom4"     "hogenom4dna"  "homolens"
 [13] "homolensdna"  "hobacnucl"    "hobacprot"     "phever2"
 [17] "phever2dna"   "refseq"       "nrsub"         "greviews"
 [21] "bacterial"    "protozoan"    "ensbacteria"  "ensprotists"
 [25] "ensfungi"     "ensmetazoa"   "ensplants"    "mito"
 [29] "polymorphix"  "emglib"       "taxobacgen"   "refseqViruses"
```

Alas, the ACNUC sub-databases do not have a one-to-one correspondence with the NCBI sub-databases (the NCBI Protein database, NCBI EST database, NCBI Genome database, etc.)!

Three of the most important sub-databases in ACNUC which can be searched from R are:

- "genbank": this contains DNA and RNA sequences from the NCBI Sequence Database, except for certain classes of sequences (eg. draft genome sequence data from genome sequencing projects)

- "refseq": this contains DNA and RNA sequences from Refseq, the curated part of the NCBI Sequence Database

- "refseqViruses": this contains DNA, RNA and proteins sequences from viruses from RefSeq

You can find more information about what each of these ACNUC databases contains by looking at the ACNUC website.

You can carry out complex queries using the "query()" function from the SeqinR package. If you look at the help page for the query() function (by typing "help(query)", you will see that it allows you to specify criteria that you require the sequences to fulfill.

For example, to search for a sequence with a particular NCBI accession, you can use the "AC=" argument in "query()". The "query()" function will then search for sequences in the NCBI Sequence Database that match your criteria.

Just as you can use "AC=" to specify an accession in a search, you can specify that you want to find sequences whose NCBI records contain a certain keywords by using "K=" as an argument (input) to the "query()" function. Likewise you can limit a search to either DNA or mRNA sequences by using the "M=" argument for the "query()" function. Here are some more possible arguments you can use in the "query()" function:

Argument	Example	Restricts your search to sequences:
"AC="	"AC=NC_001477"	With a particular accession number
"SP="	"SP=Chlamydia"	From a particular organism or taxon
"M="	"M=mRNA"	Of a specific type (eg. mRNA)
"J="	"J=Nature"	Described in a paper published in a particular journal
"R="	"R=Nature/460/352"	Described in a paper in a particular journal, volume and start-page
"AU="	"AU=Smith"	Described in a paper, or submitted to NCBI, by a particular author

The full list of possible arguments for the "query()" funtion are given on its help page. Here are some examples using the query function:

Input to the query() function	Searches for sequences:
"AC=NC_001477"	With accession number NC_001477
"R=Nature/460/352"	Published in *Nature* **460**:352-358
"SP=Chlamydia trachomatis"	From the bacterium *Chlamydia trachomatis*
"AU=Berriman"	Published in a paper, or submitted to NCBI, by someone called Berriman
"K=flagellin OR K=fibrinogen"	Which have the keyword 'flagellin' or 'fibrinogen'
"SP=Mycobacterium leprae AND K=dnaA"	Which are from *M. leprae*, and have the keyword "dnaA"
"SP=Homo sapiens AND K=colon cancer"	Which are from human, and have the keyword "colon cancer"
"SP=Homo sapiens AND K=malaria"	Which are from human, and have the keyword "malaria"
"SP=Homo sapiens AND M=mrna"	Which are mRNA sequences from human
"SP=Bacteria"	Which are sequences from Bacteria

As explained above, the ACNUC database contains the NCBI sequence data organised into several sub-databases, and you can view the list of those sub-databases by using the "choosebank()" function from the SeqinR package. When you want to use "query()" to carry out a particular sub-database (eg. "genbank", which contains DNA and RNA sequences from the NCBI Sequence Database), you need to first specify the database that you want to search by using the "choosebank()" function, for example:

```
> choosebank("genbank") # Specify that we want to search the 'genbank' ACNUC sub-
↪database
```

Likewise, to specify that we want to search the 'refseq' ACNUC sub-database, which contains sequences from the NCBI RefSeq database, we would type:

```
> choosebank("refseq") # Specify that we want to search the 'refseq' ACNUC sub-
↪database
```

Once you have specified which ACNUC sub-database you want to search, you can carry out a search of that sub-database by using the "query()" function. You need to pass the "query()" function both a name for your query (which you can make up), and the query itself (which will be in the format of the examples in the table above). For example, if we want to search for RefSeq sequences from Bacteria, we might decide to call our query "RefSeqBact", and we would call the "query()" function as follows:

```
> query("RefSeqBact", "SP=Bacteria")
```

As explained below, the results of the search are stored in a list variable called "RefSeqBact", and can be retrieved from that list variable. The last thing to do once you have completed your search is to close the connection to the ACNUC sub-database that you were searching, by typing:

```
> closebank()
```

Thus, there are three steps involved in carrying out a query using SeqinR: first use "choosebank()" to select the ACNUC sub-database to search, secondly use "query()" to query the database, and thirdly use "closebank()" to close the connection to the ACNUC sub-database.

Another example could be to search for mRNA sequences from the parasitic worm *Schistosoma mansoni* in the NCBI Nucleotide database. The appropriate ACNUC sub-database to search is the "genbank" ACNUC sub-

database. We may decide to call our search "SchistosomamRNA". Therefore, to carry out the search, we type in R:

```
> choosebank("genbank")
> query("SchistosomamRNA", "SP=Schistosoma mansoni AND M=mrna")
> closebank()
```

Example: finding the sequence for the DEN-1 Dengue virus genome

Another example could be to search for the DEN-1 Dengue virus genome sequence, which has accession NC_001477. This is a viral genome sequence, and so should be in the ACNUC sub-database "refSeqViruses". Thus to search for this sequence, calling our search "Dengue1", we type in R:

```
> choosebank("refseqViruses")
> query("Dengue1", "AC=NC_001477")
```

The result of the search is now stored in the list variable *Dengue1*. Remember that a list is an R object that is like a vector, but can contain elements that are numeric and/or contain characters. In this case, the list *Dengue1* contains information on the NCBI records that match the query (ie. information on the NCBI record for accession NC_001477).

If you look at the help page for "query()", the details of the arguments are given under the heading "Arguments", and the details of the results (outputs) are given under the heading "Value". If you read this now, you will see that it tells us that the result of the "query()" function is a list with six different named elements, named "call", "name", "nelem", "typelist", "req", and "socket". The content of each of these six named elements is explained, for example, the "nelem" element contains the number of sequences that match the query, and the "req" element contains their accession numbers.

In our example, the list object *Dengue1* is an output of the "query()" function, and so has each of these six named elements, as we can find out by using the "attributes()" function, and looking at the named elements listed under the heading "$names":

```
> attributes(Dengue1)
  $names
  [1] "call"     "name"     "nelem"    "typelist" "req"      "socket"
  $class
  [1] "qaw"
```

As explained in the brief introduction to R, we can retrieve the value for each of the named elements in the list *Dengue1* by using "$", followed by the element's name, for example, to get the value of the element named "nelem" in the list *Dengue1*, we type:

```
> Dengue1$nelem
  [1] 1
```

This tells us that there was one sequence in the 'refseqViruses' ACNUC database that matched the query. This is what we would expect, as there should only be one sequence corresponding to accession NC_001477.

To obtain the accession numbers of the sequence found, we can type:

```
> Dengue1$req
  [[1]]
      name      length      frame      ncbicg
  "NC_001477"   "10735"       "0"        "1"
```

As expected, the accession number of the matching sequence is NC_001477.

When you type "attributes(Dengue1)" you can see that there are two headings, "$names", and "$class". As explained above, the named elements of the list variable *Dengue1* are listed under the heading "$names". In fact, the headings "$names" and "$class" are two *attributes* of the list variable *Dengue1*. We can retrieve the values of

the attributes of a variable using the "attr()" function. For example, to retrieve the value of the attribute "$names" of *Dengue1*, we type:

```
> attr(Dengue1, "names")
  [1] "call"    "name"    "nelem"   "typelist" "req"      "socket"
```

This gives us the value of the attribute "$names", which contains the the names of the named elements of the list variable *Dengue1*. Similarly, we can retrieve the value of the a attribute "$class" of *Dengue1*, we type:

```
> attr(Dengue1, "class")
  [1] "qaw"
```

This tells us that the value of the attribute "$class" is "qaw".

The final step in retrieving a genomic DNA sequence is to use the "getSequence()" function to tell R to retrieve the sequence data. The command below uses "getSequence()" to retrieve the sequence data for the DEN-1 Dengue virus genome, and puts the sequence into a variable *dengueseq*:

```
> dengueseq <- getSequence(Dengue1$req[[1]])
```

Note that the input to the getSequence() command is Dengue1$req[[1]], which contains the name of the NCBI record that the list *Dengue1* contains information about.

Once you have retrieved a sequence, you can then print it out. The variable *dengueseq* is a vector containing the nucleotide sequence. Each element of the vector contains one nucleotide of the sequence. Therefore, we can print out the first 50 nucleotides of the DEN-1 Dengue genome sequence by typing:

```
> dengueseq[1:50]
  [1] "a" "g" "t" "t" "g" "t" "t" "a" "g" "t" "c" "t" "a" "c" "g" "t" "g" "g" "a"
  [20] "c" "c" "g" "a" "c" "a" "a" "g" "a" "a" "c" "a" "g" "t" "t" "t" "c" "g" "a"
  [39] "a" "t" "c" "g" "g" "a" "a" "g" "c" "t" "t" "g"
```

Note that dengueseq[1:50] refers to the elements of the vector *dengueseq* with indices from 1-50. These elements contain the first 50 nucleotides of the DEN-1 Dengue virus genome sequence.

As well as retrieving the DNA (or RNA or protein) sequence itself, SeqinR can also retrieve all the *annotations* for the sequence, for example, information on when the sequence was sequenced, who sequenced it, what organism is it from, what paper was it described in, what genes were identified in the sequence, and so on.

Once you have retrieved a sequence using SeqinR, you can retrieved its annotations by using the "getAnnot()" function. For example, to view the annotations for the DEN-1 Dengue virus genome sequence, we type:

```
> annots <- getAnnot(Dengue1$req[[1]])
```

This stores the annotations information from the NCBI record for the DEN-1 Dengue virus sequence in a vector variable *annots*, with one line of the NCBI record in each element of the vector. Therefore, we can print out the first 20 lines of the NCBI record by typing:

```
> annots[1:20]
  [1] "LOCUS       NC_001477              10735 bp ss-RNA     linear   VRL 08-DEC-
→2008"
  [2] "DEFINITION  Dengue virus type 1, complete genome."
  [3] "ACCESSION   NC_001477"
  [4] "VERSION     NC_001477.1  GI:9626685"
  [5] "DBLINK      Project: 15306"
  [6] "KEYWORDS    ."
  [7] "SOURCE      Dengue virus 1"
  [8] "  ORGANISM  Dengue virus 1"
  [9] "            Viruses; ssRNA positive-strand viruses, no DNA stage;␣
→Flaviviridae;"
  [10] "            Flavivirus; Dengue virus group."
  [11] "REFERENCE   1  (bases 1 to 10735)"
  [12] "  AUTHORS   Puri,B., Nelson,W.M., Henchal,E.A., Hoke,C.H., Eckels,K.H.,"
```

```
[13] "            Dubois,D.R., Porter,K.R. and Hayes,C.G."
[14] "  TITLE     Molecular analysis of dengue virus attenuation after serial␣
↪passage"
[15] "            in primary dog kidney cells"
[16] "  JOURNAL   J. Gen. Virol. 78 (PT 9), 2287-2291 (1997)"
[17] "  PUBMED    9292016"
[18] "REFERENCE   2  (bases 1 to 10735)"
[19] "  AUTHORS   McKee,K.T. Jr., Bancroft,W.H., Eckels,K.H., Redfield,R.R.,"
[20] "            Summers,P.L. and Russell,P.K."
```

On the left of the annotations, you will see that there is a column containing the field name. For example, the line of the with "ACCESSION" in the left column is the accession field, which contains the accession for the sequence (NC_001477 for the DEN-1 Dengue virus).

The line with "ORGANISM" in the left column is the organism field, and usually contains the Latin name for the organism ("Dengue virus 1" here). The line with "AUTHORS" in the left column is the authors field, and contain the names of authors that wrote papers to describe the sequence and/or the names of the people who submitted the sequence to the NCBI Database.

When you have finished your running your query and getting the corresponding sequences and annotations, close the connection to the ACNUC sub-database:

```
> closebank()
```

Example: finding the sequences published in *Nature* 460:352-358

We described above how to search for the sequences published in *Nature* **460**:352-358, using the NCBI website. A second method is to use the SeqinR R package to search the ACNUC databases (which contain the NCBI sequence data) from R.

If you look at the help page the "query()" function, you see that you can query for sequences published in a particular paper using R=refcode, specifying the reference as refcode such as in jcode/volume/page (e.g., JMB/13/5432 or R=Nature/396/133). For the paper *Nature* **460**:352-358, we would need to use the refcode 'R=Nature/460/352'.

First we need to specify which of the ACNUC databases we want to search. For example, to specify that we want to search the "genbank" ACNUC database, which contains DNA and RNA sequences from the NCBI Nucleotide database, we type:

```
> choosebank("genbank") # Specify that we want to search the 'genbank' ACNUC sub-
↪database
```

We can then search the 'genbank' database for sequences that match a specific set of criteria by using the "query()" function. For example, to search for sequences that were published in *Nature* **460**:352-358, we type:

```
> query('naturepaper', 'R=Nature/460/352')
```

The line above tells R that we want to store the results of the query in an R list variable called *naturepaper*. To get the value of the element named "nelem" in the list *naturepaper*, we type:

```
> naturepaper$nelem
 [1] 19022
```

This tells us that there were 19022 sequences in the 'genbank' ACNUC database that matched the query. The 'genbank' ACNUC database contains DNA or RNA sequences from the NCBI Nucleotide database. Why don't we get the same number of sequences as found by carrying out the search on the NCBI website (where we found 50890 hits to the NCBI Nucleotide database)? The reason is that the ACNUC 'genbank' database does not contain all the sequences in the NCBI Nucleotide database, for example, it does not contain sequences that are in RefSeq or many short DNA sequences from sequencing projects.

To obtain the accession numbers of the first five of the 19022 sequences, we can type:

```
> accessions <- naturepaper$req
> accessions[1:5]
  [[1]]
        name       length        frame       ncbicg
   "FN357292"   "4179495"          "0"          "1"
  [[2]]
        name       length        frame       ncbicg
   "FN357293"   "2211188"          "0"          "1"
  [[3]]
        name       length        frame       ncbicg
   "FN357294"   "1818661"          "0"          "1"
  [[4]]
        name       length        frame       ncbicg
   "FN357295"   "2218116"          "0"          "1"
  [[5]]
        name       length        frame       ncbicg
   "FN357296"   "3831198"          "0"          "1"
```

This tells us that the NCBI accessions of the first five sequences (of the 19022 DNA or RNA sequences found that were published in *Nature* **460**:352-358) are FN357292, FN357293, FN357294, FN357295, and FN357296.

To retrieve these first five sequences, and print out the first 10 nucleotide bases of each sequence, we use the getSequence() command, typing:

```
> for (i in 1:5)
  {
      seqi <- getSequence(naturepaper$req[[i]])
      print(seqi[1:10])
  }
[1] "t" "t" "g" "t" "c" "g" "a" "t" "t" "a"
[1] "g" "g" "t" "c" "c" "t" "t" "a" "a" "g"
[1] "g" "c" "c" "t" "g" "a" "c" "c" "a" "t"
[1] "t" "a" "t" "t" "t" "c" "c" "a" "a" "t"
[1] "c" "a" "a" "t" "c" "a" "c" "t" "c" "a"
```

Note that the input to the getSequence() command is Dengue1$req[[i]], which contains the name of *i* th NCBI record that the list *naturepaper* contains information about.

Once we have carried out our queries and retrieved the sequences, the final step is to close the connection to the ACNUC sub-database that we searched ("genbank" here):

```
> closebank()
```

Saving sequence data in a FASTA-format file

Once you have retrieved a sequence, or set of sequences from the NCBI Database, using SeqinR, it is conveninent to save the sequences in a file in FASTA format. This can be done using the "write.fasta()" function in the SeqinR package, which was introduced in Chapter 1.

If you look at the help page for the "write.fasta()" function, you will see that as input it takes a list of vectors, where each vector contains one DNA, RNA or protein sequence.

For example, if you retrieve the sequences of human tRNAs from the NCBI Database by querying the ACNUC "genbank" sub-database, you can save the sequences in a FASTA format file called "humantRNAs.fasta" by typing:

```
> choosebank("genbank")                          # select the ACNUC sub-
↪database to be searched
> query("humtRNAs", "SP=homo sapiens AND M=TRNA")  # specify the query
> myseqs <- getSequence(humtRNAs)                # get the sequences
> mynames <- getName(humtRNAs)                   # get the names of the␣
↪sequences
```

```
> write.fasta(myseqs, mynames, file.out="humantRNAs.fasta")
> closebank()
```

In the above code, we get the sequences of the human tRNAs using the function "getSequence()" from the SeqinR package. We also use a function "getName()" from the SeqinR package to get the sequences' names. Then we use the "write.fasta()" function to write the sequences to a FASTA file "humantRNAs.fasta". The "write.fasta()" takes as arguments: the list *myseqs* containing the sequences, the list *mynames* containing the names of the sequences, and the name of the output file ("humantRNAs.fasta" here).

Finding the genome sequence for a particular species

Microbial genomes are generally smaller than eukaryotic genomes (*Escherichia coli* has about 5 million base pair in its genome, while the human genome is about 3 billion base pairs). Because they are considerably less expensive to sequence, many microbial genome sequencing projects have been completed.

If you don't know the accession number for a genome sequence (eg. for *Mycobacterium leprae*, the bacterium that causes leprosy), how can you find it out?

The easiest way to do this is to look at the NCBI Genome website, which lists all fully sequenced genomes and gives the accession numbers for the corresponding DNA sequences.

If you didn't know the accession number for the *Mycobacterium leprae* genome, you could find it on the NCBI Genome website by following these steps:

1. Go to the NCBI Genome website (http://www.ncbi.nlm.nih.gov/sites/entrez?db=Genome)

2. On the homepage of the NCBI Genome website, it gives links to the major subdivisions of the Genome database, which include Eukaryota, Prokaryota (Bacteria and Archaea), and Viruses. Click on 'Prokaryota', since *Mycobacterium leprae* is a bacterium. This will bring up a list of all fully sequenced bacterial genomes, with the corresponding accession numbers. Note that more than one genome (from various strains) may have been sequenced for a particular species.

3. Use 'Find' in the 'Edit' menu of your web browser to search for 'Mycobacterium leprae' on the webpage. You should find that the genomes of several different *M. leprae* strains have been sequenced. One of these is *M. leprae* TN, which has accession number NC_002677.

The list of sequenced genomes on the NCBI Genomes website is not a definitive list; that is, some sequenced genomes may be missing from this list. If you want to find out whether a particular genome has been sequenced, but you don't find it NCBI Genomes website's list, you should search for it by following these steps:

1. Go to the NCBI website (www.ncbi.nlm.nih.gov).

2. Select 'Genome' from the drop-down list above the search box.

3. Type the name of the species you are interested in in the search box (eg. **"Mycobacterium leprae"[ORGN]**). Press 'Search'.

Note that you could also have found the *Mycobacterium leprae* genome sequence by searching the NCBI Nucleotide database, as the NCBI Genome database is just a subset of the NCBI Nucleotide database.

How many genomes have been sequenced, or are being sequenced now?

On the NCBI Genome website (http://www.ncbi.nlm.nih.gov/sites/entrez?db=Genome), the front page gives a link to a list of all sequenced genomes in the groups Eukaryota, Prokaryota (Bacteria and Archaea) and Viruses. If you click on one of these links (eg. Prokaryota), at the top of the page it will give the number of sequenced genomes in that group (eg. number of sequenced prokaryotic genomes). For example, in this screenshot (from January 2011), we see that there were 1409 complete prokaryotic genomes (94 archaeal, 1315 bacterial):

Another useful website that lists genome sequencing projects is the Genomes OnLine Database (GOLD), which lists genomes that have been completely sequenced, or are currently being sequenced. To find the number of complete or ongoing bacterial sequencing projects, follow these steps:

1. Go to the GOLD website (http://genomesonline.org/).

2. Click on the yellow 'Enter GOLD' button in the centre of the webpage. On the subsequent page, it will give the number of ongoing bacterial, archaeal and eukaryotic genome sequencing projects.

3. Click on the 'Bacterial Ongoing' link to see the list of ongoing bacterial genome sequencing projects. By default, just the first 100 projects are listed, and the rest are listed on subsequent pages. In one of the columns of the page, this gives the university or institute that the genome was sequenced in. Other columns give the taxonomic information for the organism, and links to the sequence data.

4. Find the number of published genome sequencing projects. Go back one page, to the page with the 'Bacterial Ongoing' link. You will see that this page also lists the number of complete published genomes. To see a list of these genomes, click on 'Complete Published'. This will bring up a page that gives the number of published genomes at the top of the page. In one column of the page, this gives the university or institute that the genome was sequenced in.

As explained above, it is possible to identify genome sequence data in the NCBI Genome database. The GOLD database also gives some information about ongoing genome projects. Often, the GOLD database lists some ongoing projects that are not yet present in the NCBI Genome Database, because the sequence data has not yet been submitted to the NCBI Database. If you are interested in finding out how many genomes have been sequenced or are currently being sequenced for a particular species (eg. *Mycobacterium leprae*), it is a good idea to look at both the NCBI Genome database and at GOLD.

Summary

In this chapter, you have learnt how to retrieve sequences from the NCBI Sequence database, as well as to find out how many genomes have been sequenced or are currently being sequenced for a particular species.

Links and Further Reading

There is detailed information on how to search the NCBI database on the NCBI Help website at http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=helpentrez?part=EntrezHelp.

There is more information about the GOLD database in the paper describing GOLD by Liolios *et al*, which is available at http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2808860/?tool=pubmed.

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 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).

Acknowledgements

Thank you to Noel O'Boyle for helping in using Sphinx, http://sphinx.pocoo.org, to create this document, and github, https://github.com/, to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, to build and distribute this document.

Thank you to Andrew Lloyd and David Lynn, who generously shared their practical on sequence databases with me, which inspired many of the examples in this practical.

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Exercises

Answer the following questions. For each question, please record your answer, and what you did/typed to get this answer.

Model answers to the exercises are given in Answers to the exercises on Sequence Databases.

Q1. What information about the rabies virus sequence (NCBI accession NC_001542) can you obtain from its annotations in
What does it say in the DEFINITION and ORGANISM fields of its NCBI record? Note: rabies virus is the virus responsible for rabies, which is classified by the WHO as a neglected tropical disease.

Q2. How many nucleotide sequences are there from the bacterium *Chlamydia trachomatis* in the NCBI Sequence Database?
Note: the bacterium *Chlamydia trachomatis* is responsible for causing trachoma, which is classified by the WHO as a neglected tropical disease.

Q3. How many nucleotide sequences are there from the bacterium *Chlamydia trachomatis* in the *RefSeq* part of the NCBI Sequence Database?

Q4. How many nucleotide sequences were submitted to NCBI by Matthew Berriman?

Q5. How many nucleotide sequences from nematode worms are there in the RefSeq Database? Note that several parasitic nematode worms cause neglected tropical diseases, including *Brugia malayi* and *Wucheria bancrofti*, which cause lymphatic filariasis; *Loa loa*, which causes subcutaneous filariasis; *Onchocerca volvulus*, which causes onchocerciasis; and *Necator americanus*, which causes soil-transmitted helminthiasis.

Q6. How many nucleotide sequences for collagen genes from nematode worms are there in the NCBI Database?

Q7. How many *mRNA sequences* for collagen genes from nematode worms are there in the NCBI Database?

Q8. How many *protein sequences* for collagen proteins from nematode worms are there in the NCBI database?

Q9. What is the accession number for the *Trypanosoma cruzi* genome in NCBI? Do you see genome sequences for more than one strain of *Trypanosoma cruzi*? Note that the *Trypanosoma cruzi* causes Chagas disease, which is classified as a neglected tropical disease by the WHO.

Q10. How many fully sequenced nematode worm species are represented in the NCBI Genome database?

REVISION EXERCISES 1

These are some revision exercises on sequence statistics and sequence databases.

Exercises

Answer the following questions. For each question, please record your answer, and what you did/typed to get this answer.

Model answers to the exercises are given in Answers to Revision Exercises 1.

Q1.

What is the length of (total number of base-pairs in) the *Schistosoma mansoni* mitochondrial genome (NCBI accession NC_002545), and how many As, Cs, Gs and Ts does it contain?

> You must search for this sequence via the NCBI website, as it is not present in the ACNUC database. Note: *Schistmosoma mansoni* is a parasitic worm that is responsible for causing schistosomiasis, which is classified by the WHO as a neglected tropical disease.

Q2.

What is the length of the *Brugia malayi* mitochondrial genome (NCBI accession NC_004298), and how many As, Cs, Gs and Ts does it contain?

> You must search for this sequence via the NCBI website, as it is not present in the ACNUC database. Note: *Brugia malayi* is a parasitic worm responsible for causing lymphatic filariasis, which is classified by the WHO as a neglected tropical disease.

Q3.

What is the probability of the *Brugia malayi* mitochondrial genome sequence (NCBI accession NC_004298), according to a multinomial model in which the probabilities of As, Cs, Gs and Ts (pA, pC, pG, and pT) are set equal to the fraction of As, Cs, Gs and Ts in the *Schistosoma mansoni* mitochondrial genome?

Q4.

What are the top three most frequent 4-bp words (4-mers) in the genome of the bacterium *Chlamydia trachomatis* strain D/UW-3/CX (NCBI accession NC_000117), and how many times do they occur in its sequence?

> Note: *Chlamydia trachomatis* is a bacterium responsible for trachoma, which is classified by the WHO as a neglected tropical disease.

Q5.

Write an R function to generate a random DNA sequence that is *n* letters long (that is, *n* bases long) using a multinomial model in which the probabilities *pA*, *pC*, *pG*, and *pT* are set equal to the fraction of As, Cs, Gs and Ts in the *Schistosoma mansoni* mitochondrial genome (here *pA* stands for the probability of As, *pC* is the probability of Cs, etc.)

> Hint: look at the help page for the "sample()" function in R, as it might be useful to use within your R function.

Q6.

Give an example of using your function from Q5 to calculate a random sequence that is 20 letters long, using a multinomial model with *pA* =0.28, *pC* =0.21, *pG* =0.22, and *pT* =0.29.

Q7.

How many protein sequences from rabies virus are there in the NCBI Protein database? You must search for these sequences via the NCBI website, as it's not possible to do this search using SeqinR. Note: rabies virus is the virus responsible for rabies, which is classified by the WHO as a neglected tropical disease.

Q8.

What is the NCBI accession for the Mokola virus genome? Note: Mokola virus 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.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Pairwise Sequence Alignment

UniProt

In the previous chapter you learnt how to retrieve DNA and protein sequences from the NCBI database. The NCBI database is a key database in bioinformatics because it contains essentially all DNA sequences ever sequenced.

As mentioned in the previous chapter, a subsection of the NCBI database called RefSeq consists of high quality DNA and protein sequence data. Furthermore, the NCBI entries for the RefSeq sequences have been *manually curated*, which means that biologists employed by NCBI have added additional information to the NCBI entries for those sequences, such as details of scientific papers that describe the sequences.

Another extremely important manually curated database is UniProt (www.uniprot.org), which focuses on protein sequences. UniProt aims to contains manually curated information on all known protein sequences. While many of the protein sequences in UniProt are also present in RefSeq, the amount and quality of manually curated information in UniProt is much higher than that in RefSeq.

For each protein in UniProt, the UniProt curators read all the scientific papers that they can find about that protein, and add information from those papers to the protein's UniProt entry. For example, for a human protein, the UniProt entry for the protein usually includes information about the biological function of the protein, in what human tissues it is expressed, whether it interacts with other human proteins, and much more. All this information has been manually gathered by the UniProt curators from scientific papers, and the papers in which the found the information are always listed in the UniProt entry for the protein.

Just like NCBI, UniProt also assigns an *accession* to each sequence in the UniProt database. Although the same protein sequence may appear in both the NCBI database and the UniProt database, it will have different NCBI and UniProt accessions. However, there is usually a link on the NCBI entry for the protein sequence to the UniProt entry, and vice versa.

Viewing the UniProt webpage for a protein sequence

If you are given the UniProt accession for a protein, to find the UniProt entry for the protein, you first need to go the UniProt website, www.uniprot.org. At the top of the UniProt website, you will see a search box, and you can type the accession of the protein that you are looking for in this search box, and then click on the "Search" button to search for it.

For example, if you want to find the sequence for the chorismate lyase protein from *Mycobacterium leprae* (the bacterium which causes leprosy), which has UniProt accession Q9CD83, you would type just "Q9CD83" in the search box and press "Search":

The UniProt entry for UniProt accession Q9CD83 will then appear in your web browser. The picture below shows the top part of the UniProt entry for accession Q9CD83. You can see there is a lot of information about the protein in its UniProt entry.

Beside the heading "Organism" you can see the organism is given as *Mycobacterium leprae*. Beside the heading "Taxonomic lineage", you can see "Bacteria > Actinobacteria > Actinobacteridae > Actinomycetales > Corynebacterineae > Mycobacteriaceae > Mycobacterium".

This tells us that *Mycobacterium* is a species of bacteria, which belongs to a group of related bacteria called the Mycobacteriaceae, which itself belongs to a larger group of related bacteria called the Corynebacterineae, which itself belongs to an even larger group of related bacteria called the Actinomycetales, which itself belongs to the Actinobacteridae, which itself belongs to a huge group of bacteria called the Actinobacteria.

Beside the heading "Sequence length" we see that the sequence is 210 amino acids long (210 letters long). Further down, beside the heading "Function", it says that the function of this protein is that it "Removes the pyruvyl group from chorismate to provide 4-hydroxybenzoate (4HB)". This tells us this protein is an enzyme (a protein that increases the rate of a specific biochemical reaction), and tells us what is the particular biochemical reaction that this enzyme is involved in.

Further down the UniProt page for this protein, you will see a lot more information, as well as many links to webpages in other biological databases, such as NCBI. The huge amount of information about proteins in UniProt means that if you want to find out about a particular protein, the UniProt page for that protein is a great place to start.

Names and origin

Protein names	*Recommended name:* **Chorismate--pyruvate lyase** EC=4.1.3.- *Alternative name(s):* 4-HB synthase p-hydroxybenzoic acid synthase
Gene names	Ordered Locus Names:ML0133
Organism	**Mycobacterium leprae** [Complete proteome] [HAMAP]
Taxonomic identifier	1769 [NCBI]
Taxonomic lineage	Bacteria › Actinobacteria › Actinobacteridae › Actinomycetales › Corynebacterineae › Mycobacteriace

Protein attributes

Sequence length	210 AA.
Sequence status	Complete.
Protein existence	Inferred from homology.

General annotation (Comments)

Function	Removes the pyruvyl group from chorismate to provide 4-hydroxybenzoate (4HB). Involved in the synth (p-HBADs) and phenolic glycolipids (PGL) that play important roles in the pathogenesis of mycobacte
Catalytic activity	Chorismate = 4-hydroxybenzoate + pyruvate.
Sequence similarities	Belongs to the chorismate--pyruvate lyase type 2 family.

Retrieving a UniProt protein sequence via the UniProt website

To retrieve a FASTA-format file containing the sequence for a particular protein, you need to look at the top right of the UniProt entry for the protein on the UniProt website.

You will see a small orange button labelled "FASTA", which you should click on:

UniProt › UniProtKB					Downloads · Contact · Documentation/Help
Search	Blast *	Align	Retrieve	ID Mapping *	

Search in
Protein Knowledgebase (UniProtKB) ▾   **Query**   [Search] [Clear] Advanced Search »

Q9CD83 (PHBS_MYCLE) ⭐ Reviewed, UniProtKB/Swiss-Prot
Last modified February 8, 2011. Version 37. 🔊 History...

⊡ Clusters with 100%, 90%, 50% identity | 📄 Documents (1) | 🗐 Third-party data

┌Contribute─────────
💬 Send feedback
💬 Read comments (0) or add your own

text xml rdf/xml gff fasta

The FASTA-format sequence for the accession will now appear in your web browser. To save it as a file, go to the "File" menu of your web browser, choose "Save page as", and save the file. Remember to give the file a sensible name (eg. "Q9CD83.fasta" for accession Q9CD83), and in a place that you will remember (eg. in the "My Documents" folder).

For example, you can retrieve the protein sequences for the chorismate lyase protein from *Mycobacterium leprae* (which has UniProt accession Q9CD83) and for the chorismate lyase protein from *Mycobacterium ulcerans* (UniProt accession A0PQ23), and save them as FASTA-format files (eg. "Q9CD83.fasta" and "A0PQ23.fasta", as described above.

Note that *Mycobacterium leprae* is the bacterium which causes leprosy, while *Mycobacterium ulcerans* is a related bacterium which causes Buruli ulcer, both of which are classified by the WHO as neglected tropical diseases.

Note that the *M. leprae* and *M. ulcerans* chorismate lyase proteins are an example of a pair of homologous (related) proteins in two related species of bacteria.

Once you have downloaded the protein sequences for UniProt accessions Q9CD83 and A0PQ23 and saved them as FASTA-format files (eg. "Q9CD83.fasta" and "A0PQ23.fasta"), you can read them into R using the read.fasta() function in the SeqinR R package (as described in chapter 1).

Remember that the read.fasta() function expects that you have put your FASTA-format files in the "My Documents" folder on your computer.

For example, the following commands will read the FASTA-format files Q9CD83.fasta and A0PQ23.fasta into R, and store the two protein sequences in two vectors *lepraeseq* and *ulceransseq*:

```
> library("seqinr")
> leprae <- read.fasta(file = "Q9CD83.fasta")
> ulcerans <- read.fasta(file = "A0PQ23.fasta")
> lepraeseq <- leprae[[1]]
> ulceransseq <- ulcerans[[1]]
> lepraeseq # Display the contents of the vector "lepraeseq"
  [1] "m" "t" "n" "r" "t" "l" "s" "r" "e" "e" "i" "r" "k" "l" "d" "r" "d" "l"
 [19] "r" "i" "l" "v" "a" "t" "n" "g" "t" "l" "t" "r" "v" "l" "n" "v" "v" "a"
 [37] "n" "e" "e" "i" "v" "v" "d" "i" "i" "n" "q" "q" "l" "l" "d" "v" "a" "p"
 [55] "k" "i" "p" "e" "l" "e" "n" "l" "k" "i" "g" "r" "i" "l" "q" "r" "d" "i"
 [73] "l" "l" "k" "g" "q" "k" "s" "g" "i" "l" "f" "v" "a" "a" "e" "s" "l" "i"
 [91] "v" "i" "d" "l" "l" "p" "t" "a" "i" "t" "t" "y" "l" "t" "k" "t" "h" "h"
[109] "p" "i" "g" "e" "i" "m" "a" "a" "s" "r" "i" "e" "t" "y" "k" "e" "d" "a"
[127] "q" "v" "w" "i" "g" "d" "l" "p" "c" "w" "l" "a" "d" "y" "g" "y" "w" "d"
[145] "l" "p" "k" "r" "a" "v" "g" "r" "r" "y" "r" "i" "i" "a" "g" "g" "q" "p"
[163] "v" "i" "i" "t" "t" "e" "y" "f" "l" "r" "s" "v" "f" "q" "d" "t" "p" "r"
[181] "e" "e" "l" "d" "r" "c" "q" "y" "s" "n" "d" "i" "d" "t" "r" "s" "g" "d"
[199] "r" "f" "v" "l" "h" "g" "r" "v" "f" "k" "n" "l"
```

Retrieving a UniProt protein sequence using SeqinR

An alternative method of retrieving a UniProt protein sequence is to use the SeqinR package to query the ACNUC sub-database "swissprot", which contains protein sequences from UniProt.

We use the query() function from SeqinR to query this database, as described in chapter3.

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

```
> library("seqinr")
> choosebank("swissprot")
> query("leprae", "AC=Q9CD83")
> lepraeseq <- getSequence(leprae$req[[1]])
> query("ulcerans", "AC=A0PQ23")
> ulceransseq <- getSequence(ulcerans$req[[1]])
> closebank()
> lepraeseq # Display the contents of "lepraeseq"
  [1] "M" "T" "N" "R" "T" "L" "S" "R" "E" "E" "I" "R" "K" "L" "D" "R" "D" "L"
 [19] "R" "I" "L" "V" "A" "T" "N" "G" "T" "L" "T" "R" "V" "L" "N" "V" "V" "A"
 [37] "N" "E" "E" "I" "V" "V" "D" "I" "I" "N" "Q" "Q" "L" "L" "D" "V" "A" "P"
 [55] "K" "I" "P" "E" "L" "E" "N" "L" "K" "I" "G" "R" "I" "L" "Q" "R" "D" "I"
 [73] "L" "L" "K" "G" "Q" "K" "S" "G" "I" "L" "F" "V" "A" "A" "E" "S" "L" "I"
 [91] "V" "I" "D" "L" "L" "P" "T" "A" "I" "T" "T" "Y" "L" "T" "K" "T" "H" "H"
[109] "P" "I" "G" "E" "I" "M" "A" "A" "S" "R" "I" "E" "T" "Y" "K" "E" "D" "A"
[127] "Q" "V" "W" "I" "G" "D" "L" "P" "C" "W" "L" "A" "D" "Y" "G" "Y" "W" "D"
[145] "L" "P" "K" "R" "A" "V" "G" "R" "R" "Y" "R" "I" "I" "A" "G" "G" "Q" "P"
[163] "V" "I" "I" "T" "T" "E" "Y" "F" "L" "R" "S" "V" "F" "Q" "D" "T" "P" "R"
[181] "E" "E" "L" "D" "R" "C" "Q" "Y" "S" "N" "D" "I" "D" "T" "R" "S" "G" "D"
[199] "R" "F" "V" "L" "H" "G" "R" "V" "F" "K" "N" "L"
```

Comparing two 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 R package.

For example, if we want to create a dotplot of the sequences for the chorismate lyase proteins from *Mycobacterium leprae* and *Mycobacterium ulcerans*, we would type:

```
> dotPlot(lepraeseq, ulceransseq)
```
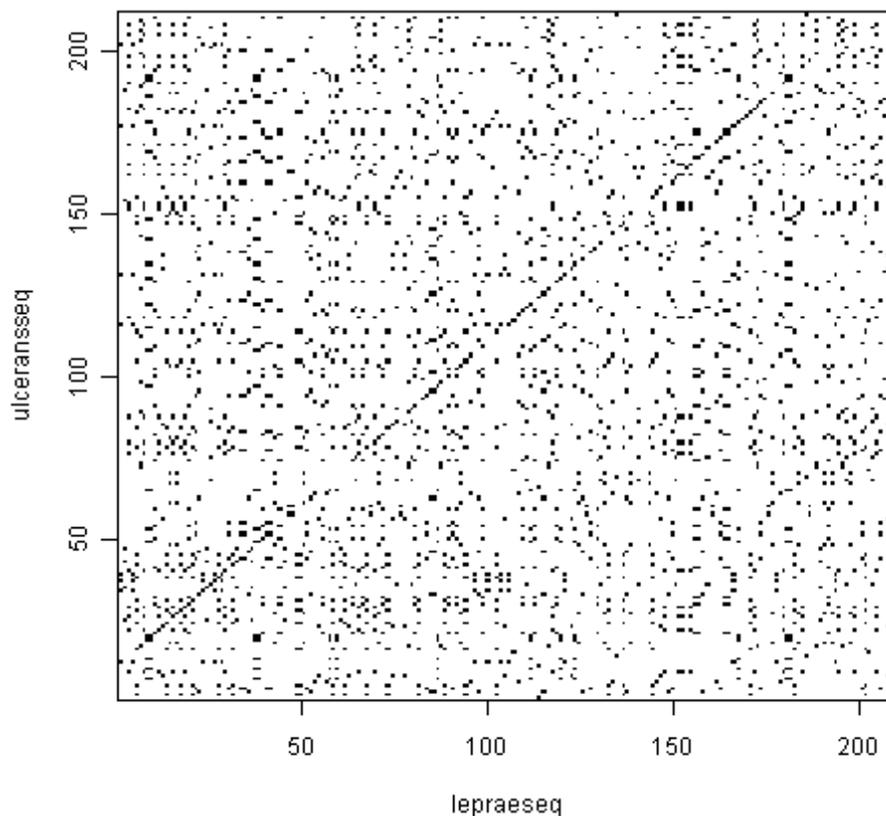

In the dotplot above, the *M. leprae* sequence is plotted along the *x*-axis (horizontal axis), and the *M. ulcerans* sequence is plotted along the *y*-axis (vertical axis). The dotplot displays a dot at points where there is an identical amino acid in the two sequences.

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 homologues (related proteins).

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, 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 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. Thus, for the alignment:

```
G A A T T C
G A T T - A
```

we would compute a score of 2 + 2 -1 + 2 -2 - 1 = 2. Similarly, the score for the following alignment is 2 + 2 -2 + 2 + 2 -1 = 5:

```
G A A T T C
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 (eg. 4 rows and 4 columns for DNA sequences). 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. The Biostrings package is part of a set of R packages for bioinformatics analysis known as Bioconductor (www.bioconductor.org/).

To use the Biostrings package, you will first need to install the package (see the instructions here).

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 (eg. 'N' = A/C/G/T).

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:

```
> library(Biostrings)
> sigma <- nucleotideSubstitutionMatrix(match = 2, mismatch = -1, baseOnly = TRUE)
> sigma # Print out the matrix
   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 (eg. -8) to every gap position, it is common instead to assign a *gap opening penalty* to the first position in a gap (eg. -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 penalise a 3-letter gap as much as we would penalise three separate 1-letter gaps, 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 R 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), the pairwiseAlignment() function takes the two sequences that you want to align, the scoring matrix, the gap opening penalty, and 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, to find the score for the optimal global alignment between the sequences 'GAATTC' and 'GATTA', we type:

```
> s1 <- "GAATTC"
> s2 <- "GATTA"
> globalAligns1s2 <- pairwiseAlignment(s1, s2, substitutionMatrix = sigma,
→gapOpening = -2,
gapExtension = -8, scoreOnly = FALSE)
> globalAligns1s2 # Print out the optimal alignment and its score
Global Pairwise Alignment (1 of 1)
pattern: [1] GAATTC
subject: [1] GA-TTA
score: -3
```

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

Note that 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.

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.

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 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. The "data()" function is used to load these data files. You can use the data() function in R to load a data set of BLOSUM matrices that comes with R Biostrings() package.

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  0 -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  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")
Data sets in package 'Biostring':
BLOSUM100                               Scoring matrices
BLOSUM45                                Scoring matrices
BLOSUM50                                Scoring matrices
BLOSUM62                                Scoring matrices
BLOSUM80                                Scoring matrices
```

To find the optimal global alignment between the protein sequences "PAWHEAE" and "HEAGAWGHEE" using the Needleman-Wunsch algorithm using the BLOSUM50 matrix, we type:

```
> data(BLOSUM50)
> s3 <- "PAWHEAE"
> s4 <- "HEAGAWGHEE"
> globalAligns3s4 <- pairwiseAlignment(s3, s4, substitutionMatrix = "BLOSUM50",
↪gapOpening = -2,
gapExtension = -8, scoreOnly = FALSE)
> globalAligns3s4 # Print out the optimal global alignment and its score
Global Pairwise Alignment (1 of 1)
pattern: [1] P---AWHEAE
subject: [1] 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.

Aligning UniProt sequences

We discussed above how you can search for UniProt accessions and retrieve the corresponding protein sequences, either via the UniProt website or using the SeqinR R package.

In the examples given above, you learnt 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 (eg. "ACGTA"), rather than as a vector of characters (eg. 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 c2s() function in the SeqinR package:

```
> lepraeseqstring <- c2s(lepraeseq)      # Make a string that contains the sequence␣
→in "lepraeseq"
> ulceransseqstring <- c2s(ulceransseq) # Make a string that contains the sequence␣
→in "ulceransseq"
```

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 *lepraeseqstring* and *ulcer-ansseqstring* to uppercase:

```
> lepraeseqstring <- toupper(lepraeseqstring)
> ulceransseqstring <- toupper(ulceransseqstring)
> lepraeseqstring # Print out the content of "lepraeseqstring"
 [1]
→"MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLLDVAPKIPELENLKIGRILQRDILLKGQKSGILFVAAESLIVIDL1
→"
```

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

```
> globalAlignLepraeUlcerans <- pairwiseAlignment(lepraeseqstring,␣
→ulceransseqstring,
  substitutionMatrix = BLOSUM50, gapOpening = -2, gapExtension = -8, scoreOnly =␣
→FALSE)
> globalAlignLepraeUlcerans # Print out the optimal global alignment and its score
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [1] MT-----NR--T---LSREEIRKLDRDLRILVATN...
→QDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKN
  subject: [1] MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATN...EDNSREEPIRHQRS--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 (see above). Therefore, we need to have a function to print out the whole alignment (see below).

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 "printPairwiseAlignment()" below will do this for you:

```
> printPairwiseAlignment <- function(alignment, chunksize=60, returnlist=FALSE)
  {
     require(Biostrings)          # This function requires the Biostrings package
     seq1aln <- pattern(alignment) # Get the alignment for the first sequence
     seq2aln <- subject(alignment) # Get the alignment for the second sequence
     alnlen  <- nchar(seq1aln)     # Find the number of columns in the alignment
     starts  <- seq(1, alnlen, by=chunksize)
     n        <- length(starts)
     seq1alnresidues <- 0
     seq2alnresidues <- 0
     for (i in 1:n) {
        chunkseq1aln <- substring(seq1aln, starts[i], starts[i]+chunksize-1)
        chunkseq2aln <- substring(seq2aln, starts[i], starts[i]+chunksize-1)
        # Find out how many gaps there are in chunkseq1aln:
        gaps1 <- countPattern("-",chunkseq1aln) # countPattern() is from␣
→Biostrings package
        # Find out how many gaps there are in chunkseq2aln:
        gaps2 <- countPattern("-",chunkseq2aln) # countPattern() is from␣
→Biostrings package
        # Calculate how many residues of the first sequence we have printed so far␣
→in the alignment:
```

```
        seq1alnresidues <- seq1alnresidues + chunksize - gaps1
        # Calculate how many residues of the second sequence we have printed so
→far in the alignment:
        seq2alnresidues <- seq2alnresidues + chunksize - gaps2
        if (returnlist == 'FALSE')
        {
            print(paste(chunkseq1aln,seq1alnresidues))
            print(paste(chunkseq2aln,seq2alnresidues))
            print(paste(' '))
        }
    }
    if (returnlist == 'TRUE')
    {
        vector1 <- s2c(substring(seq1aln, 1, nchar(seq1aln)))
        vector2 <- s2c(substring(seq2aln, 1, nchar(seq2aln)))
        mylist <- list(vector1, vector2)
        return(mylist)
    }
}
```

To use this function you first need to copy and paste this function into R. You can then use our function printPairwiseAlignment() to print out the alignment between the *M. leprae* and *M. ulcerans* chorismate lyase proteins (we stored this alignment in the *globalAlignLepraeUlcerans* variable, see above), in blocks of 60 alignment columns:

```
> printPairwiseAlignment(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.

Since we are printing out an alignment that contained gaps in the first 60 alignment columns, the first 60 alignment columns ends before the 60th amino acid in the *M. leprae* sequence.

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, the classic bioinformatics algorithm for finding optimal local alignments.

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

```
> localAlignLepraeUlcerans <- pairwiseAlignment(lepraeseqstring, ulceransseqstring,
  substitutionMatrix = BLOSUM50, gapOpening = -2, gapExtension = -8, scoreOnly =
→FALSE, type="local")
> localAlignLepraeUlcerans # Print out the optimal local alignment and its score
  Local PairwiseAlignedFixedSubject (1 of 1)
```

```
   pattern:  [1] MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVV...
→IITTEYFLRSVFQDTPREELDRCQYSNDIDTRSG
   subject: [11] MTECHLSDEEIRKLNRDLRILIATNGTLTRILNVL...
→IIITEYFLRSVFEDNSREEPIRHQRSVGTSARSG
   score: 761
> printPairwiseAlignment(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.

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 1/7*th* of the circle is taken up by a pie labelled "P", 2/7*ths* by a pie labelled "A", 1/7*th* by a pie labelled "W", 1/7*th* by a pie labelled "H", and 2/7*ths* by a pie labelled "E":

$p_P=0.14$, $p_A=0.28$, $p_W=0.14$, $p_H=0.14$, $p_E=0.28$
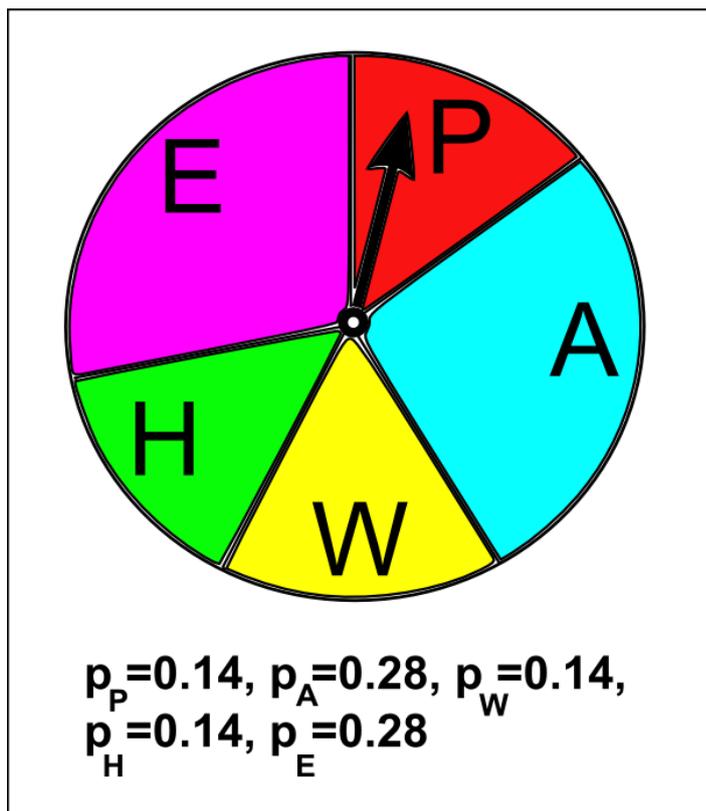
To generate a sequence using the multinomial model, we keep spinning the arrow in the centre of the roulette wheel, and write down the letter that the arrow stops on after each spin. To generate a sequence that is 7 letters long, we can spin the arrow 7 times. To generate 1000 sequences that are each 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 generateSeqsWithMultinomialModel() below:

```
> generateSeqsWithMultinomialModel <- function(inputsequence, X)
  {
     # Change the input sequence into a vector of letters
     require("seqinr") # This function requires the SeqinR package.
     inputsequencevector <- s2c(inputsequence)
     # Find the frequencies of the letters in the input sequence
↪"inputsequencevector":
     mylength <- length(inputsequencevector)
     mytable <- table(inputsequencevector)
     # Find the names of the letters in the sequence
     letters <- rownames(mytable)
     numletters <- length(letters)
     probabilities <- numeric() # Make a vector to store the probabilities of␣
↪letters
     for (i in 1:numletters)
     {
        letter <- letters[i]
        count <- mytable[[i]]
        probabilities[i] <- count/mylength
     }
     # Make X random sequences using the multinomial model with probabilities
↪"probabilities"
     seqs <- numeric(X)
     for (j in 1:X)
     {
        seq <- sample(letters, mylength, rep=TRUE, prob=probabilities) # Sample␣
↪with replacement
```

```
        seq <- c2s(seq)
        seqs[j] <- seq
    }
    # Return the vector of random sequences
    return(seqs)
}
```

The function generateSeqsWithMultinomialModel() 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' (ie. 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 <- generateSeqsWithMultinomialModel('PAWHEAE',1000)
> randomseqs[1:10] # Print out the first 10 random sequences
  [1] "EHHEWEA" "EAEEEAH" "WAHAWEP" "PPAPAAW" "HEPWWAA" "APAAAAA" "EAHAPHP"
  [8] "AAPEEWE" "HEAAAAP" "EWAAPEP"
```

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"
> pairwiseAlignment(s4, randomseqs[1], substitutionMatrix = "BLOSUM50", gapOpening
↪= -2,
  gapExtension = -8, scoreOnly = FALSE)
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [2] EAGAWGHEE
  subject: [1] EHHEW--EA
  score: -7
```

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] -7
```

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 'HEAGAWGHEE' (ie. -5) to the distribution of scores for aligning 'HEAGAWGHEE' to the random sequences.

```
> randomscores <- double(1000) # Create a numeric vector with 1000 elements
> for (i in 1:1000)
  {
    score <- pairwiseAlignment(s4, randomseqs[i], substitutionMatrix = "BLOSUM50",
      gapOpening = -2, gapExtension = -8, scoreOnly = TRUE)
```

```
    randomscores[i] <- score
}
```

The code above first uses the double() function to create a numeric vector *randomscores* for storing real numbers (ie. 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.

Once we have run the 'for loop', we can make a histogram plot of the 1000 scores in vector *randomscores* by typing:

```
> hist(randomscores, col="red") # Draw a red histogram
```
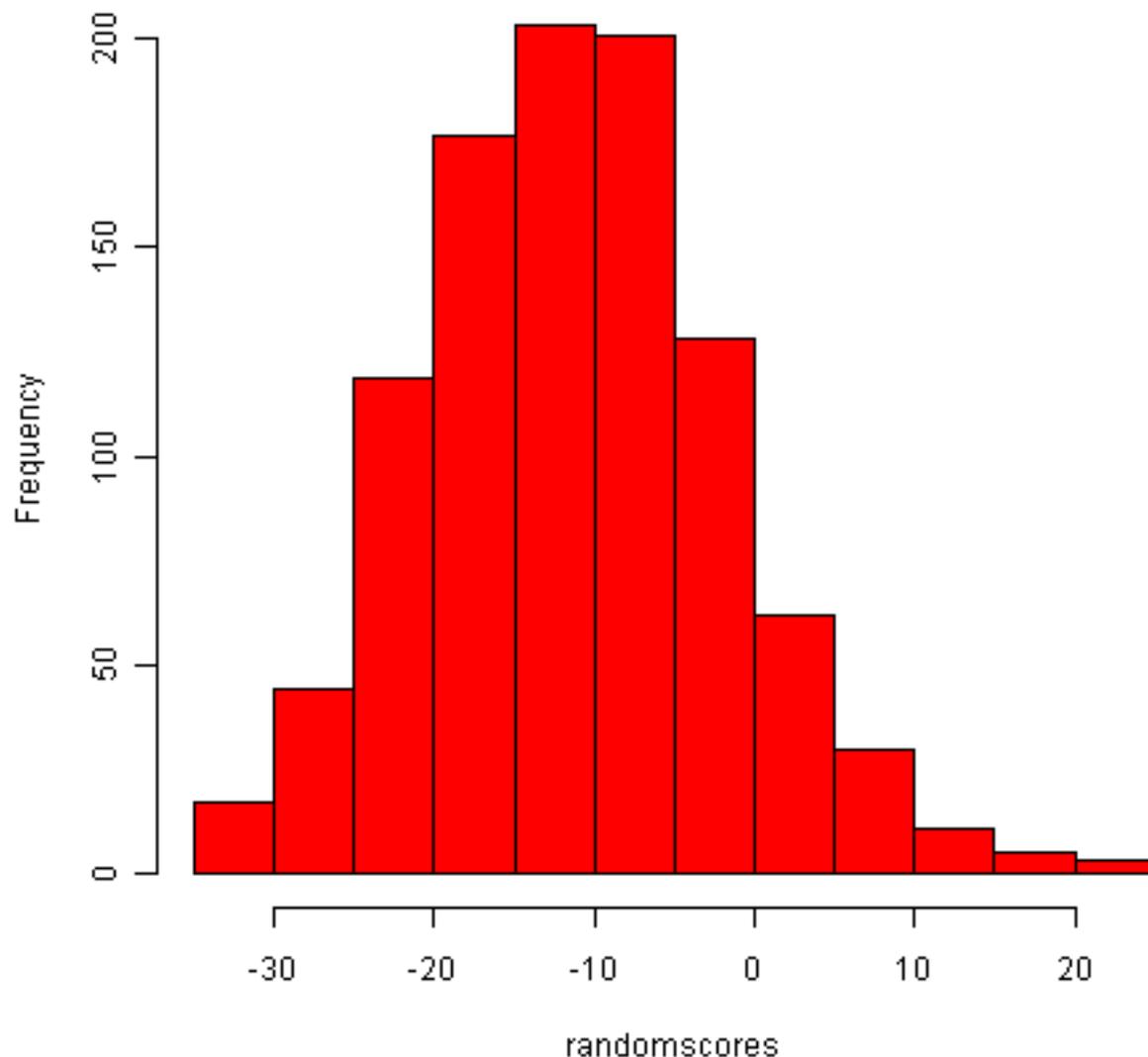

We can see from the histogram that quite a lot of the random sequences seem to have higher alignment scores than -5 when aligned to 'HEAGAWGHEE' (where -5 is the alignment score for 'PAWHEAE' and 'HEAGAWGHEE').

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' (ie. -5) by chance.

```
> sum(randomscores >= -5)
[1] 266
```

We see that 266 of the 1000 alignments of random sequences to 'HEAGAWGHEE' 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 'HEAGAWGHEE' 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).

Summary

In this practical, you will have learnt to use the following R functions:

1. data() for reading in data that comes with an R package

2. double() for creating a numeric vector for storing real (non-integer) numbers

3. 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 learnt the following R functions that belong to the bioinformatics packages:

1. nucleotideSubstitutionMatrix() in the Biostrings package for making a nucleotide scoring matrix

2. pairwiseAlignment() in the Biostrings package for making a global alignment between two sequences

3. c2s() in the SeqinR package for converting a sequence stored in a vector to a string of characters

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.

Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the Matlab case study on the Eyeless protein (www.computational-genomics.net/case_studies/eyeless_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/).

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).

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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 Sequence Alignment.

Q1. Download FASTA-format files of the *Brugia malayi* Vab-3 protein (UniProt accession A8PZ80) and the *Loa loa* Vab-3 pr
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.

Q2. What is the alignment score for the optimal global alignment between the *Brugia malayi* Vab-3 protein and the *Loa loa* V
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.

Q3. Use the printPairwiseAlignment() function to view the optimal global alignment between *Brugia malayi* Vab-3 protein a
Do you see any regions where the alignment is very good (lots of identities and few gaps)?

Q4. What global alignment score do you get for the two Vab-3 proteins, when you use the *BLOSUM62* alignment matrix, a g
Which scoring matrix do you think is more appropriate for using for this pair of proteins: BLOSUM50 or BLOSUM62?

Q5. What is the statistical significance of the optimal global alignment for the *Brugia malayi* and *Loa loa* Vab-3 proteins mad
In other words, what is the probability of getting a score as large as the real alignment score for Vab-3 by chance?

Q6. What is the optimal global alignment score between the *Brugia malayi* Vab-6 protein and the *Mycobacterium leprae* chor
Is the alignment score statistically significant (what is the *P*- value?)? Does this surprise you?

Multiple Alignment and Phylogenetic trees

Retrieving a list of sequences from UniProt

In previous chapters, you learnt how to search for DNA or protein sequences in sequence databases such as the NCBI database and UniProt, using the SeqinR package (see chapter3.html).

For example, in the previous chapter), you learnt how to retrieve a single sequence from UniProt.

Oftentimes, it is useful to retrieve several sequences from UniProt at once if you have a list of UniProt accessions. The R function "retrieveseqs()" below is useful for this purpose:

```
> retrieveseqs <- function(seqnames,acnucdb)
  {
     myseqs <- list()    # Make a list to store the sequences
     require("seqinr")   # This function requires the SeqinR R package
     choosebank(acnucdb)
     for (i in 1:length(seqnames))
     {
        seqname <- seqnames[i]
        print(paste("Retrieving sequence",seqname,"..."))
        queryname <- "query2"
        query <- paste("AC=",seqname,sep="")
        query(`queryname`,`query`)
        seq <- getSequence(query2$req[[1]]) # Makes a vector "seq" containing the␣
→sequence
        myseqs[[i]] <- seq
     }
     closebank()
     return(myseqs)
  }
```

You need to cut and paste this function into R to use it. As its input, you need to give it the function a vector containing the accessions for the sequences you wish to retrieve, as well as the name of the ACNUC sub-database that the sequences should be retrieved from. In this case, we want to retrieve sequences from UniProt, so the sequences should be in the "swissprot" ACNUC sub-database.

The retrieveseqs() function returns a list variable, in which each element is a vector containing one of the sequences.

For example, to retrieve the protein sequences for UniProt accessions P06747, P0C569, O56773 and Q5VKP1 (the accessions for rabies virus phosphoprotein, Mokola virus phosphoprotein, Lagos bat virus phosphoprotein and Western Caucasian bat virus phosphoprotein, respectively), you can type:

```
> seqnames <- c("P06747", "P0C569", "O56773", "Q5VKP1")   # Make a vector␣
→containing the names of the sequences
> seqs <- retrieveseqs(seqnames,"swissprot")              # Retrieve the sequences␣
→and store them in list variable "seqs"
> length(seqs)                                            # Print out the number of sequences␣
→retrieved
 [1] 4
> seq1 <- seqs[[1]]                                       # Get the first sequence
> seq1[1:20]                                              # Print out the first 20 letters of␣
→the first sequence
 [1] "M" "S" "K" "I" "F" "V" "N" "P" "S" "A" "I" "R" "A" "G" "L" "A" "D" "L" "E"
 [20] "M"
> seq2 <- seqs[[2]]                                       # Get the second sequence
> seq2[1:20]                                              # Print out the first 20 letters of␣
→the second sequence
 [1] "M" "S" "K" "D" "L" "V" "H" "P" "S" "L" "I" "R" "A" "G" "I" "V" "E" "L" "E"
 [20] "M"
```

The commands above use the function retrieveseqs() to retrieve two UniProt sequences. The sequences are returned in a list variable *seqs*. To access the elements in an R list variable, you need to use double square brackets. Therefore, the second element of the list variable is accessed by typing *seqs[[2]]*. Each element of the list variable *seqs* contains a vector which stores one of the sequences.

Rabies virus is the virus responsible for rabies, which is classified by the WHO as a neglected tropical disease. Mokola virus 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.

Once you have retrieved the sequences using retrieveseqs(), you can then use the function write.fasta() from the SeqinR package to write the sequences to a FASTA-format file. As its arguments (inputs), the write.fasta() function takes the list variable containing the sequences, and a vector containing the names of the sequences, and the name that you want to give to the FASTA-format file. For example:

```
> write.fasta(seqs, seqnames, file="phosphoproteins.fasta")
```

The command above will write the sequences in list variable *seqs* to a FASTA-format file called "phosphoproteins.fasta" in the "My Documents" folder on your computer.

Installing the CLUSTAL multiple alignment software

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 most phylogenetic analyses.

One commonly used multiple alignment software package is CLUSTAL. In order to build an alignment using CLUSTAL, you first need to install the CLUSTAL program on your computer.

To install CLUSTAL on your computer, you need to follow these steps:

- Go to the http://www.clustal.org/download/current/ website.

- Right-click on the link to file clustalx-Z.Z.Z-win.msi (where Z represents some number) and choose "Save link as..." and then save the file in your "My Documents" folder.

- Once the file has downloaded, double-click on the icon for file clustalx-Z.Z.Z-win.msi (where Z is some number).

- You will be asked "Are you sure you want to run this software?" Press "Run".

- You will then see "Welcome to the ClustalX2 setup wizard". Press "Next".

- You will be asked where to install ClustalX2. Select your "My Documents" folder.

- Keep pressing 'yes' or 'Next' until the screen says "Completing the ClustalX2 setup wizard". Then press "Finish".

CLUSTAL should now be installed on your computer.

Creating a multiple alignment of protein, DNA or mRNA sequences using CLUSTAL

Once you have installed CLUSTAL, you can now align your sequences using CLUSTAL by following these steps:

- Go to the "Start" menu on the bottom left of your Windows screen. Select "All Programs" from the menu, then select "ClustalX2" from the menu that appears. This will start up CLUSTAL.

- The CLUSTAL window should appear. To load the DNA or protein sequences that you want to align into CLUSTAL, go to the CLUSTAL "File" menu, and choose "Load sequences".

- Select the FASTA-format file containing your sequences (eg. phosphoproteins.fasta) to load it into CLUSTAL.

- This should read the sequences into CLUSTAL. They have not been aligned yet, but will be displayed in the CLUSTAL window.

- You can use the scrollbar on the right to scroll down and look at all the sequences. You can use the scrollbar on the bottom to scroll from left to right, and look along the length of the sequences.

- Before you align the sequences using CLUSTAL, you need to tell CLUSTAL to make the output alignment file in PHYLIP alignment format, so that you can read it into R. To do this, go to the "Alignment" menu in CLUSTAL, choose "Output Format Options". A form will appear, and in this form you should select "PHYLIP format" and deselect "CLUSTAL format", and then press "OK".

- To now align the sequences using CLUSTAL, go to the CLUSTAL "Alignment" menu, and choose "Do Complete Alignment".

- A menu box will pop up, asking you where to save the output guide-tree file (eg. "phosphoproteins.dnd") and the output alignment file (called "phosphoproteins.phy"). You should choose to save them in your "My Documents" folder (so that you can easily read them into R from "My Documents" at a later stage).

- CLUSTAL will now align the sequences. This will take a couple of minutes (eg. 2-5 minutes). You will see that at the bottom of the CLUSTAL window, it tells you which pair of sequences it is aligning at a particular point in time. If the numbers keep changing, it means that CLUSTAL is still working away, and the alignment is not finished yet. Be patient!

Once CLUSTAL has finished making the alignment, it will be displayed in the CLUSTAL window. For example, here is the CLUSTAL alignment for rabies virus phosphoprotein, Mokola virus phosphoprotein, and Lagos bat virus phosphoprotein:
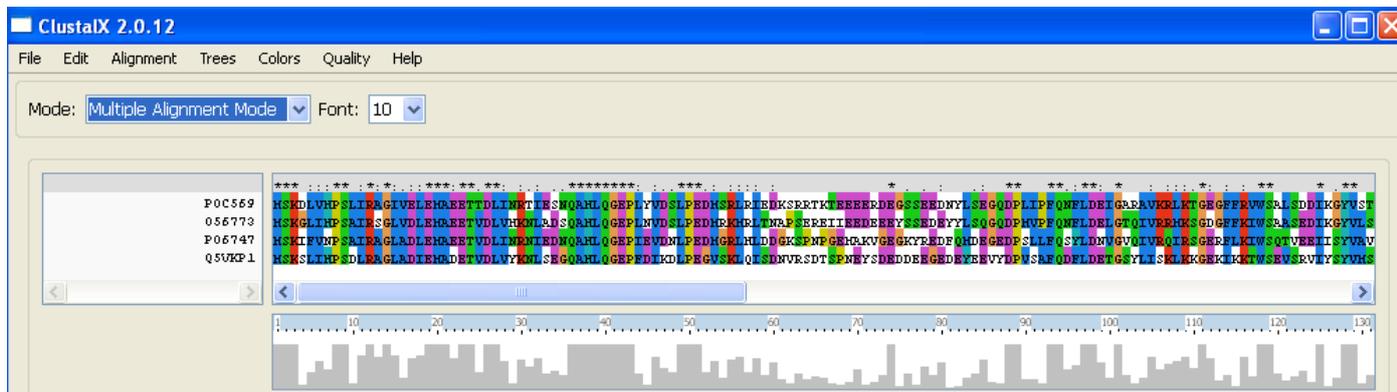
The alignment displayed in CLUSTAL has a row for each of your sequences. CLUSTAL colours sets of chemically similar amino acids in similar colours. For example, tyrosine (Y) is coloured blue-green, while the chemically similar amino acid phenylalanine (F) is coloured blue. You can scroll to the right and left along the alignment using the scrollbar at the bottom of the Jalview window.

Below the alignment, you can see a grey plot, showing the level of conservation at each point of the sequence. This shows a high grey bar if the conservation in a region is high (there is high percent identity between the sequence), and a low grey bar if it is low (there is low percent identity). This can give you an idea of which are the best conserved regions of the alginment.

For example, for the alignment of the four virus phosphoproteins, we can see that the region in alignment columns 35 to 45 approximately is very well conserved, while the region in alignment columns 60 to 70 is poorly conserved.

The CLUSTAL alignment will have been saved in a file in your "My Documents" folder called "something.phy" (eg. phosphoproteins.phy). This is a PHYLIP-format alignment file, which you can now read into R for further analysis.

Reading a multiple alignment file into R

To read a sequence alignment into R from a file, you can use the read.alignment() function in the SeqinR package. For example, to read in the multiple sequence alignment of the virus phosphoproteins into R, we type:

```
> virusaln  <- read.alignment(file = "phosphoproteins.phy", format = "phylip")
```

The *virusaln* variable is a list variable that stores the alignment.

An R list variable can have named elements, and you can access the named elements of a list variable by typing the variable name, followed by "$", followed by the name of the named element.

The list variable *virusaln* has named elements "nb", "nam", "seq", and "com".

In fact, the named element "seq" contains the alignment, which you can view by typing:

```
> virusaln$seq
  [[1]]
  [1] "mskdlvhpsliragivelemaeettdlinrtiesnqahlqgeplyvdslpedmsrlriedksrrtk..."
  [[2]]
  [1] "mskglihpsairsglvdlemaeetvdlvhknladsqahlqgeplnvdslpedmrkmrltnapsere..."
  [[3]]
  [1] "mskifvnpsairagladlemaeetvdlinrniednqahlqgepievdnlpedmgrlhlddgkspnp..."
  [[4]]
  [1] "mskslihpsdlragladiemadetvdlvyknlsegqahlqgepfdikdlpegvsklqisdnvrsdt..."
```

Only the first part of the alignment stored in *virusaln$seq* is shown here, as it is very long.

Viewing a long multiple alignment

If you want to view a long multiple alignment, it is convenient to view the multiple alignment in blocks.

The R function "printMultipleAlignment()" below will do this for you:

```
> printMultipleAlignment <- function(alignment, chunksize=60)
  {
     # this function requires the Biostrings package
     require("Biostrings")
     # find the number of sequences in the alignment
     numseqs <- alignment$nb
     # find the length of the alignment
     alignmentlen <- nchar(alignment$seq[[1]])
     starts <- seq(1, alignmentlen, by=chunksize)
     n <- length(starts)
     # get the alignment for each of the sequences:
     aln <- vector()
     lettersprinted <- vector()
     for (j in 1:numseqs)
     {
        alignmentj <- alignment$seq[[j]]
        aln[j] <- alignmentj
        lettersprinted[j] <- 0
     }
     # print out the alignment in blocks of 'chunksize' columns:
     for (i in 1:n) { # for each of n chunks
        for (j in 1:numseqs)
        {
           alnj <- aln[j]
           chunkseqjaln <- substring(alnj, starts[i], starts[i]+chunksize-1)
           chunkseqjaln <- toupper(chunkseqjaln)
           # Find out how many gaps there are in chunkseqjaln:
           gapsj <- countPattern("-",chunkseqjaln) # countPattern() is from␣
→Biostrings package
           # Calculate how many residues of the first sequence we have printed so␣
→far in the alignment:
           lettersprinted[j] <- lettersprinted[j] + chunksize - gapsj
           print(paste(chunkseqjaln,lettersprinted[j]))
        }
        print(paste(' '))
```

```
        }
    }
```

As its inputs, the function "printMultipleAlignment()" takes the input alignment, and the number of columns to print out in each block.

For example, to print out the multiple alignment of virus phosphoproteins (which we stored in variable *virusaln*, see above) in blocks of 60 columns, we type:

```
> printMultipleAlignment(virusaln, 60)
  [1] "MSKDLVHPSLIRAGIVELEMAEETTDLINRTIESNQAHLQGEPLYVDSLPEDMSRLRIED 60"
  [1] "MSKGLIHPSAIRSGLVDLEMAEETVDLVHKNLADSQAHLQGEPLNVDSLPEDMRKMRLTN 60"
  [1] "MSKIFVNPSAIRAGLADLEMAEETVDLINRNIEDNQAHLQGEPIEVDNLPEDMGRLHLDD 60"
  [1] "MSKSLIHPSDLRAGLADIEMADETVDLVYKNLSEGQAHLQGEPFDIKDLPEGVSKLQISD 60"
  [1] " "
  [1] "KSRRTKTEEEERDEGSSEEDNYLSEGQDPLIPFQNFLDEIGARAVKRLKTGEGFFRVWSA 120"
  [1] "APSEREIIEEDEEEYSSEDEYYLSQGQDPMVPFQNFLDELGTQIVRRMKSGDGFFKIWSA 120"
  [1] "GKSPNPGEMAKVGEGKYREDFQMDEGEDPSLLFQSYLDNVGVQIVRQIRSGERFLKIWSQ 120"
  [1] "NVRSDTSPNEYSDEDDEEGEDEYEEVYDPVSAFQDFLDETGSYLISKLKKGEKIKKTWSE 120"
  [1] " "
  [1] "LSDDIKGYVSTNIM-TSGERDTKSIQIQTEPTASVSSGNESRHDSESMHDPNDKKDHTPD 179"
  [1] "ASEDIKGYVLSTFM-KPETQATVSKPTQTDSLSVPRPSQGYTSVPRDKPSNSESQGGGVK 179"
  [1] "TVEEIISYVAVNFP-NPPGKSSEDKSTQTTGRELKKETTPTPSQRESQSSKARMAAQTAS 179"
  [1] "VSRVIYSYVMSNFPPRPPKPTTKDIAVQADLKKPNEIQKISEHKSKSEPSPREPVVEMHK 180"
  [1] " "
  [1] "HDVVPDIESSTDKGEIRDIEGEVAHQVAESFSKKYKFPSRSSGIFLWNFEQLKMNLDDIV 239"
  [1] "PKKVQKSEWTRDTDEISDIEGEVAHQVAESFSKKYKFPSRSSGIFLWNFEQLKMNLDDIV 239"
  [1] "GPPALEWSATNEEDDLS-VEAEIAHQIAESFSKKYKFPSRSSGILLYNFEQLKMNLDDIV 238"
  [1] "HATLE-----NPEDDEGALESEIAHQVAESYSKKYKFPSKSSGIFLWNFEQLKMNLDDIV 235"
  [1] " "
  [1] "KAAMNVPGVERIAEKGGKLPLRCILGFVALDSSKRFRLLADNDKVARLIQEDINSYMARL 299"
  [1] "KTSMNVPGVDKIAEKGGKLPLRCILGFVSLDSSKRFRLLADTDKVARLMQDDIHNYMTRI 299"
  [1] "KEAKNVPGVTRLARDGSKLPLRCVLGWVALANSKKFQLLVESNKLSKIMQDDLNRYTSC- 297"
  [1] "QVARGVPGISQIVERGGKLPLRCMLGYVGLETSKRFRSLVNQDKLCKLMQEDLNAYSVSS 295"
  [1] " "
  [1] "EEAE-- 357"
  [1] "EEIDHN 359"
  [1] "------ 351"
  [1] "NN---- 351"
  [1] " "
```

Discarding very poorly conserved regions from an alignment

It is often a good idea to discard very poorly conserved regions from a mulitple alignment before building a phylogenetic tree, as the very poorly conserved regions are likely to be regions that are either not homologous between the sequences being considered (and so do not add 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, "cleanAlignment()":

```
> cleanAlignment <- function(alignment, minpcnongap, minpcid)
  {
    # make a copy of the alignment to store the new alignment in:
    newalignment <- alignment
    # find the number of sequences in the alignment
    numseqs <- alignment$nb
    # empty the alignment in "newalignment")
    for (j in 1:numseqs) { newalignment$seq[[j]] <- "" }
    # find the length of the alignment
```

```r
    alignmentlen <- nchar(alignment$seq[[1]])
    # look at each column of the alignment in turn:
    for (i in 1:alignmentlen)
    {
        # see what percent of the letters in this column are non-gaps:
        nongap <- 0
        for (j in 1:numseqs)
        {
            seqj <- alignment$seq[[j]]
            letterij <- substr(seqj,i,i)
            if (letterij != "-") { nongap <- nongap + 1}
        }
        pcnongap <- (nongap*100)/numseqs
        # Only consider this column if at least minpcnongap % of the letters are
→not gaps:
        if (pcnongap >= minpcnongap)
        {
            # see what percent of the pairs of letters in this column are identical:
            numpairs <- 0; numid <- 0
            # find the letters in all of the sequences in this column:
            for (j in 1:(numseqs-1))
            {
                seqj <- alignment$seq[[j]]
                letterij <- substr(seqj,i,i)
                for (k in (j+1):numseqs)
                {
                    seqk <- alignment$seq[[k]]
                    letterkj <- substr(seqk,i,i)
                    if (letterij != "-" && letterkj != "-")
                    {
                        numpairs <- numpairs + 1
                        if (letterij == letterkj) { numid <- numid + 1}
                    }
                }
            }
            pcid <- (numid*100)/(numpairs)
            # Only consider this column if at least %minpcid of the pairs of
→letters are identical:
            if (pcid >= minpcid)
            {
                for (j in 1:numseqs)
                {
                    seqj <- alignment$seq[[j]]
                    letterij <- substr(seqj,i,i)
                    newalignmentj <- newalignment$seq[[j]]
                    newalignmentj <- paste(newalignmentj,letterij,sep="")
                    newalignment$seq[[j]] <- newalignmentj
                }
            }
        }
    }
    return(newalignment)
}
```

The function cleanAlignment() takes three arguments (inputs): the input alignment; the minimum percent of letters in an alignment column that must be non-gap characters for the column to be kept; and 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 column with letters "T", "A", "T", "-" (in four sequences), then 75% of the letters are non-gap characters; and the pairs of letters are "T,A", "T,T", and "A,T", and 33% of the pairs of letters are identical.

We can use the function cleanAlignment() to discard the very poorly aligned columns from a multiple alignment.

For example, if you look at the multiple alignment for the virus phosphoprotein sequences (which we printed out using function printMultipleAlignment(), 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.

Therefore, to filter out the well conserved columns of the alignment, and discard the very poorly conserved columns, we can type:

```
> cleanedvirusaln <- cleanAlignment(virusaln, 30, 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:

```
> printMultipleAlignment(cleanedvirusaln)
  [1] "MSKLVHPSIRAGIVELEMAEETTDLIRTIQAHLQGEPVDLPEDMRLIDREEEDEGDPFQF 60"
  [1] "MSKLIHPSIRSGLVDLEMAEETVDLVKNLQAHLQGEPVDLPEDMKMLNSEEEEQGDPFQF 60"
  [1] "MSKFVNPSIRAGLADLEMAEETVDLIRNIQAHLQGEPVDLPEDMRLLDSAERDEGDPFQY 60"
  [1] "MSKLIHPSLRAGLADIEMADETVDLVKNLQAHLQGEPIKLPEGVKLIDREEEEEVDPFQF 60"
  [1] " "
  [1] "LDEGVKGEFRWSSIGYVNIMSTSIQTHSDESGEDEEVAHQVAESFSKKYKFPSRSSGIFL 120"
  [1] "LDEGVKGDFKWSSIGYVTFMPTSKQTSDSETDEDEEVAHQVAESFSKKYKFPSRSSGIFL 120"
  [1] "LDNGVRGEFKWSVISYVNFPPSDKQTSSSSTDD-EEIAHQIAESFSKKYKFPSRSSGILL 119"
  [1] "LDEGIKGEIKWSSISYVNFPPTDIQAHSS--DDAEEIAHQVAESYSKKYKFPSKSSGIFL 118"
  [1] " "
  [1] "WNFEQLKMNLDDIVKANVPGVIAEGGKLPLRCLGVLSKRFRLLADKVRLIQEDINYEE 180"
  [1] "WNFEQLKMNLDDIVKSNVPGVIAEGGKLPLRCLGVLSKRFRLLADKVRLMQDDIHYEE 180"
  [1] "YNFEQLKMNLDDIVKANVPGVLARGSKLPLRCLGVLSKKFQLLVNKLKIMQDDLNY-- 177"
  [1] "WNFEQLKMNLDDIVQAGVPGIIVEGGKLPLRCLGVLSKRFRSLVDKLKLMQEDLNYNN 178"
  [1] " "
```

The filtered alignment is shorter, but 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.

Calculating genetic distances between protein sequences

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 divergence between two sequences, and is usually measured in quantity of evolutionary change (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 alignment as input. Based on the multiple alignment that you give it, dist.alignment() calculates the genetic distance between each pair of proteins in the multiple alignment. For example, to calculate genetic distances between the virus phosphoproteins based on the multiple sequence alignment stored in *virusaln*, we type:

```
> virusdist <- dist.alignment(virusaln)                    # Calculate the
↪genetic distances
> virusdist                                                 # Print out the
↪genetic distance matrix
              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. If you remember from above, P06747 is rabies virus phosphoprotein, P0C569 is Mokola virus phosphoprotein, O56773 is Lagos bat virus phosphoprotein and 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 or indels that have occurred since they shared a common ancestor, and the longer ago their common ancestor probably lived.

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.

For example, the NCBI accession AF049118 contains mRNA sequence for Mokola virus phosphoprotein, RefSeq AF049114 contains mRNA sequence for Mokola virus phosphoprotein, and AF049119 contains the mRNA sequence for Lagos bat virus phosphoprotein, while AF049115 contains the mRNA sequence for Duvenhage virus phosphoprotein.

To retrieve these sequences from the NCBI database, we can search the ACNUC "genbank" sub-database (since these are nucleotide sequences), by typing:

```
> seqnames <- c("AF049118", "AF049114", "AF049119", "AF049115")  # Make a vector
→containing the names of the sequences
> seqs <- retrieveseqs(seqnames,"genbank")                       # Retrieve the
→sequences and store them in list variable "seqs"
```

We can then write out the sequences to a FASTA-format file by typing:

```
> write.fasta(seqs, seqnames, file="virusmRNA.fasta")
```

We can then use CLUSTAL to create a PHYLIP-format alignment of the sequences, and store it in the alignment file "virusmRNA.phy". This picture shows part of the alignment:
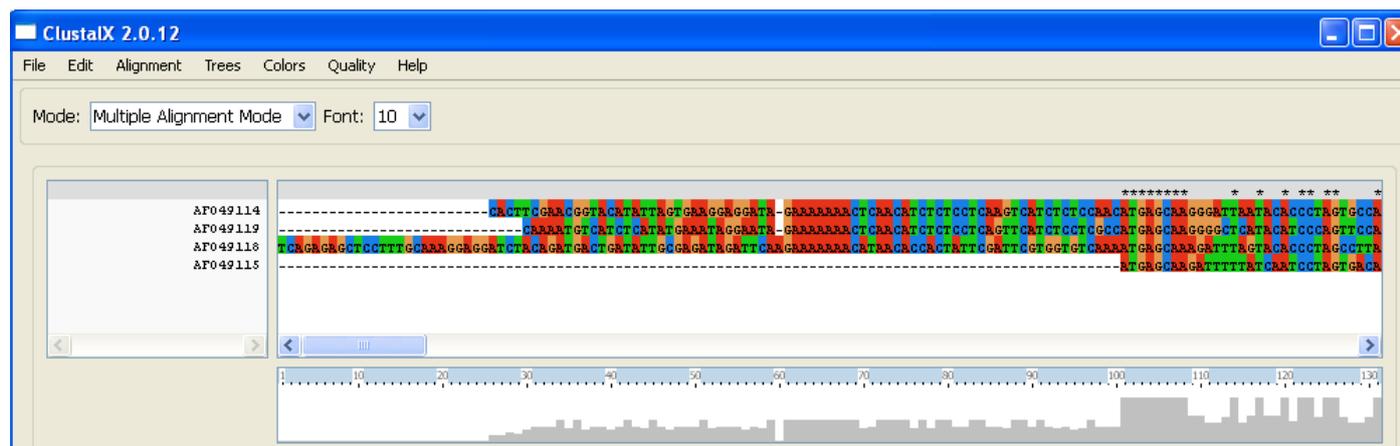
We can then read the alignment into R:

```
> virusmRNAaln  <- read.alignment(file = "virusmRNA.phy", format = "phylip")
```

We saw above that the function dist.alignment() can be used to calculate a genetic distance matrix based on a protein sequence alignment.

You can calculate a genetic distance for DNA or mRNA sequences using the dist.dna() function in the Ape R package. dist.dna()</tt> takes a multiple alignment 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.

For example, to calculate the genetic distance between each pair of mRNA sequences for the virus phosphoproteins, we type:

```
> virusmRNAalnbin <- as.DNAbin(virusmRNAaln) # Convert the alignment to "DNAbin"␣
↪format
> virusmRNAdist <- dist.dna(virusmRNAalnbin) # Calculate the genetic distance␣
↪matrix
> virusmRNAdist                              # Print out the genetic distance␣
↪matrix
          AF049114  AF049119  AF049118
  AF049119 0.3400576
  AF049118 0.5235850 0.5637372
  AF049115 0.6854129 0.6852311 0.7656023
```

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 *neighbour-joining algorithm*.

You can build a phylogenetic tree using the neighbour-joining algorithm with the the Ape R package. First you will need to install the "ape" package (see instructions on how to install R packages).

The following R function "unrootedNJtree()" builds a phylogenetic tree based on an alignment of sequences, using the neighbour-joining algorithm, using functions from the "ape" package.

The "unrootedNJtree()" function takes an alignment of sequences its input, calculates pairwise distances between the sequences based on the alignment, and then builds a phylogenetic tree based on the pairwise distances. It returns the phylogenetic tree, and also makes a picture of that tree:

```
> unrootedNJtree <- function(alignment,type)
  {
     # this function requires the ape and seqinR packages:
     require("ape")
     require("seqinr")
     # define a function for making a tree:
     makemytree <- function(alignmentmat)
     {
        alignment <- ape::as.alignment(alignmentmat)
        if      (type == "protein")
        {
           mydist <- dist.alignment(alignment)
        }
        else if (type == "DNA")
        {
           alignmentbin <- as.DNAbin(alignment)
           mydist <- dist.dna(alignmentbin)
        }
        mytree <- nj(mydist)
        mytree <- makeLabel(mytree, space="") # get rid of spaces in tip names.
        return(mytree)
     }
     # infer a tree
     mymat  <- as.matrix.alignment(alignment)
     mytree <- makemytree(mymat)
```

```
    # bootstrap the tree
    myboot <- boot.phylo(mytree, mymat, makemytree)
    # plot the tree:
    plot.phylo(mytree,type="u")    # plot the unrooted phylogenetic tree
    nodelabels(myboot,cex=0.7)     # plot the bootstrap values
    mytree$node.label <- myboot    # make the bootstrap values be the node labels
    return(mytree)
}
```

To use the function to make a phylogenetic tree, you must first copy and paste the function into R. You can then use it to make a tree, for example of the virus phosphoproteins, based on the sequence alignment:

```
> virusalntree <- unrootedNJtree(virusaln,type="protein")
```
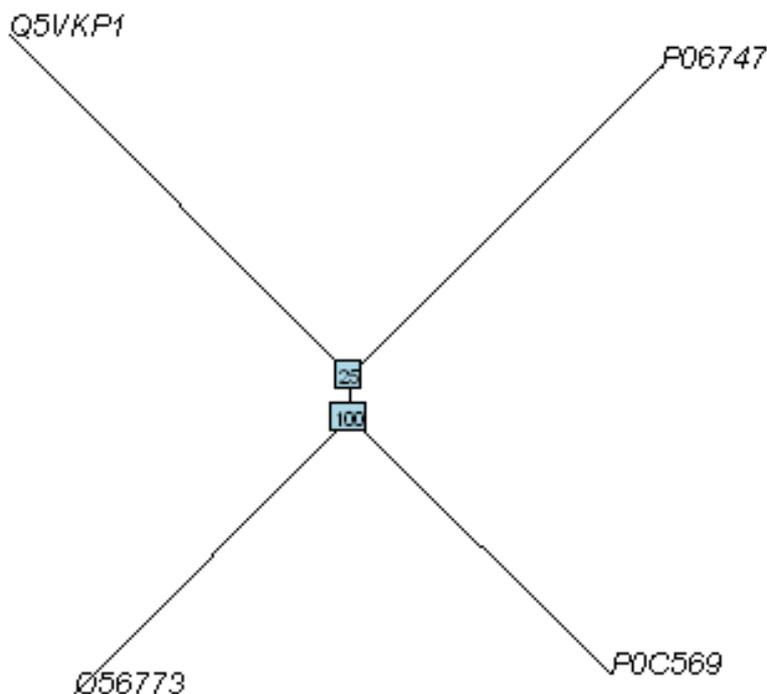

Note that you need to specify that the type of sequences that you are using are protein sequences when you use unrootedNJtree(), by setting "type=protein".

We can see that Q5VKP1 (Western Caucasian bat virus phosphoprotein) and P06747 (rabies virus phosphoprotein) have been grouped together in the tree, and that O56773 (Lagos bat virus phosphoprotein) and P0C569 (Mokola virus phosphoprotein) are grouped together in the tree.

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.

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.

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* (eg. 200) of randomly sampled columns from the alignment. Each "resample" of the alignment (eg. 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.

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 containg 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.

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.

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 (as far as I know) 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 relatd 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 *cleanedvirusaln* (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:

```
> cleanedvirusalntree <- unrootedNJtree(cleanedvirusaln,type="protein")
```
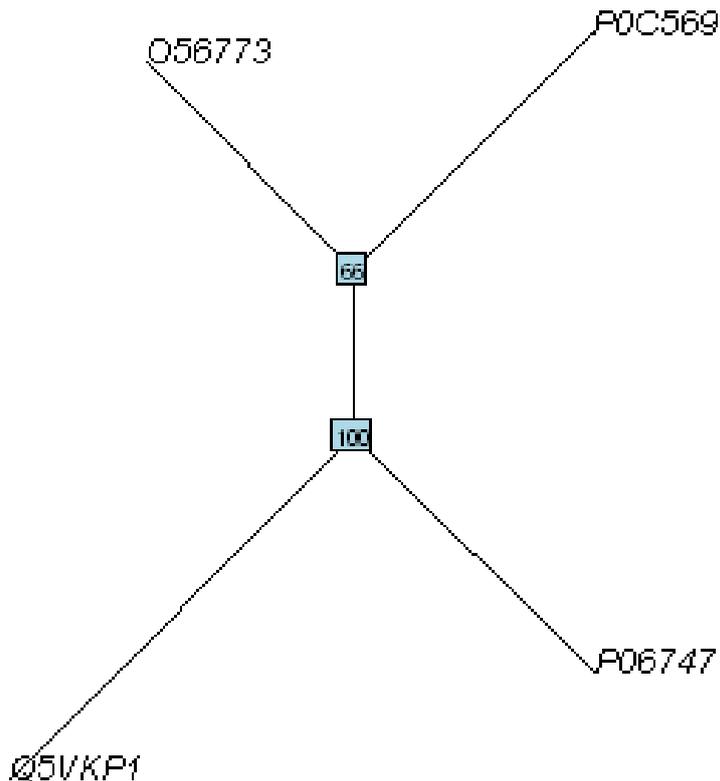
Here O56773 and P0C569 are grouped together, and Q5VKP1 and P06747 are grouped together, as in the phylogenetic tree based on the raw (unfiltered) multiple alignment (see above). Thus, filtering the multiple alignment does not have an effect on the tree.

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).

Building a rooted phylogenetic tree for protein sequences

In order to convert the unrooted tree into a rooted tree, we need to add an outgroup sequence. Normally, the outgroup sequence is a sequence that we know from some prior knowledge to be more distantly related to the other sequences under study than they are to each other.

For example, the protein Fox-1 is involved in determining the sex (gender) of an embryo in the nematode worm *Caenorhabditis elegans* (UniProt accession Q10572). Related proteins are found in other nematodes, including *Caenorhabditis remanei* (UniProt E3M2K8), *Caenorhabditis briggsae* (A8WS01), *Loa loa* (E1FUV2), and *Brugia malayi* (UniProt A8NSK3).

Note that *Caenorhabditis elegans* is a model organism commonly studied in molecular biology. The nematodes *Loa loa*, and *Brugia malayi* are parasitic nematodes that cause filariasis, which is classified by the WHO as a neglected tropical disease.

The UniProt database contains a distantly related sequence from the fruitfly *Drosophila melanogaster* (UniProt accession Q9VT99). If we were to build a phylogenetic tree of the nematode worm Fox-1 homologues, the distantly related sequence from fruitfly would probably be a good choice of outgroup, since the protein is from a different animal group (insects) than the nematode worms. Thus, it is likely that the fruitfly protein is more distantly related to all the nematode proteins than they are to each other.

To retrieve the sequences from UniProt we can use the "retrieveseqs()" function (see above):

```
> seqnames <- c("Q10572","E3M2K8","Q8WS01","E1FUV2","A8NSK3","Q9VT99")
> seqs <- retrieveseqs(seqnames,"swissprot")
```

We can then write out the sequences to a FASTA file:

```
> write.fasta(seqs, seqnames, file="fox1.fasta")
```

We can then use CLUSTAL to create a PHYLIP-format alignment of the sequences, and store it in the alignment file "fox1.phy". This picture shows part of the alignment (the alignment is quite long, so not all of it is shown):

We can then read the alignment into R:

```
> fox1aln  <- read.alignment(file = "fox1.phy", format = "phylip")
```

The next step is to build a phylogenetic tree of the proteins, which again we can do using the neighbour-joining algorithm.

This time we have an outgroup in our set of sequences, so we can build a rooted tree. The function "rootedNJtree()" can be used to build a rooted tree. It returns the phylogenetic tree, and also makes a picture of the tree:

```
> rootedNJtree <- function(alignment, theoutgroup, type)
  {
     # load the ape and seqinR packages:
     require("ape")
     require("seqinr")
     # define a function for making a tree:
     makemytree <- function(alignmentmat, outgroup=`theoutgroup`)
     {
        alignment <- ape::as.alignment(alignmentmat)
        if      (type == "protein")
        {
           mydist <- dist.alignment(alignment)
        }
        else if (type == "DNA")
        {
           alignmentbin <- as.DNAbin(alignment)
           mydist <- dist.dna(alignmentbin)
        }
        mytree <- nj(mydist)
        mytree <- makeLabel(mytree, space="") # get rid of spaces in tip names.
        myrootedtree <- root(mytree, outgroup, r=TRUE)
        return(myrootedtree)
     }
     # infer a tree
     mymat  <- as.matrix.alignment(alignment)
     myrootedtree <- makemytree(mymat, outgroup=theoutgroup)
     # bootstrap the tree
```

1.8. Multiple Alignment and Phylogenetic trees   **77**

```
    myboot <- boot.phylo(myrootedtree, mymat, makemytree)
    # plot the tree:
    plot.phylo(myrootedtree, type="p")   # plot the rooted phylogenetic tree
    nodelabels(myboot,cex=0.7)           # plot the bootstrap values
    mytree$node.label <- myboot   # make the bootstrap values be the node labels
    return(mytree)
}
```

The function takes the alignment and the name of the outgroup as its inputs. For example, to use it to make a phylogenetic tree of the *C. elegans* Fox-1 protein and its homologues, using the fruitfly protein (UniProt Q9VT99) as the outgroup, we type:

```
> fox1alntree <- rootedNJtree(fox1aln, "Q9VT99",type="protein")
```
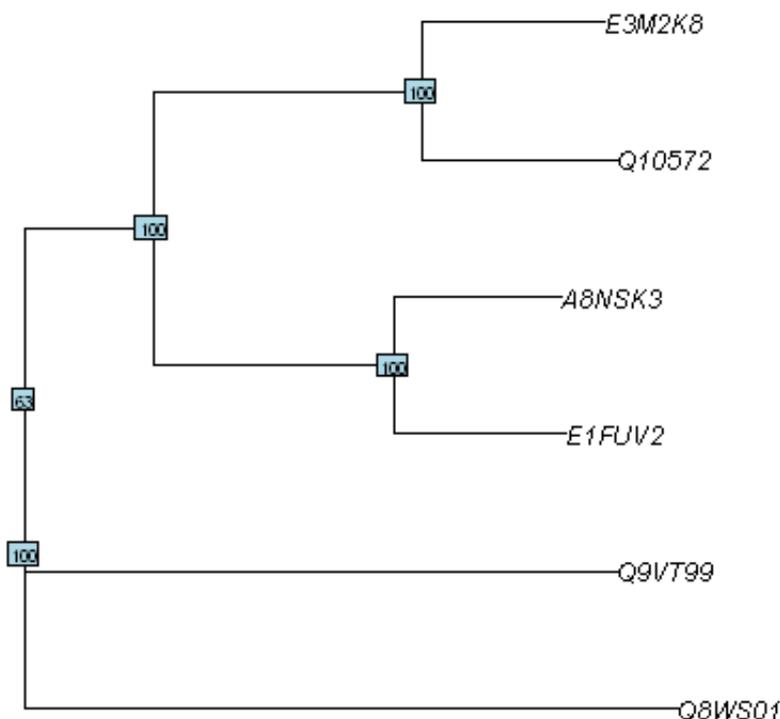

Here we can see that E3M2K8 (*C. remanei* Fox-1 homologue) and Q10572 (*C. elegans* Fox-1) have been grouped together with bootstrap 100%, and A8NSK3 (*Brugia malayi* Fox-1 homologue) and E1FUV2 (*Loa loa* Fox-1 homologue) have been grouped together with bootstrap 100%. These four proteins have also been grouped together in a larger clade with bootstrap 100%.

Compared to these four proteins, the Q8WS01 (*C. briggsae* Fox-1 homologue) and Q9VT99 (fruitfly outgroup) seem to be relatively distantly related.

As this is a rooted tree, we know the direction that evolutionary time ran. Say we call the ancestor of the four sequences (E3M2K8, Q10572, A8NSK3, E1FUV2) *ancestor1*, the ancestor of the two sequences (E3M2K8, Q10572) *ancestor2*, and the ancestor of the two sequences (A8NSK3, E1FUV2) *ancestor3*.

Because it is a rooted tree, we know that time ran from left to right along the branches of the tree, so that *ancestor1* was the ancestor of *ancestor2*, and *ancestor1* was also the ancestor of *ancestor3*. In other words, *ancestor1* lived before *ancestor2* or *ancestor3*; *ancestor2* and *ancestor3* were descendants of *ancestor1*.

Another way of saying this is that E3M2K8 and Q10572 shared a common ancestor with each other more recently than they did with A8NSK3 and E1FUV2.

The lengths of branches in this tree are proportional to the amount of evolutionary change (estimated number of mutations) that occurred along the branches. The branches leading back from E3M2K8 and Q10572 to their last

common ancestor are slightly longer than the branches leading back from A8NSK3 and E1FUV2 to their last common ancestor.

This indicates that there has been more evolutionary change in E3M2K8 (*C. remanei* Fox-1 homologue) and Q10572 (*C. elegans* Fox-1) proteins since they diverged, than there has been in A8NSK3 (*Brugia malayi* Fox-1 homologue) and E1FUV2 (*Loa loa* Fox-1 homologue) since they diverged.

Building a phylogenetic tree for DNA or mRNA sequences

In the example above, a phylogenetic tree was built for protein sequences. The genomes of distantly related organisms such as vertebrates will have accumulated many mutations since they diverged. Sometimes, so many mutations have occurred since the organisms diverged that their DNA sequences are hard to align correctly and it is also hard to accurately estimate evolutionary distances from alignments of those DNA sequences.

In contrast, as many mutations at the DNA level are synonymous at the protein level, protein sequences diverge at a slower rate than DNA sequences. This is why for reasonably distantly related organisms such as vertebrates, it is usually preferable to use protein sequences for phylogenetic analyses.

If you are studying closely related organisms such as primates, few mutations will have occurred since they diverged. As a result, if you use protein sequences for a phylogenetic analysis, there may be too few amino acid substitutions to provide enough 'signal' to use for the phylogenetic analysis. Therefore, it is often preferable to use DNA sequences for a phylogenetic analysis of closely related organisms such as primates.

We can use the functions unrootedNJtree() and rootedNJtree() described above to build unrooted or rooted neighbour-joining phylogenetic trees based on an alignment of DNA or mRNA sequences. In this case, we need to use "type=DNA" as an argument in these functions, to tell them that we are making a tree of nucleotide sequences, not protein sequences.

For example, to build an unrooted phylogenetic tree based on the alignment of the virus phosphoprotein mRNA sequences, we type in R:

```
> virusmRNAaln  <- read.alignment(file = "virusmRNA.phy", format = "phylip")
> virusmRNAalntree <- unrootedNJtree(virusmRNAaln, type="DNA")
```
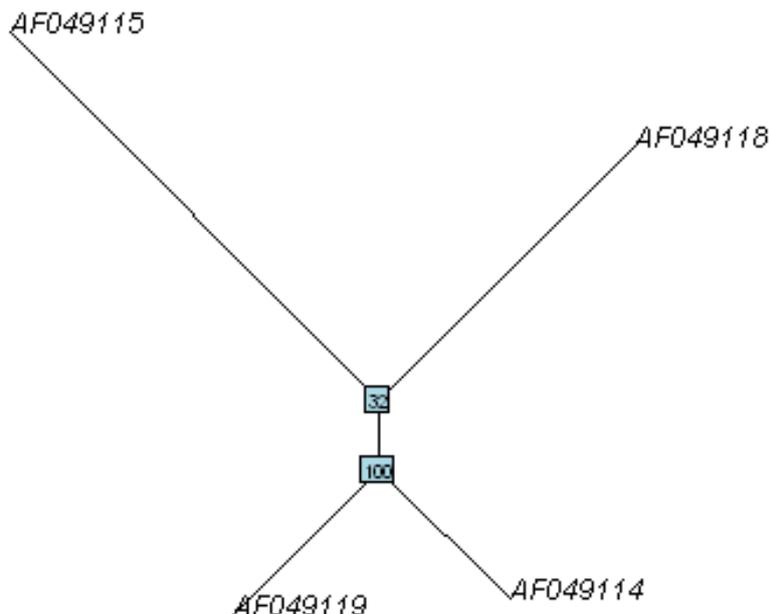
Saving a phylogenetic tree as a Newick-format tree file

A commonly used format for representing phylogenetic trees is the Newick format. Once you have built a phylogenetic tree using R, it is convenient to store it as a Newick-format tree file. This can be done using the "write.tree()" function in the Ape R package.

For example, to save the unrooted phylogenetic tree of virus phosphoprotein mRNA sequences as a Newick-format tree file called "virusmRNA.tre", we type:

```
> write.tree(virusmRNAalntree, "virusmRNA.tre")
```

The Newick-format file "virusmRNA.tre" should now appear in your "My Documents" folder.

Summary

In this practical, you have learnt the following R functions that belong to the bioinformatics packages:

1. read.alignment() from the SeqinR package for reading in a multiple alignment

2. dist.alignment() from the SeqinR package for calculating genetic distances between protein sequences

3. dist.dna() from the Ape package for calculating genetic distances between DNA or mRNA sequences

Links and Further Reading

Some links are included here for further reading.

For background reading on phylogenetic trees, it is recommended to read Chapter 7 of *Introduction to Computational Genomics: a case studies approach* by Cristianini and Hahn (Cambridge University Press; www.computational-genomics.net/book/).

Another more in-depth (but very accesssible) book on phylogenetics is *Molecular Evolution: A Phylogenetic Approach* by Roderic DM Page and Edward C Holmes.

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.

For more in-depth information and more examples on the Ape package for phylogenetic analysis, look at the Ape documentation, ape.mpl.ird.fr/.

If you are using the Ape package for a phylogenetic analysis project, it would be worthwhile to obtain a copy of the book *Analysis of Phylogenetics and Evolution with R* by Emmanuel Paradis, published by Springer, which has many nice examples of using R for phylogenetic analyses.

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.

Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the Matlab case study on SARS (www.computational-genomics.net/case_studies/sars_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/).

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

Thank you to Emmanuel Paradis and François Michonneau for help in using the Ape package.

Thank you also to Klaus Schliep for helpful comments.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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 Multiple Alignment and Phylogenetic Trees.

Q1. Calculate the genetic distances between the following NS1 proteins from different Dengue virus strains: Dengue virus 1

> Note: Dengue virus causes Dengue fever, which is classified by the WHO as a neglected tropical disease. There are four main types of Dengue virus, Dengue virus 1, Dengue virus 2, Dengue virus 3, and Dengue virus 4.

Q2. Build an unrooted phylogenetic tree of the NS1 proteins from Dengue virus 1, Dengue virus 2, Dengue virus 3 and Dengue virus 4, using the neighbour-joining algorithm. Which are the most closely related proteins, based on the tree? Based on the bootstrap values in the tree, how confident are you of this?

Q3. Build an unrooted phylogenetic tree of the NS1 proteins from Dengue viruses 1-4, based on a filtered alignment of the four proteins (keeping alignment columns in which at least 30% of letters are not gaps, and in which at least 30% of pairs of letters are identical). Does this differ from the tree based on the unfiltered alignment (in Q2)? Can you explain why?

Q4. The Zika virus is related to Dengue viruses, but is not a Dengue virus, and so therefore can be used as an outgroup in phylogenetic trees of Dengue virus sequences. UniProt accession Q32ZE1 consists of a sequence with similarity to the Dengue NS1 protein, so seems to be a related protein from Zika virus. Build a rooted phylogenetic tree of the Dengue NS1 proteins based on a filtered alignment (keeping alignment columns in which at least 30% of letters are not gaps, and in which at least 30% of pairs of letters are identical), using the Zika virus protein as the outgroup. Which are the most closely related Dengue virus proteins, based on the tree? What extra information does this tree tell you, compared to the unrooted tree in Q2?

REVISION EXERCISES 2

These are some revision exercises on sequence alignment and phylogenetic trees.

Exercises

Answer the following questions. For each question, please record your answer, and what you did/typed to get this answer.

Model answers to the exercises are given in Answers to Revision Exercises 2.

Q1.

One of the key proteins produced by rabies virus is the rabies phosphoprotein (also known as rabies virus protein P). The U

> Note: rabies virus is the virus responsible for rabies, which is classified by the WHO as a neglected tropical disease. Mokola virus 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.

Q2.

The function "makeDotPlot1()" below is an R function that makes a dotplot of two sequences by plotting a dot at every position where the two sequences share an identical letter. Use this function to make a dotplot of the rabies virus phosphoprotein and the Mokola virus phosphoprotein, setting the argument "dotsize" to 0.1 (this determines the radius of each dot plotted). Are there any long regions of similarity between the two proteins (if so, where are they)? Do you find the same regions as found in Q1, and if not, can you explain why?

```r
> makeDotPlot1 <- function(seq1,seq2,dotsize=1)
  {
     length1 <- length(seq1)
     length2 <- length(seq2)
     # make a plot:
     x <- 1
     y <- 1
     plot(x,y,ylim=c(1,length2),xlim=c(1,length1),col="white") # make an empty plot
     # now plot dots at every position where the two sequences have the same
→letter:
     for (i in 1:length1)
     {
        letter1 <- seq1[i]
        for (j in 1:length2)
        {
           letter2 <- seq2[j]
           if (letter1 == letter2)
           {
              # add a point to the plot
              points(x=i,y=j,cex=dotsize,col="blue",pch=7)
           }
        }
     }
  }
```

Q3.

Adapt the R code in Q2 to write a function that makes a dotplot using a window of size x letters, where a dot is plotted in the first cell of the window if y or more letters compared in that window are identical in the two sequences.

Q4.

Use the dotPlot() function in the SeqinR R package to make a dotplot of rabies virus phosphoprotein and Mokola virus phosphoprotein, using a window size of 3 and a threshold of 3. Use your own R function from Q3 to make a dotplot of rabies virus phosphoprotein and Mokola virus phosphoprotein, using a windowsize (x) of 3 and a threshold (y) of 3. Are the two plots similar or different, and can you explain why?

Q5.

Write an R function to calculate an unrooted phylogenetic tree with bootstraps, using the minimum evolution method (rather than the neighbour-joining method, which is used by the function unrootedNJtree).

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Computational Gene-finding

The genetic code

A protein-coding gene starts with an "ATG", which is followed by an integer (whole) number of codons (DNA triplets) that code for amino acids, and ends 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".

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:

```
> library(seqinr)
> tablecode()
```

Genetic code 1 : standard

TTT	Phe	T CT	Ser	T AT	Tyr	T GT	Cys
TT C	Phe	T CC	Ser	T AC	Tyr	T GC	Cys
TT A	Leu	T CA	Ser	T AA	Stp	T GA	Stp
TT G	Leu	T CG	Ser	T AG	Stp	T GG	Trp
CTT	Leu	CCT	Pro	CAT	His	CGT	Arg
CT C	Leu	CCC	Pro	CA C	His	CGC	Arg
CT A	Leu	CCA	Pro	CA A	Gln	CGA	Arg
CT G	Leu	CCG	Pro	CA G	Gln	CGG	Arg
ATT	Ile	ACT	Thr	AAT	Asn	A GT	Ser
AT C	Ile	ACC	Thr	AA C	Asn	A GC	Ser
AT A	Ile	ACA	Thr	AA A	Lys	A GA	Arg
AT G	Met	ACG	Thr	AA G	Lys	A GG	Arg
GTT	Val	GCT	Ala	GAT	Asp	GGT	Gly
GT C	Val	GCC	Ala	GA C	Asp	GGC	Gly
GT A	Val	GCA	Ala	GA A	Glu	GGA	Gly
GT G	Val	GCG	Ala	GA G	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).

Finding start and stop codons in a DNA sequence

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 R package, which identifies all occurrences of a particular motif (eg. "ATG") in a sequence. As input, the matchPattern() function requires that the sequences be in the form of a string of single characters.

For example, we can look for all "ATG"s in the sequence "AAAATGCAGTAACCCATGCCC" by typing:

```
> library("Biostrings")
> s1 <- "aaaatgcagtaacccatgccc"
> 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. In fact, we can see these by looking at the sequence "AAAATGCAGTAACCCATGCCC".

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 following R function "findPotentialStartsAndStops()" can be used to find all potential start and stop codons in a DNA sequence:

```
> findPotentialStartsAndStops <- function(sequence)
  {
    # Define a vector with the sequences of potential start and stop codons
    codons            <- c("atg", "taa", "tag", "tga")
    # Find the number of occurrences of each type of potential start or stop codon
    for (i in 1:4)
    {
      codon <- codons[i]
      # Find all occurrences of codon "codon" in sequence "sequence"
      occurrences <- matchPattern(codon, sequence)
      # Find the start positions of all occurrences of "codon" in sequence
→"sequence"
      codonpositions <- attr(occurrences,"start")
      # Find the total number of potential start and stop codons in sequence
→"sequence"
      numoccurrences <- length(codonpositions)
      if (i == 1)
      {
        # Make a copy of vector "codonpositions" called "positions"
        positions <- codonpositions
        # Make a vector "types" containing "numoccurrences" copies of "codon"
        types <- rep(codon, numoccurrences)
      }
      else
      {
        # Add the vector "codonpositions" to the end of vector "positions":
        positions   <- append(positions, codonpositions,␣
→after=length(positions))
        # Add the vector "rep(codon, numoccurrences)" to the end of vector
→"types":
        types       <- append(types, rep(codon, numoccurrences),␣
→after=length(types))
      }
```

```
    }
    # Sort the vectors "positions" and "types" in order of position along the
↪input sequence:
    indices <- order(positions)
    positions <- positions[indices]
    types <- types[indices]
    # Return a list variable including vectors "positions" and "types":
    mylist <- list(positions,types)
    return(mylist)
}
```

To use the function, you will need to copy and paste it into R. For example, we can use this function to find potential start and stop codons in sequence *s1*:

```
> s1 <- "aaaatgcagtaacccatgccc"
> findPotentialStartsAndStops(s1)
[[1]]
[1]   4 10 16

[[2]]
[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 findPotentialStartsAndStops() 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 learnt that you can retrieve a sequence for an NCBI accession using the "getncbiseq()" function. Thus, to retrieve the genome sequence of the DEN-1 Dengue virus (NCBI accession NC_001477), we can type:

```
> dengueseq <- getncbiseq("NC_001477")
```

The variable *dengueseq* is a vector variable, and each letter in the DEN-1 Dengue virus DNA sequence is stored in one element of this vector.

Dengue virus causes Dengue fever, which is classified as a neglected tropical disease by the WHO.

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:

```
> dengueseqstart <- dengueseq[1:500] # Take the first 500 nucleotides of the DEN-1
↪Dengue sequence
> length(dengueseqstart)                # Find the length of the "dengueseqstart"
↪start vector
[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 findPotentialStartsAndStops() function described above. However, the findPotentialStartsAndStops() 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:

```
> library("seqinr")                    # Load 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" "a"
 [20] "c" "c" "g" "a" "c" "a" "a" "g" "a" "a" "c" "a" "g" "t" "t" "t" "c" "g" "a"
```

```
    [39] "a" "t" "c" "g" "g" "a" "a" "g" "c" "t" "t" "g" "c" "t" "t" "a" "a" "c" "g"
    [58] "t" "a" "g" "t" "t" "c" "t" "a" "a" "c" "a" "g" "t" "t" "t" "t" "t" "t" "a"
    [77] "t" "t" "a" "g" "a" "g" "a" "g" "c" "a" "g" "a" "t" "c" "t" "c" "t" "g" "a"
    [96] "t" "g" "a" "a" "c" "a" "a" "c" "c" "a" "a" "c" "g" "g" "a" "a" "a" "a" "a"
   [115] "g" "a" "c" "g" "g" "g" "t" "c" "g" "a" "c" "c" "g" "t" "c" "t" "t" "t" "c"
   [134] "a" "a" "t" "a" "t" "g" "c" "t" "g" "a" "a" "a" "c" "g" "c" "g" "c" "g" "a"
   [153] "g" "a" "a" "a" "c" "c" "g" "c" "g" "t" "g" "t" "c" "a" "a" "c" "t" "g" "t"
   [172] "t" "t" "c" "a" "c" "a" "g" "t" "t" "g" "g" "c" "g" "a" "a" "g" "a" "g" "a"
   [191] "t" "t" "c" "t" "c" "a" "a" "a" "a" "g" "g" "a" "t" "t" "g" "c" "t" "t" "t"
   [210] "c" "a" "g" "g" "c" "c" "a" "a" "g" "g" "a" "c" "c" "c" "a" "t" "g" "a" "a"
   [229] "a" "t" "t" "g" "g" "t" "g" "a" "t" "g" "g" "c" "t" "t" "t" "t" "a" "t" "a"
   [248] "g" "c" "a" "t" "t" "c" "c" "t" "a" "a" "g" "a" "t" "t" "t" "c" "t" "a" "g"
   [267] "c" "c" "a" "t" "a" "c" "c" "t" "c" "c" "a" "a" "c" "a" "g" "c" "a" "g" "g"
   [286] "a" "a" "t" "t" "t" "t" "g" "g" "c" "t" "a" "g" "a" "t" "g" "g" "g" "g" "c"
   [305] "t" "c" "a" "t" "t" "c" "a" "a" "g" "a" "a" "g" "a" "a" "t" "g" "g" "a" "g"
   [324] "c" "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" "a"
   [362] "a" "a" "c" "a" "t" "g" "t" "t" "g" "a" "a" "c" "a" "t" "a" "a" "t" "g" "a"
   [381] "a" "c" "a" "g" "g" "a" "g" "g" "a" "a" "a" "a" "g" "a" "t" "c" "t" "g" "t"
   [400] "g" "a" "c" "c" "a" "t" "g" "c" "t" "c" "c" "t" "c" "a" "t" "g" "c" "t" "g"
   [419] "c" "t" "g" "c" "c" "c" "a" "c" "a" "g" "c" "c" "c" "t" "g" "g" "c" "g" "t"
   [438] "t" "c" "c" "a" "t" "c" "t" "g" "a" "c" "c" "a" "c" "c" "c" "g" "a" "g" "g"
   [457] "g" "g" "g" "a" "g" "a" "g" "c" "c" "g" "c" "a" "c" "a" "t" "g" "a" "t" "a"
   [476] "g" "t" "t" "a" "g" "c" "a" "a" "g" "c" "a" "g" "g" "a" "a" "a" "g" "a" "g"
   [495] "g" "a" "a" "a" "a" "t"
> dengueseqstartstring <- c2s(dengueseqstart) # Convert the vector "dengueseqstart
↪" to a string of characters
> dengueseqstartstring                       # Print out the variable string of␣
↪characters "dengueseqstartstring"
  [1]
↪"agttgttagtctacgtggaccgacaagaacagtttcgaatcggaagcttgcttaacgtagttctaacagttttttattagagagcagatctctga
↪"
```

We can then find potential start and stop codons in the first 500 nucleotides of the DEN-1 Dengue virus sequence
by typing:

```
> findPotentialStartsAndStops(dengueseqstartstring)
  [[1]]
  [1]    7   53   58   64   78   93   95   96  137  141  224  225  234  236  246  255  264  295  298␣
↪318
  [21]  365  369  375  377  378  399  404  413  444  470  471  474  478
  [[2]]
  [1] "tag" "taa" "tag" "taa" "tag" "tga" "atg" "tga" "atg" "tga" "atg" "tga" "tga"
  [14] "atg" "tag" "taa" "tag" "tag" "atg" "atg" "atg" "tga" "taa" "atg" "tga" "tga
↪"
  [27] "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.

Reading frames

Potential start and stop codons in a 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 findPotentialStartsAndStops() 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.

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 following function plotPotentialStartsAndStops() plots the potential start and stop codons in the three different reading frames of a DNA sequence:

```
> plotPotentialStartsAndStops <- function(sequence)
  {
     # Define a vector with the sequences of potential start and stop codons
     codons <- c("atg", "taa", "tag", "tga")
     # Find the number of occurrences of each type of potential start or stop codon
```

```
    for (i in 1:4)
    {
        codon <- codons[i]
        # Find all occurrences of codon "codon" in sequence "sequence"
        occurrences <- matchPattern(codon, sequence)
        # Find the start positions of all occurrences of "codon" in sequence
↪"sequence"
        codonpositions <- attr(occurrences,"start")
        # Find the total number of potential start and stop codons in sequence
↪"sequence"
        numoccurrences <- length(codonpositions)
        if (i == 1)
        {
            # Make a copy of vector "codonpositions" called "positions"
            positions   <- codonpositions
            # Make a vector "types" containing "numoccurrences" copies of "codon"
            types       <- rep(codon, numoccurrences)
        }
        else
        {
            # Add the vector "codonpositions" to the end of vector "positions":
            positions   <- append(positions, codonpositions,␣
↪after=length(positions))
            # Add the vector "rep(codon, numoccurrences)" to the end of vector
↪"types":
            types       <- append(types, rep(codon, numoccurrences),␣
↪after=length(types))
        }
    }
    # Sort the vectors "positions" and "types" in order of position along the␣
↪input sequence:
    indices <- order(positions)
    positions <- positions[indices]
    types <- types[indices]
    # Make a plot showing the positions of the start and stop codons in the input␣
↪sequence:
    # Draw a line at y=0 from 1 to the length of the sequence:
    x  <- c(1,nchar(sequence))
    y <- c(0,0)
    plot(x, y, ylim=c(0,3), type="l", axes=FALSE, xlab="Nucleotide", ylab=
↪"Reading frame",
        main="Predicted start (red) and stop (blue) codons")
    segments(1,1,nchar(sequence),1)
    segments(1,2,nchar(sequence),2)
    # Add the x-axis at y=0:
    axis(1, pos=0)
    # Add the y-axis labels:
    text(0.9,0.5,"+1")
    text(0.9,1.5,"+2")
    text(0.9,2.5,"+3")
    # Draw in each predicted start/stop codon:
    numcodons <- length(positions)
    for (i in 1:numcodons)
    {
        position <- positions[i]
        type <- types[i]
        remainder <- (position-1) %% 3
        if    (remainder == 0) # +1 reading frame
        {
            if (type == "atg") { segments(position,0,position,1,lwd=1,col="red") }
            else               { segments(position,0,position,1,lwd=1,col="blue")}
        }
        else if (remainder == 1)
```

```
        {
            if (type == "atg") { segments(position,1,position,2,lwd=1,col="red") }
            else               { segments(position,1,position,2,lwd=1,col="blue")}
        }
        else if (remainder == 2)
        {
            if (type == "atg") { segments(position,2,position,3,lwd=1,col="red") }
            else               { segments(position,2,position,3,lwd=1,col="blue")}
        }
    }
}
```

To use this function, you will first need to copy and paste it into R.

For example, to plot the potential start and stop codons in the first 500 nucleotides of the DEN-1 Dengue virus genome, we type:

```
> plotPotentialStartsAndStops(dengueseqstartstring)
```


Predicted start (red) and stop (blue) codons
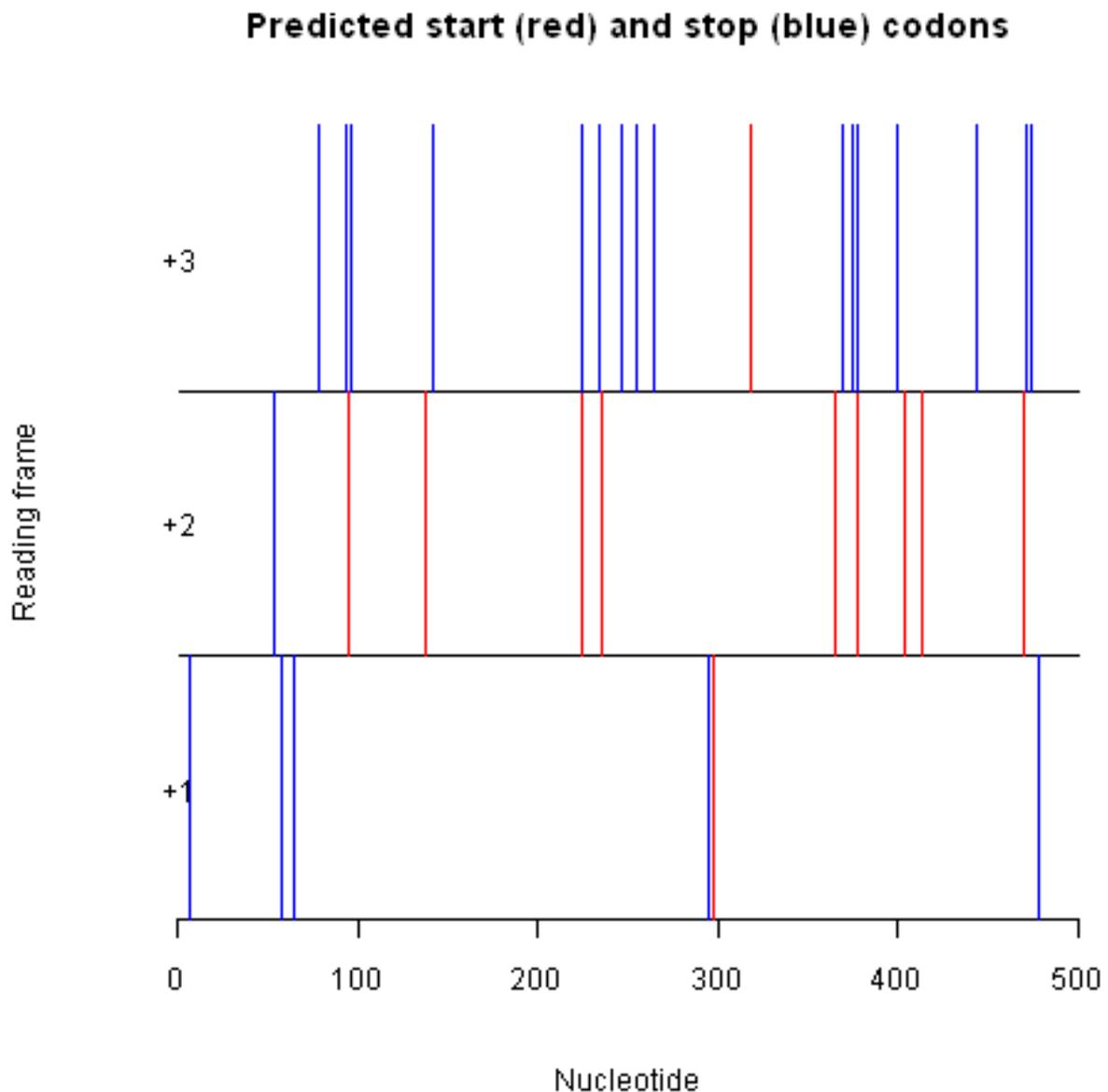
In the picture produced by plotPotentialStartsAndStops(), the x-axis represents the input sequence (*dengueseqstartstring* here). The potential start codons are represented by vertical red lines, and potential stop codons are

---

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.

The following function findORFsinSeq() finds ORFs in an input sequence:

```
> findORFsinSeq <- function(sequence)
  {
     require(Biostrings)
     # Make vectors "positions" and "types" containing information on the
→positions of ATGs in the sequence:
     mylist <- findPotentialStartsAndStops(sequence)
     positions <- mylist[[1]]
     types <- mylist[[2]]
     # Make vectors "orfstarts" and "orfstops" to store the predicted start and
→stop codons of ORFs
     orfstarts <- numeric()
     orfstops <- numeric()
     # Make a vector "orflengths" to store the lengths of the ORFs
     orflengths <- numeric()
     # Print out the positions of ORFs in the sequence:
     # Find the length of vector "positions"
     numpositions <- length(positions)
     # There must be at least one start codon and one stop codon to have an ORF.
     if (numpositions >= 2)
     {
        for (i in 1:(numpositions-1))
        {
           posi <- positions[i]
           typei <- types[i]
           found <- 0
           while (found == 0)
           {
              for (j in (i+1):numpositions)
              {
                 posj  <- positions[j]
                 typej <- types[j]
                 posdiff <- posj - posi
                 posdiffmod3 <- posdiff %% 3
                 # Add in the length of the stop codon
                 orflength <- posj - posi + 3
                 if (typei == "atg" && (typej == "taa" || typej == "tag" || typej
→== "tga") && posdiffmod3 == 0)
                    {
                       # Check if we have already used the stop codon at posj+2 in an
→ORF
                       numorfs <- length(orfstops)
                       usedstop <- -1
                       if (numorfs > 0)
                       {
                          for (k in 1:numorfs)
                          {
                             orfstopk <- orfstops[k]
                             if (orfstopk == (posj + 2)) { usedstop <- 1 }
```

```
                      }
                    }
                    if (usedstop == -1)
                    {
                       orfstarts <- append(orfstarts, posi,␣
→after=length(orfstarts))
                       orfstops <- append(orfstops, posj+2,␣
→after=length(orfstops)) # Including the stop codon.
                       orflengths <- append(orflengths, orflength,␣
→after=length(orflengths))
                    }
                    found <- 1
                    break
                  }
                  if (j == numpositions) { found <- 1 }
             }
          }
       }
    }
    # Sort the final ORFs by start position:
    indices <- order(orfstarts)
    orfstarts <- orfstarts[indices]
    orfstops <- orfstops[indices]
    # Find the lengths of the ORFs that we have
    orflengths <- numeric()
    numorfs <- length(orfstarts)
    for (i in 1:numorfs)
    {
       orfstart <- orfstarts[i]
       orfstop <- orfstops[i]
       orflength <- orfstop - orfstart + 1
       orflengths <- append(orflengths,orflength,after=length(orflengths))
    }
    mylist <- list(orfstarts, orfstops, orflengths)
    return(mylist)
  }
```

You will need to copy and paste this function into R before you can use it. For example, we can use it to find all ORFs in the sequence *s1*:

```
> s1 <- "aaaatgcagtaacccatgccc"
> findORFsinSeq(s1)
[[1]]
[1] 4

[[2]]
[1] 12

[[3]]
[1] 9
```

The function findORFsinSeq() 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 findORFsinSeq() 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 findORFsinSeq() to find the ORFs in the first 500 nucleotides of the DEN-1 Dengue virus genome sequence by typing:

```
> findORFsinSeq(dengueseqstartstring)
  [[1]]
  [1] 298 318
  [[2]]
  [1] 480 371
  [[3]]
  [1] 183  54
```

The result from findORFsinSeq() 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 following function "plotORFsinSeq()" plots the positions of ORFs in a sequence:

```
> plotORFsinSeq <- function(sequence)
  {
     # Make vectors "positions" and "types" containing information on the␣
  ↪positions of ATGs in the sequence:
     mylist <- findPotentialStartsAndStops(sequence)
     positions <- mylist[[1]]
     types <- mylist[[2]]
     # Make vectors "orfstarts" and "orfstops" to store the predicted start and␣
  ↪stop codons of ORFs
     orfstarts <- numeric()
     orfstops <- numeric()
     # Make a vector "orflengths" to store the lengths of the ORFs
     orflengths <- numeric()
     # Print out the positions of ORFs in the sequence:
     numpositions <- length(positions) # Find the length of vector "positions"
     # There must be at least one start codon and one stop codon to have an ORF.
     if (numpositions >= 2)
     {
        for (i in 1:(numpositions-1))
        {
           posi <- positions[i]
           typei <- types[i]
           found <- 0
           while (found == 0)
           {
              for (j in (i+1):numpositions)
              {
                 posj <- positions[j]
                 typej <- types[j]
                 posdiff <- posj - posi
                 posdiffmod3 <- posdiff %% 3
                 orflength <- posj - posi + 3 # Add in the length of the stop codon
                 if (typei == "atg" && (typej == "taa" || typej == "tag" || typej␣
  ↪== "tga") && posdiffmod3 == 0)
                 {
                    # Check if we have already used the stop codon at posj+2 in an␣
  ↪ORF
                    numorfs <- length(orfstops)
                    usedstop <- -1
                    if (numorfs > 0)
                    {
                       for (k in 1:numorfs)
                       {
                          orfstopk <- orfstops[k]
                          if (orfstopk == (posj + 2)) { usedstop <- 1 }
                       }
                    }
                    if (usedstop == -1)
                    {
```

```
                    orfstarts <- append(orfstarts, posi,␣
→after=length(orfstarts))
                    orfstops <- append(orfstops, posj+2,␣
→after=length(orfstops)) # Including the stop codon.
                    orflengths <- append(orflengths, orflength,␣
→after=length(orflengths))
                }
                found <- 1
                break
            }
            if (j == numpositions) { found <- 1 }
        }
      }
    }
  }
  # Sort the final ORFs by start position:
  indices <- order(orfstarts)
  orfstarts <- orfstarts[indices]
  orfstops <- orfstops[indices]
  # Make a plot showing the positions of ORFs in the input sequence:
  # Draw a line at y=0 from 1 to the length of the sequence:
  x <- c(1,nchar(sequence))
  y <- c(0,0)
  plot(x, y, ylim=c(0,3), type="l", axes=FALSE, xlab="Nucleotide", ylab=
→"Reading frame", main="Predicted ORFs")
  segments(1,1,nchar(sequence),1)
  segments(1,2,nchar(sequence),2)
  # Add the x-axis at y=0:
  axis(1, pos=0)
  # Add the y-axis labels:
  text(0.9,0.5,"+1")
  text(0.9,1.5,"+2")
  text(0.9,2.5,"+3")
  # Make a plot of the ORFs in the sequence:
  numorfs <- length(orfstarts)
  for (i in 1:numorfs)
  {
     orfstart <- orfstarts[i]
     orfstop <- orfstops[i]
     remainder <- (orfstart-1) %% 3
     if    (remainder == 0) # +1 reading frame
     {
        rect(orfstart,0,orfstop,1,col="cyan",border="black")
     }
     else if (remainder == 1)
     {
        rect(orfstart,1,orfstop,2,col="cyan",border="black")
     }
     else if (remainder == 2)
     {
        rect(orfstart,2,orfstop,3,col="cyan",border="black")
     }
  }
}
```

To use this function, you will first need to copy and paste it into R. You can then use this function to plot the positions of the ORFs in *dengueseqstartstring* by typing:

```
> plotORFsinSeq(dengueseqstartstring)
```

Predicted ORFs
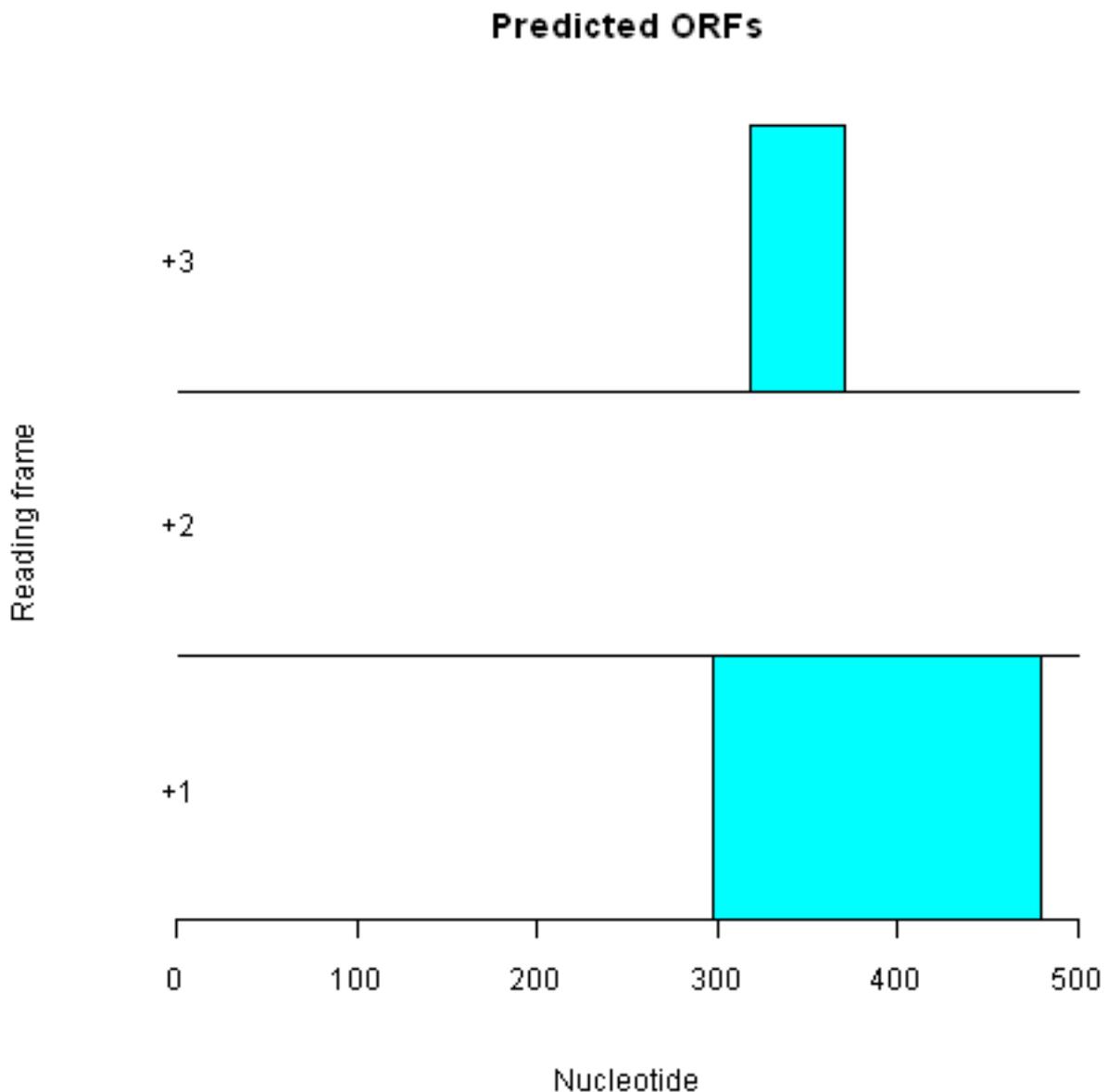
The picture produced by plotORFsinSeq() represents the two ORFs in the first 500 nucleotides of the lambda genome as blue rectangles.

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 plotPotentialStartsAndStops() above.

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 findORFsinSeq() 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)
> myorf # Print out the sequence of "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 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 the SeqinR package. 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
> myorfvector              # Print out the value of "myorfvector"
[1] "a" "t" "g" "c" "a" "g" "t" "a" "a"
> 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.

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 findORFsinSeq() 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 calculate the complement of a sequence, and the rev() function 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 "AAAATGCTTAAACCATTGCCC", and we want to find the reverse strand sequence, we type:

```
> forward <- "AAAATGCTTAAACCATTGCCC"
> forwardvector <- s2c(forward)              # Convert the string of characters
→to a vector
> forwardvector                              # Print out the vector containing
→the forward strand sequence
 [1] "A" "A" "A" "A" "T" "G" "C" "T" "T" "A" "A" "A" "C" "C" "A" "T" "T" "G" "C" "C
→" "C"
> reversevector <- rev(comp(forwardvector))  # Find the reverse strand sequence,
→by finding the reverse complement
> reversevector                              # Print out the vector containing
→the reverse strand sequence
 [1] "g" "g" "g" "c" "a" "a" "t" "g" "g" "t" "t" "t" "a" "a" "g" "c" "a" "t" "t" "t
→" "t"
> reverse <- c2s(reversevector)              # Convert the vector to a string of
→characters
> reverse                                    # Print out the string of
→characters containing the reverse strand sequence
[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:

```
> complement <- comp(forwardvector)          # Find the complement of the␣
↪forward strand sequence
> reversevector <- rev(complement)           # Reverse the order of the letters␣
↪in sequence "complement", to
                                             # find the reverse strand␣
↪sequence (the reverse complement sequence)
```

Once we have inferred the reverse strand sequence, we can then use the findORFsinSeq() function to find ORFs in the reverse strand sequence:

```
> findORFsinSeq(reverse)
[[1]]
[1] 6

[[2]]
[1] 14

[[3]]
[1] 9
```

This indicates that there is one ORF of length 9 bp in the reverse strand of sequence "AAAATGCTTAAAC-CATTGCCC", 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.

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 may 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.

y 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.

Identifying significant open reading frames

How long does an ORF need to be in order for us to be confident that it probably corresponds to a real gene? This is a difficult question.

One approach to answer this is to ask: what is the longest ORF found in a random sequence of the same length and nucleotide composition as our original sequence?

The ORFs in a random sequence do not correspond to real genes, but are just due to potential start and stop codons that have occurred by chance in those sequences (since, by definition, a random sequence is one that was generated randomly, rather than by evolution as in a real organism).

Thus, by looking at the lengths of ORFs in the random sequence, we can see what is the longest ORF that is likely to occur by chance alone.

But where can we get random sequences from? In a previous chapter, you learnt that you can generate random sequences using a multinomial model with a particular probability of each letter (a particular probability of A, C, G, and T in the case of random DNA sequences).

In that previous chapter, we used the function generateSeqsWithMultinomialModel() to generate random sequences using a multinomial model in which the probability of each letter is set equal to the fraction of an input sequence that consists of that letter. This function takes two arguments, the input sequence, and the number of the random sequences that you want to generate.

For example, to create a random sequence of the same length as 'AAAATGCTTAAACCATTGCCC', using a multinomial model in which the probabilities of A, C, G and T are set equal to their fractions in this sequence, we copy and paste the generateSeqsWithMultinomialModel() into R, then type:

```
> myseq    <- "AAAATGCTTAAACCATTGCCC"
> generateSeqsWithMultinomialModel(myseq, 1) # Generate one random sequence using
↪the multinomial model
[1] "AACAATTCTACCCTATTCTTC"
```

We can then use the findORFsinSeq() function to find ORFs in this random sequence. If we repeat this 10 times, we can find the lengths of the ORFs found in the 10 random sequences. We can then compare the lengths of the ORFs found in the original sequence, to the lengths of the ORFs found in the random sequences.

For example, to compare the lengths of ORFs found in the DEN-1 Dengue virus genome sequence *dengueseq* to the lengths of ORFs found in 10 random sequences generated using a multinomial model in which the probabilities of the four bases are set equal to their fractions in the DEN-1 Dengue virus sequence, we type:

```
> dengueseqstring <- c2s(dengueseq)              # Convert the Dengue sequence to a
↪string of characters
> mylist <- findORFsinSeq(dengueseqstring)    # Find ORFs in "dengueseqstring"
> orflengths <- mylist[[3]]                      # Find the lengths of ORFs in
↪"dengueseqstring"
> randseqs <- generateSeqsWithMultinomialModel(dengueseqstring, 10) # Generate 10
↪random sequences using the multinomial model
> randseqorflengths <- numeric()                 # Tell R that we want to make a new
↪vector of numbers
> for (i in 1:10)
  {
     print(i)
     randseq <- randseqs[i]                      # Get the ith random sequence
     mylist <- findORFsinSeq(randseq)            # Find ORFs in "randseq"
     lengths <- mylist[[3]]                       # Find the lengths of ORFs in
↪"randseq"
     randseqorflengths <- append(randseqorflengths, lengths,
↪after=length(randseqorflengths))
  }
```

This may take a little time to run, however, the for loop above prints out the value of *i* each time that it starts the loop, so you can see how far it has got.

In the code above, we retrieve the lengths of the ORFs found by function findORFsinSeq() by taking the third element of the list returned by this function. As mentioned above, the third element of the list returned by this function is a vector containing the lengths of all the ORFs found in the input sequence.

We can then plot a histogram of the lengths of the ORFs in the real DEN-1 Dengue genome sequence (*orflengths*) beside a histogram of the lengths of the ORFs in the 10 random sequences (*randseqorflengths*):

```
> par(mfrow = c(1,2))                    # Make a picture with two plots side-by-
↪side (one row, two columns)
> bins <- seq(0,11000,50)                # Set the bins for the histogram
> hist(randseqorflengths, breaks=bins, col="red", xlim=c(0,1000))
> hist(orflengths, breaks=bins, col="red", xlim=c(0,1000))
```
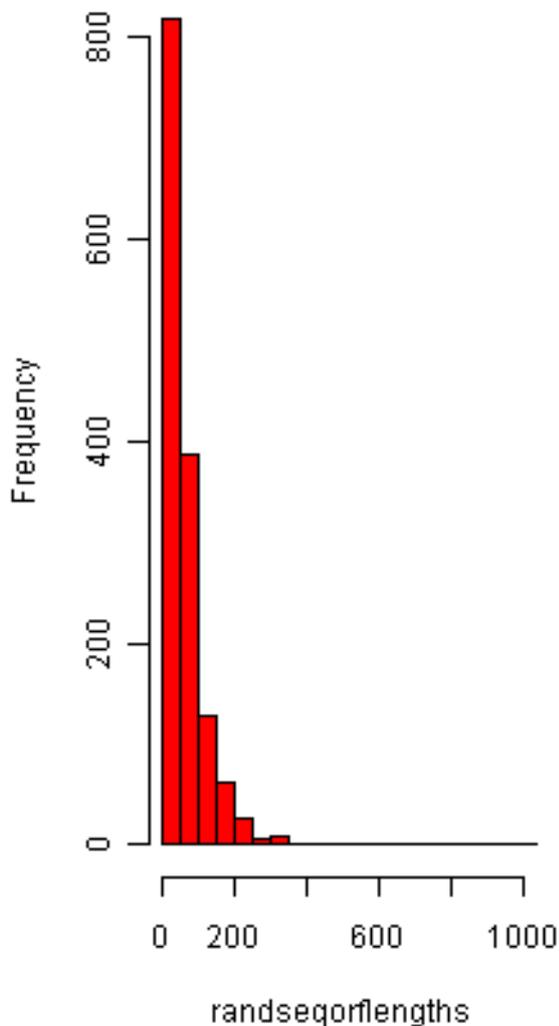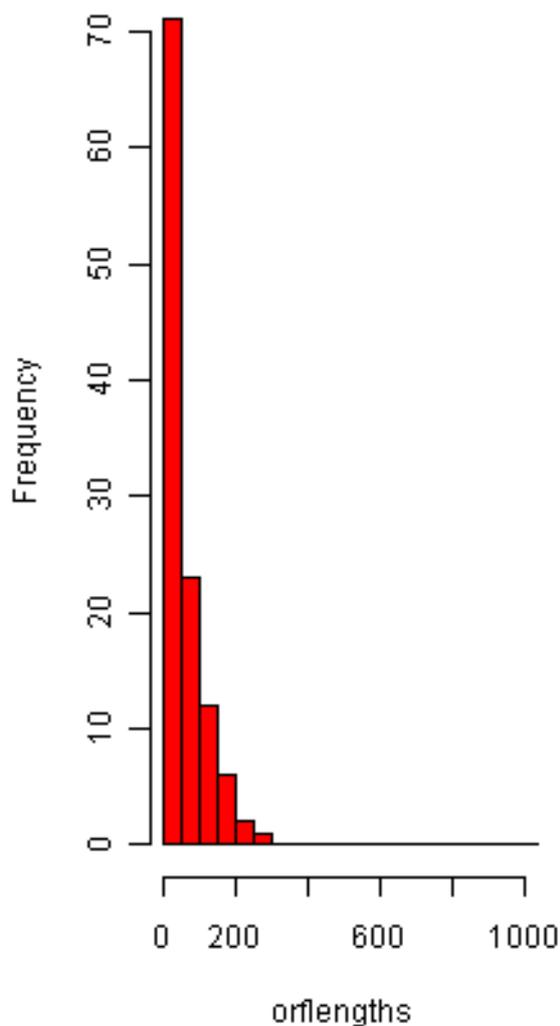

In other words, the histogram of the lengths of the ORFs in the 10 random sequences gives us an idea of the length distribution of ORFs that you would expect by chance alone in a random DNA sequence (generated by a multinomial model in which the probabilities of the four bases are set equal to their frequencies in the DEN-1 Dengue virus genome sequence).

We can calculate the longest of the ORFs that occurs in the random sequences, using the max() function, which can be used to find the largest element in a vector of numbers:

```
> max(randseqorflengths)
[1] 342
```

This indicates that the longest ORF that occurs in the random sequences is 342 nucleotides long. Thus, it is possible for an ORF of up to 342 nucleotides to occur by chance alone in a random sequence of the same length and roughly the same composition as the DEN-1 Dengue virus genome.

Therefore, we could use 342 nucleotides as a threshold, and discard all ORFs found in the DEN-1 Dengue virus

genome that are shorter than this, under the assumption that they probably arose by chance and probably do not correspond to real genes. How many ORFs would be left in the DEN-1 Dengue virus genome sequence if we used 342 nucleotides as a threshold?

```
> summary(orflengths > 342)
    Mode   FALSE    TRUE    NA's
 logical     115       1       0
```

If we did use 342 nucleotides as a threshold, there would only be 1 ORF left in the DEN-1 Dengue virus genome. Some of the 115 shorter ORFs that we discarded may correspond to real genes.

Generally, we don't want to miss many real genes, we may want to use a more tolerant threshold. For example, instead of discarding all Dengue ORFs that are shorter than the longest ORF found in the 10 random sequences, we could discard all Dengue ORFs that are shorter than the longest 99% of ORFs in the random sequences.

We can use the quantile() function to find *quantiles* of a set of numbers. The *99th* quantile for a set of numbers is the value *x* such that 99% of the numbers in the set have values less than *x*.

For example, to find the 99th quantile of *randomseqorflengths*, we type:

```
> quantile(randseqorflengths, probs=c(0.99))
99%
248.07
```

This means that 99% of the ORFs in the random sequences have lengths less than 248 nucleotides long. In other words, the longest of the longest 99% of ORFs in the random sequences is 248 nucleotides.

Thus, if we were using this as a threshold, we would discard all ORFs from the DEN-1 Dengue genome that are 248 nucleotides or shorter. This will result in fewer ORFs being discarded than if we used the more stringent threshold of 342 nucleotides (ie. discarding all ORFs of <342 nucleotides), so we will probably have discarded fewer ORFs that correspond to real genes. Unfortunately, it probably means that we will also have kept more false positives at the same time, that is, ORFs that do not correspond to real genes.

Summary

In this practical, you will have learnt to use the following R functions:

1. substring() for cutting out a substring of a string of characters (eg. a subsequence of a DNA sequence)

2. rev() for reversing the order of the elements in a vector

3. hist() to make a histogram plot

4. max() to find the largest element in a vector of numbers

5. quantile() to find quantiles of a set of numbers that correspond to particular probabilities

All of these functions belong to the standard installation of R.

You have also learnt the following R functions that belong to the bioinformatics packages:

1. tablecode() in the SeqinR package for viewing the genetic code

2. MatchPattern() in the Biostrings package for finding all occurrences of a motif in a sequence

3. translate() in the SeqinR package to get the predicted protein sequence for an ORF

4. s2c() in the SeqinR package to convert a sequence stored as a string of characters into a vector

5. c2s() in the SeqinR package to convert a sequence stored in a vector into a string of characters

6. comp() in the SeqinR package to find the complement of a DNA sequence

Links and Further Reading

Some links are included here for further reading.

For background reading on computational gene-finding, it is recommended to read Chapter 2 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.

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.

There is also a very nice chapter on "Analyzing Sequences", which includes examples of using the SeqinR and Biostrings packages 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).

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.

Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the Matlab case study on the *Haemophilus influenzae* genome (www.computational-genomics.net/case_studies/haemophilus_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/).

Thank you to Jean Lobry and Simon Penel for helpful advice on using the SeqinR package.

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 Computational Gene-finding.

Q1. How many ORFs are there on the forward strand of the DEN-1 Dengue virus genome (NCBI accession NC_001477)?

Q2. What are the coordinates of the rightmost (most 3', or last) ORF in the forward strand of the DEN-1 Dengue virus genome?

Q3. What is the predicted protein sequence for the rightmost (most 3', or last) ORF in the forward strand of the DEN-1 Dengue virus genome?

Q4. How many ORFs are there of 30 nucleotides or longer in the forward strand of the DEN-1 Dengue virus genome sequence?

Q5. How many ORFs longer than 248 nucleotides are there in the forward strand of the DEN-1 Dengue genome sequence?

Q6. If an ORF is 248 nucleotides long, what length in amino acids will its predicted protein sequence be?

Q7. How many ORFs are there on the forward strand of the rabies virus genome (NCBI accession NC_001542)?
    Note: rabies virus is the virus responsible for rabies, which is classified by the WHO as a neglected tropical disease.

Q8. What is the length of the longest ORF among the 99% of longest ORFs in 10 random sequences of the same lengths and composition as the rabies virus genome sequence?

Q9. How many ORFs are there in the rabies virus genome that are longer than the threshold length that you found in Q8?

Comparative Genomics

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 (ie. the same *gene content*)? Since all life on earth shared a common ancestor at some point, any two species, for example, human and a fruitfly, must have descended from a common ancestor species.

Since the time of the common ancestor of two species (eg. 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.

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 (eg. human and mouse), it is useful to analyse 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 (eg. 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:

```
> library("biomaRt") # Load the biomaRt package in R
> listMarts()        # List all databases that can be queried
                          biomart
 1                         ensembl
 2                             snp
 3              functional_genomics
 4                            vega
 5               bacterial_mart_9
 6                   fungal_mart_9
 7             fungal_variations_9
 8                  metazoa_mart_9
 9            metazoa_variations_9
 10                   plant_mart_9
 11             plant_variations_9
 12                 protist_mart_9
 13           protist_variations_9
 14                           msd
 15                          htgt
 16                       REACTOME
 17                   WS220-testing
 ...
                                            version
```

```
1                             ENSEMBL GENES 62 (SANGER UK)
2                         ENSEMBL  VARIATION 62 (SANGER UK)
3              ENSEMBL FUNCTIONAL GENOMICS 62 (SANGER UK)
4                                      VEGA 42  (SANGER UK)
5                        ENSEMBL BACTERIA 9 (EBI UK)
6                           ENSEMBL FUNGI 9 (EBI UK)
7                 ENSEMBL FUNGI VARIATION 9 (EBI UK)
8                         ENSEMBL METAZOA 9 (EBI UK)
9               ENSEMBL METAZOA VARIATION 9 (EBI UK)
10                         ENSEMBL PLANTS 9 (EBI UK)
11             ENSEMBL PLANTS VARIATION 9 (EBI UK)
12                        ENSEMBL PROTISTS 9 (EBI UK)
13           ENSEMBL PROTISTS VARIATION 9 (EBI UK)
14                                      MSD (EBI UK)
15            WTSI MOUSE GENETICS PROJECT (SANGER UK)
16                                 REACTOME (CSHL US)
17                             WORMBASE 220 (CSHL US)
...
```

The names of the databases are listed, and then an explanation of what each database is, and what is the version of the database.

You will see that 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.

You can see above that "biomaRt" tells you which version of each database can be searched, for example, the version of the main Ensembl database that can be searched is Ensembl 62 (the current release), while the version of the Ensembl Protists database that can be searched is Ensembl Protists 9.

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 useMart() function from the "biomaRt" package:

```
> ensemblprotists <- useMart("protist_mart_9") # Specify that we want to query the
→Ensembl Protists database
```

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 useMart()), you can type:

```
> listDatasets(ensemblprotists)           # List the data sets in the Ensembl
→Protists database
          dataset                                    description
1      pramorum_eg_gene        Phytophthora ramorum genes (Phyra1_1)
2        pvivax_eg_gene               Plasmodium vivax genes (EPr 2)
3   pfalciparum_eg_gene          Plasmodium falciparum genes (2.1.4)
4  ptricornutum_eg_gene     Phaeodactylum tricornutum genes (Phatr2)
5      pchabaudi_eg_gene          Plasmodium chabaudi genes (May_2010)
6   ddiscoideum_eg_gene Dictyostelium discoideum genes (dictybase.01)
7        lmajor_eg_gene    Leishmania major strain Friedlin genes (1)
...
   version
1      Phyra1_1
2         EPr 2
3         2.1.4
4        Phatr2
5       May_2010
6  dictybase.01
7             1
...
```

You will see a long list of the organisms for which the Ensembl Protists database has genome data, including *Plasmodium vivax* and *Plasmodium falciparium* (which cause 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] 292
> attributenames[1:10]                       # Print out the first 10 entries in␣
→vector "attributenames"
 [1] "ensembl_gene_id"            "ensembl_transcript_id"
 [3] "ensembl_peptide_id"         "canonical_transcript_stable_id"
 [5] "description"                "chromosome_name"
 [7] "start_position"             "end_position"
 [9] "strand"                     "band"
> attributedescriptions[1:10]                # Print out the first 10 entries in␣
→vector "attributedescriptions"
> attributedescriptions[1:10]
 [1] "Ensembl Gene ID"           "Ensembl Transcript ID"
 [3] "Ensembl Protein ID"        "Canonical transcript stable ID(s)"
 [5] "Description"               "Chromosome Name"
 [7] "Gene Start (bp)"           "Gene End (bp)"
 [9] "Strand"                    "Band"
```

This gives us a very long list of 292 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:

```
> leishmaniagenenames <- leishmaniagenes[[1]] # Get the vector of the names of all␣
↪L. major genes
> length(leishmaniagenenames)
[1] 9379
> leishmaniagenenames[1:10]
[1] "LmjF.01.0010" "LmjF.01.0020" "LmjF.01.0030" "LmjF.01.0040" "LmjF.01.0050"
[6] "LmjF.01.0060" "LmjF.01.0070" "LmjF.01.0080" "LmjF.01.0090" "LmjF.01.0100"
```

This tells us that there are 9379 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 (eg. protein-coding, pseudogene, etc.):

```
> leishmaniagenes2 <- 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 nontranslating_cds     protein_coding          pseudogene
           84                  2               8310                  90
         rRNA             snoRNA              snRNA                tRNA
           63                741                  6                  83
```

This tells us that there are 8310 protein-coding genes, 90 pseudogenes, and various types of RNA genes (tRNA genes, rRNA genes, snRNA genes, etc.). Thus, there are 8310 human protein-coding genes.

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 analysing 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* (8310; 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)
> pfalciparumgenenames <- pfalciparumgenes[[1]] # Get the names of the P.
↪falciparum genes
> length(pfalciparumgenenames)                  # Get the number of P. falciparum
↪genes
[1] 6213
> pfalciparumgenebiotypes <- pfalciparumgenes[[2]] # Get the types of the P.
↪falciparum genes
> table(pfalciparumgenebiotypes)
  pfalciparumgenebiotypes
          ncRNA non_translating_cds        protein_coding                rRNA
            712                   1                  5428                  24
          snRNA                tRNA
              3                  45
```

This tells us that there are 5428 *Plasmodium falciparum* protein-coding genes in Ensembl. That is, *Plasmodium falciparum* seems to have less protein-coding genes than *Leishmania major* (8310 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.

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 homologues (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 *orthologues* (related genes that shared a common ancestor in the ancestor of *Leishmania* and *Plasmodium*, and arose due to the *Leishmania - Plasmodium* speciation event) or *paralogues* (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_gene", which is the *Plasmodium falciparum* orthologue of a *Leishmania major* gene.

Another possible feature that you can search for is "pfalciparum_eg_orthology_type", which describes the type of orthology relationship between a particular *Leishmania major* gene and its *Plasmodium falciparum* orthologue. For example, if a particular *Leishmania major* gene has two *Plasmodium falciparum* orthologues, the relationship between the *Leishmania major* gene and each of the *Plasmodium falciparum* orthologues 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 paralogues of each other) are both orthologues of the same *Leishmania major* gene.

Therefore, we can retrive the Ensembl identifiers of the *Plasmodium falciparum* orthologues of all *Leishmania major* genes by typing:

```
> leishmaniagenes <- getBM(attributes = c("ensembl_gene_id", "pfalciparum_eg_gene",
    "pfalciparum_eg_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* orthologues, 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* orthologues, and their orthology types, by typing:

```
> leishmaniagenenames <- leishmaniagenes[[1]]      # Get the names of all
↪Leishmania major genes
> leishmaniaPforthologues <- leishmaniagenes[[2]]  # Get the P. falciparum
↪orthologues of all L. major genes
> leishmaniaPforthologuetypes <- leishmaniagenes[[3]] # Get the orthology
↪relationship type
> leishmaniagenenames[1:10]
  [1] "LmjF.34.2510" "LmjF.14.0650" "LmjF.14.0650" "LmjF.14.0670" "LmjF.14.0670"
  [6] "LmjF.14.0680" "LmjF.14.0680" "LmjF.14.0710" "LmjF.14.0710" "LmjF.36.2350"
> leishmaniaPforthologues[1:10]
  [1] ""         "PFA0455c" "PFI0980w" "PFA0455c" "PFI0980w" "PFA0455c"
  [7] "PFI0980w" "PFA0455c" "PFI0980w" ""
> leishmaniaPforthologuetypes[1:10]
  [1] ""                  "ortholog_many2many" "ortholog_many2many"
  [4] "ortholog_many2many" "ortholog_many2many" "ortholog_many2many"
  [7] "ortholog_many2many" "ortholog_many2many" "ortholog_many2many"
  [10] ""
```

Not all *Leishmania major* genes have *Plasmodium falciparum* orthologues; this is why when we print out the first 10 elements of the vector *leishmaniaPforthologues*, some of the elements are empty.

To find out how many *Leishmania major* genes have orthologues in *Plasmodium falciparum*, we can first find the indices of the elements of the vector *leishmaniaPforthologues* 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] 7723
```

This tells us that 7723 *Leishmania major* genes do not have *Plasmodium falciparum* orthologues.

How many of the 7723 *Leishmania major* genes that do not have *Plasmodium falaciparum* orthologues 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* orthologues).

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 (eg. protein-coding, pseudogene etc.), and the third element has a vector of the *Plasmodium falciparum* orthologues' names. We can therefore find out how many protein-coding *Leishmania major* genes lack *Plasmodium falciparum* orthologues by typing:

```
> leishmaniagenenames <- leishmaniagenes3[[1]]
> leishmaniagenebiotypes <- leishmaniagenes3[[2]]
> leishmaniaPforthologues <- leishmaniagenes3[[3]]
> myindex <- leishmaniaPforthologues=="" & leishmaniagenebiotypes=="protein_coding"
> leishmaniagenenames2 <- leishmaniagenenames[myindex]
> length(leishmaniagenenames2)
[1] 6654
```

This tells us that there are 6654 *Leishmania major* protein-coding genes that lack *Plasmodium falciparum* orthologues.

Summary

In this practical, you will have learnt to use the following R functions:

1. useMart() to select a database to query (in the biomaRt package)
2. useDataset() to select a data set in a database to query (in the biomaRt package)
3. listDatasets() to get a list of all data sets in a database (in the biomaRt package)
4. listAttributes() to get a list of all features of a data set (in the biomaRt package)
5. getBM() to make a query on a database (in the biomaRt package)
6. merge() to merge R list objects that contain some named elements in common

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.

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/).

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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.

Q1. 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.

Q2. How many of the *Mycobacterium ulcerans* Ensembl genes are protein-coding genes?

Q3. How many *Mycobacterium ulcerans* protein-coding genes have *Mycobacterium leprae* orthologues?
  Note: *Mycobacterium leprae* is the bacterium that causes leprosy, which is classified by the WHO as a neglected tropical disease.

Q4. How many of the*Mycobacterium ulcerans* protein-coding genes have one-to-one orthologues in *Mycobacterium leprae*?

Q5. How many *Mycobacterium ulcerans* genes have Pfam domains?

Q6. What are the top 5 most common Pfam domains in *Mycobacterium ulcerans* genes?

Q7. How many copies of each of the top 5 domains found in Q6 are there in the *Mycobacterium ulcerans* protein set?

Q8. How many of copies are there in the *Mycobacterium lepraae* protein set, of each of the top 5 *Mycobacterium ulcerans* Pfam protein domains?

Q9. Are the numbers of copies of some domains different in the two species?

Q10. Of the differences found in Q9, are any of the differencess statistically significant?

Hidden Markov Models

A little more about R

In previous practicals, you learnt 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:

```
> myvector <- numeric()                     # Create a vector "myvector" for storing
↪numbers
> for (i in 1:10) { myvector[i] <- i*i } # Fill in the values in the vector
↪"myvector"
> myvector                                  # Print out the vector "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:

```
> for (i in 1:10) { avector[i] <- i*i }   # Try to store values in the vector
↪"avector"
Error in avector[i] <- i * i : object 'avector' not found
```

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:

```
> 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:

```
> 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 (ie. 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:

```
> 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
```

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 p_A, p_C, p_G, and p_T 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 p_A, p_C, p_G, and p_T. For example, say we may create a multinomial model where p_A=0.2, p_C=0.3, p_G=0.3, and p_T=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 $p_A + p_C + p_G + p_T$ = 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 labelled "A", "T", "G" and "C", where the p_A, p_T, p_G and p_C are the fractions of the wheel taken up by the slices with these four labels. If you spin the arrow attached to the centre of the roulette wheel, the probability that it will stop in the slice with a particular label (eg. the slice labelled "A") only depends on the fraction of the wheel taken up by that slice (p_A here; see the picture below).

$$p_A\text{=}0.2,\ p_C\text{=}0.3,\ p_G\text{=}0.3,\ p_T\text{=}0.2$$
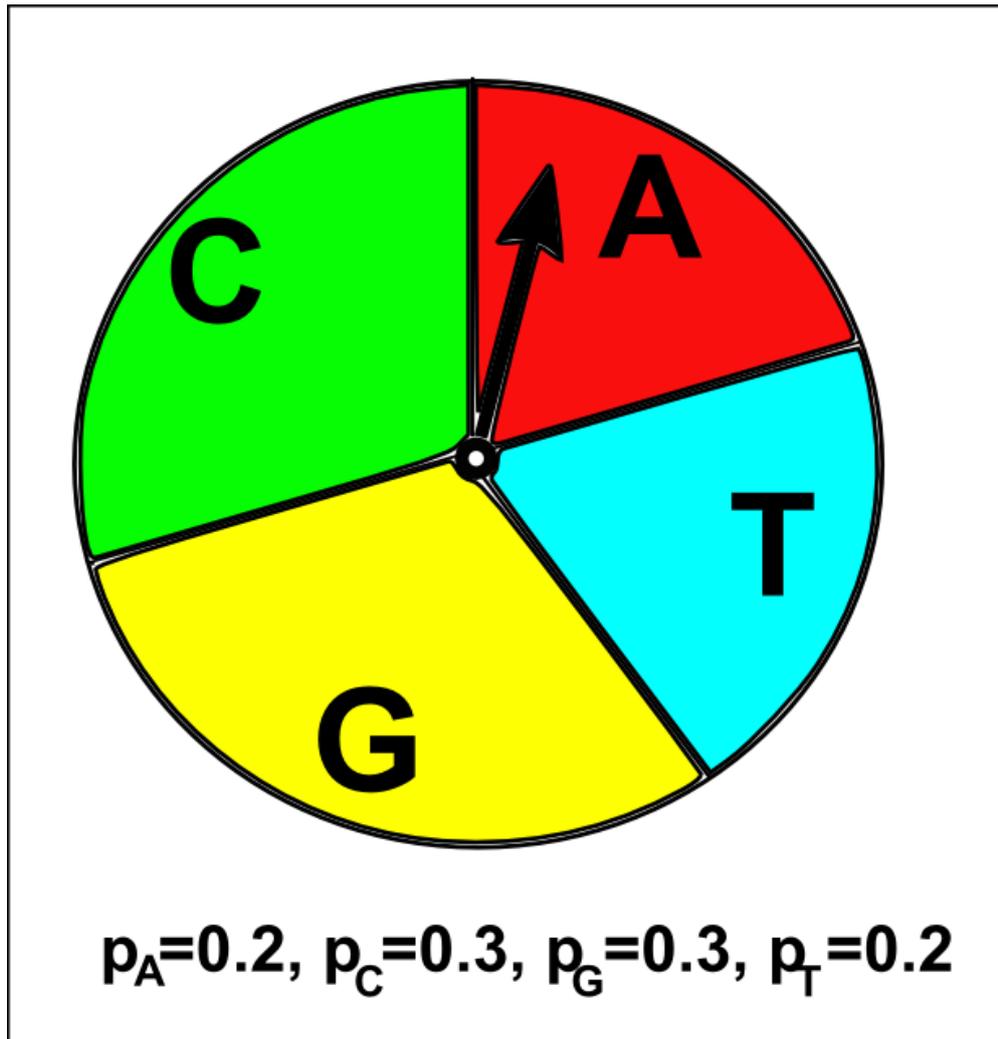
Generating a DNA sequence using a multinomial model

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 p_A, p_C, p_G, and p_T 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 p_A=0.2, p_C=0.3, p_G=0.3, and p_T=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:

```
> nucleotides    <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
> probabilities1 <- c(0.2, 0.3, 0.3, 0.2) # Set the values of the probabilities
> seqlength      <- 30                     # Set the length of the sequence
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities1) # Generate a
↪sequence
```

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

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:

```
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities1) # Generate another␣
↪sequence
[1] "T" "G" "C" "T" "A" "T" "G" "G" "T" "C" "G" "A" "A" "T" "G" "G" "G" "G" "C"
[20] "T" "A" "A" "C" "C" "G" "A" "G" "G" "C" "G"
```

In the same way, we can generate a sequence using a different multinomial model, where p_A=0.1, p_C=0.41, p_G=0.39, and p_T=0.1:

```
> probabilities2 <- c(0.1, 0.41, 0.39, 0.1) # Set the values of the probabilities␣
↪for the new model
> sample(nucleotides, seqlength, rep=TRUE, prob=probabilities2) # Generate a␣
↪sequence
[1] "G" "C" "C" "T" "C" "C" "C" "C" "G" "G" "G" "G" "G" "A" "C" "C" "C" "A" "G"
[20] "A" "G" "C" "T" "C" "G" "G" "C" "G" "G" "C"
```

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 p_C and G_T are higher for this second model than for the first model (p_C=0.41 and G_T=0.39 in the second model, versus p_C=0.3 and G_T=0.3 in the first model). That is, in the second multinomial model we are using a roulette wheel that has large slices labelled "C" and "G", while in the first multinomial model we were using a roulette wheel with relatively smaller slices labelled "C" and "G" (see the picture below).

Wheel 1	Wheel 2
p_A=0.2, p_C=0.3, p_G=0.3, p_T=0.2	p_A=0.1, p_C=0.41, p_G=0.39, p_T=0.1
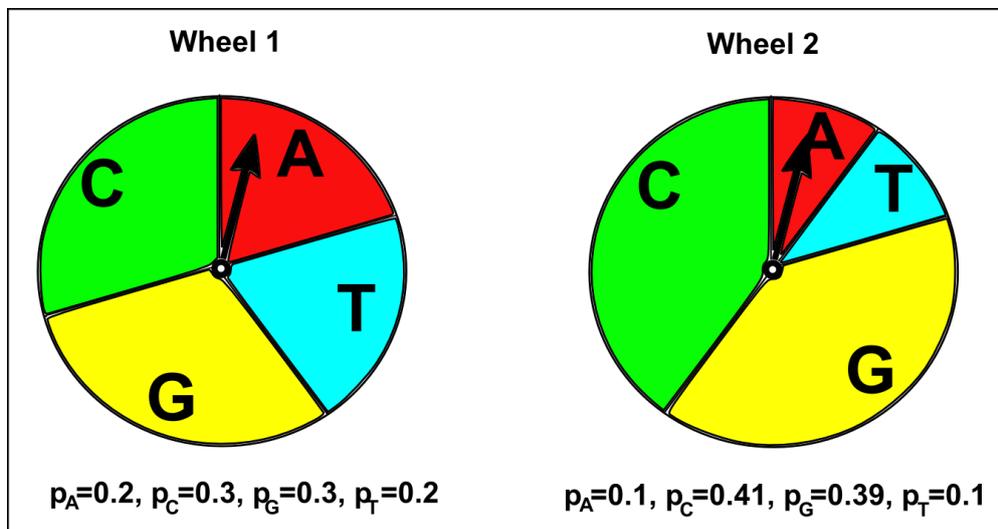
A Markov model of DNA sequence evolution

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 p_A, p_C, p_G, and p_T), 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 (p_A 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 p_A, p_C, p_G, and p_T of choosing "A", "C", "G", or "T", respectively (eg. p_A=0.2, p_C=0.3, p_G=0.3, and p_T=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. p_A=0.1, p_C=0.41, p_G=0.39, and p_T=0.1).

A Markov sequence model is like having four different roulette wheels, labelled "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 labelled "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 p_A, p_T, p_G and p_C. 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 centre 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 labelled with that nucleotide (p_A here; see the picture below).
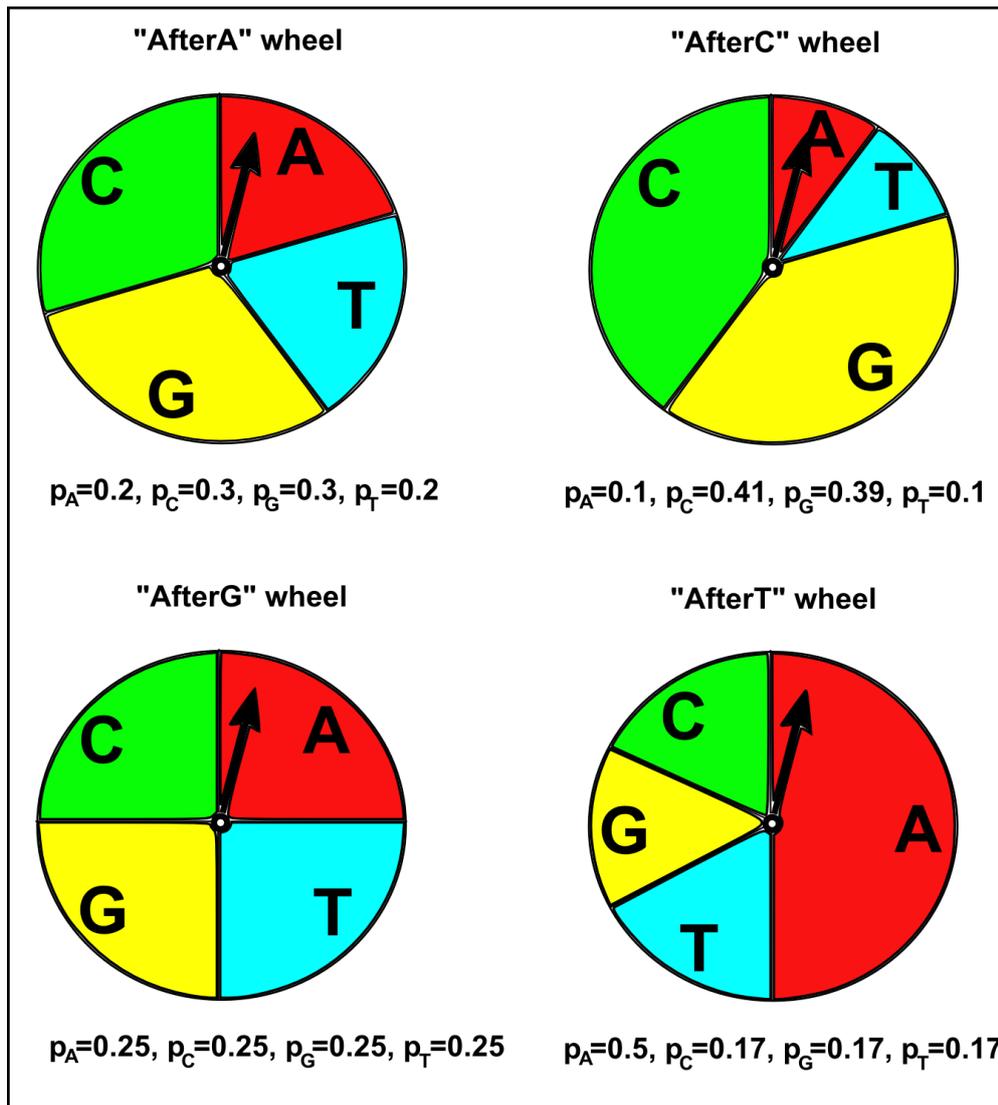
The transition matrix for a Markov model

A multinomial model of DNA sequence evolution just has four parameters: the probabilities p_A, p_C, p_G, and p_T. In contrast, a Markov model has many more parameters: four sets of probabilities p_A, p_C, p_G, and p_T, that differ according to whether the previous nucleotide was "A", "G", "T" or "C". The symbols p_{AA}, p_{AC}, p_{AG}, and p_{AT} are usually used to represent the four probabilities for the case where the previous nucleotide was "A", the symbols p_{CA}, p_{CC}, p_{CG}, and p_{CT} 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:

```
> nucleotides          <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
> afterAprobs <- c(0.2, 0.3, 0.3, 0.2)          # Set the values of the
→probabilities, where the previous nucleotide was "A"
> afterCprobs <- c(0.1, 0.41, 0.39, 0.1)        # Set the values of the
→probabilities, where the previous nucleotide was "C"
> afterGprobs <- c(0.25, 0.25, 0.25, 0.25)      # Set the values of the
→probabilities, where the previous nucleotide was "G"
```

```
> afterTprobs <- c(0.5, 0.17, 0.17, 0.17)        # Set the values of the
→probabilities, where the previous nucleotide was "T"
> mytransitionmatrix <- matrix(c(afterAprobs, afterCprobs, afterGprobs,
→afterTprobs), 4, 4, byrow = TRUE) # Create a 4 x 4 matrix
> rownames(mytransitionmatrix) <- nucleotides
> colnames(mytransitionmatrix) <- nucleotides
> mytransitionmatrix                              # Print out the transition matrix
  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 p_A, p_C, p_G, and p_T 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 (p_{AC}) 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.

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 Π_A, Π_C, Π_G, and Π_T are used to represent the probabilities of choosing "A", "C", "G", or "T" at the first position.

We can define an R function generatemarkovseq() to generate a DNA sequence using a particular Markov model:

```
> generatemarkovseq <- function(transitionmatrix, initialprobs, seqlength)
{
nucleotides      <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
mysequence       <- character()           # Create a vector for storing the new
→sequence
# Choose the nucleotide for the first position in the sequence:
firstnucleotide <- sample(nucleotides, 1, rep=TRUE, prob=initialprobs)
mysequence[1]   <- firstnucleotide        # Store the nucleotide for the first
→position of the sequence
for (i in 2:seqlength)
{
prevnucleotide <- mysequence[i-1]      # Get the previous nucleotide in the new
→sequence
# Get the probabilities of the current nucleotide, given previous nucleotide
→"prevnucleotide":
probabilities  <- transitionmatrix[prevnucleotide,]
# Choose the nucleotide at the current position of the sequence:
nucleotide     <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
mysequence[i]  <- nucleotide              # Store the nucleotide for the current
→position of the sequence
}
return(mysequence)
}
```

The function generatemarkovseq() takes as its arguments (inputs) the transition matrix for the particular Markov model; a vector containing the values of Π_A, Π_C, Π_G, and Π_T; 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 Π_A, Π_C, Π_G, and Π_T. 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 generatemarkovseq() 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. $\Pi_A = 0.25$, $\Pi_C = 0.25$, $\Pi_G = 0.25$, and $\Pi_T = 0.25$) , we type:

```
> myinitialprobs <- c(0.25, 0.25, 0.25, 0.25)
> generatemarkovseq(mytransitionmatrix, myinitialprobs, 30)
[1] "A" "T" "C" "G" "G" "G" "G" "A" "T" "A" "T" "A" "T" "A" "G" "C" "G" "C" "T" "C
→" "C" "C" "G"
[24] "A" "C" "A" "A" "A" "T" "C"
```

As you can see, there are many "A"s after "T"s in the sequence. This is because p_{TA} 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 p_{CA} and p_{CT} have low values (0.1) in this transition matrix.

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 labelled "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 p_A, p_T, p_G and p_C 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.
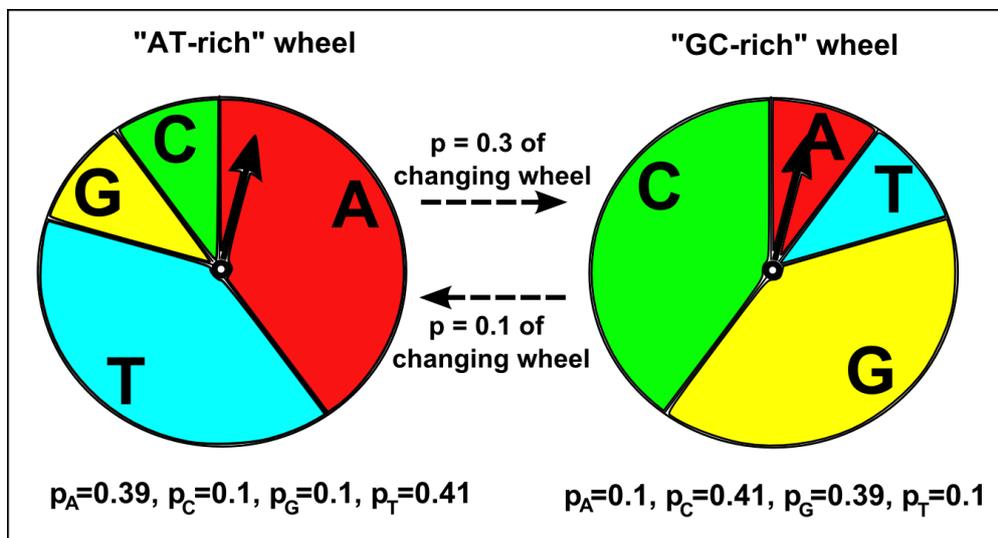
The transition matrix and emission matrix for a HMM

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:

```
> states              <- c("AT-rich", "GC-rich") # Define the names of the states
> ATrichprobs         <- c(0.7, 0.3)             # Set the probabilities of
↪switching states, where the previous state was "AT-rich"
> GCrichprobs         <- c(0.1, 0.9)             # Set the probabilities of
↪switching states, where the previous state was "GC-rich"
> thetransitionmatrix <- matrix(c(ATrichprobs, GCrichprobs), 2, 2, byrow = TRUE) #
↪Create a 2 x 2 matrix
> rownames(thetransitionmatrix) <- states
> colnames(thetransitionmatrix) <- states
> thetransitionmatrix                           # Print out the transition matrix
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, p_A=0.39, p_C=0.1, p_G=0.1, and p_T=0.41 for the AT-rich state), and the probabilities of choosing "A", "C", "G", and "T" in the GC-rich state (for example, p_A=0.1, p_C=0.41, p_G=0.39, and p_T=0.1 for the GC-rich state).

```
> nucleotides         <- c("A", "C", "G", "T")    # Define the alphabet of
↪nucleotides
> ATrichstateprobs    <- c(0.39, 0.1, 0.1, 0.41) # Set the values of the
↪probabilities, for the AT-rich state
> GCrichstateprobs    <- c(0.1, 0.41, 0.39, 0.1) # Set the values of the
↪probabilities, for the GC-rich state
> theemissionmatrix <- matrix(c(ATrichstateprobs, GCrichstateprobs), 2, 4, byrow =
↪TRUE) # Create a 2 x 4 matrix
> rownames(theemissionmatrix) <- states
> colnames(theemissionmatrix) <- nucleotides
> theemissionmatrix                             # Print out the emission matrix
   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).

Generating a DNA sequence using a HMM

The following function generatehmmseq() 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.

```
> # Function to generate a DNA sequence, given a HMM and the length of the
↪sequence to be generated.
  generatehmmseq <- function(transitionmatrix, emissionmatrix, initialprobs,
↪seqlength)
  {
     nucleotides       <- c("A", "C", "G", "T")   # Define the alphabet of
↪nucleotides
     states            <- c("AT-rich", "GC-rich") # Define the names of the states
     mysequence        <- character()             # Create a vector for storing the
↪new sequence
     mystates          <- character()             # Create a vector for storing the
↪state that each position in the new sequence
                                                  # was generated by
     # Choose the state for the first position in the sequence:
     firststate        <- sample(states, 1, rep=TRUE, prob=initialprobs)
     # Get the probabilities of the current nucleotide, given that we are in the
↪state "firststate":
     probabilities     <- emissionmatrix[firststate,]
     # Choose the nucleotide for the first position in the sequence:
     firstnucleotide <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
     mysequence[1]    <- firstnucleotide         # Store the nucleotide for the
↪first position of the sequence
     mystates[1]      <- firststate              # Store the state that the first
↪position in the sequence was generated by

     for (i in 2:seqlength)
     {
        prevstate     <- mystates[i-1]           # Get the state that the previous
↪nucleotide in the sequence was generated by
        # Get the probabilities of the current state, given that the previous
↪nucleotide was generated by state "prevstate"
        stateprobs    <- transitionmatrix[prevstate,]
        # Choose the state for the ith position in the sequence:
        state         <- sample(states, 1, rep=TRUE, prob=stateprobs)
        # Get the probabilities of the current nucleotide, given that we are in
↪the state "state":
        probabilities <- emissionmatrix[state,]
        # Choose the nucleotide for the ith position in the sequence:
        nucleotide    <- sample(nucleotides, 1, rep=TRUE, prob=probabilities)
        mysequence[i] <- nucleotide              # Store the nucleotide for the
↪current position of the sequence
        mystates[i]   <- state                   # Store the state that the current
↪position in the sequence was generated by
     }

     for (i in 1:length(mysequence))
     {
        nucleotide    <- mysequence[i]
        state         <- mystates[i]
        print(paste("Position", i, ", State", state, ", Nucleotide = ",
↪nucleotide))
     }
  }
```

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 generatehmmseq() also requires the probabilities of the choosing each of the states at the

first position (eg. $\Pi_{\text{AT-rich}}$ and $\Pi_{\text{GC-rich}}$ 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 generatehmmseq() 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. $\Pi_{\text{AT-rich}} = 0.5$, $\Pi_{\text{GC-rich}} = 0.5$), we type:

```
> theinitialprobs <- c(0.5, 0.5)
> generatehmmseq(thetransitionmatrix, theemissionmatrix, theinitialprobs, 30)
[1] "Position 1 , State AT-rich , Nucleotide =  A"
[1] "Position 2 , State AT-rich , Nucleotide =  A"
[1] "Position 3 , State AT-rich , Nucleotide =  G"
[1] "Position 4 , State AT-rich , Nucleotide =  C"
[1] "Position 5 , State AT-rich , Nucleotide =  G"
[1] "Position 6 , State AT-rich , Nucleotide =  T"
[1] "Position 7 , State GC-rich , Nucleotide =  G"
[1] "Position 8 , State GC-rich , Nucleotide =  G"
[1] "Position 9 , State GC-rich , Nucleotide =  G"
[1] "Position 10 , State GC-rich , Nucleotide =  G"
[1] "Position 11 , State GC-rich , Nucleotide =  C"
[1] "Position 12 , State GC-rich , Nucleotide =  C"
[1] "Position 13 , State GC-rich , Nucleotide =  C"
[1] "Position 14 , State GC-rich , Nucleotide =  C"
[1] "Position 15 , State GC-rich , Nucleotide =  G"
[1] "Position 16 , State GC-rich , Nucleotide =  G"
[1] "Position 17 , State GC-rich , Nucleotide =  C"
[1] "Position 18 , State GC-rich , Nucleotide =  G"
[1] "Position 19 , State GC-rich , Nucleotide =  A"
[1] "Position 20 , State GC-rich , Nucleotide =  C"
[1] "Position 21 , State GC-rich , Nucleotide =  A"
[1] "Position 22 , State AT-rich , Nucleotide =  T"
[1] "Position 23 , State GC-rich , Nucleotide =  G"
[1] "Position 24 , State GC-rich , Nucleotide =  G"
[1] "Position 25 , State GC-rich , Nucleotide =  G"
[1] "Position 26 , State GC-rich , Nucleotide =  G"
[1] "Position 27 , State GC-rich , Nucleotide =  T"
[1] "Position 28 , State GC-rich , Nucleotide =  G"
[1] "Position 29 , State GC-rich , Nucleotide =  T"
[1] "Position 30 , State GC-rich , Nucleotide =  C"
```

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 p_G and p_C 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 p_T and p_A 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.

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 following function viterbi() is a function for the Viterbi algorithm:

```r
> viterbi <- function(sequence, transitionmatrix, emissionmatrix)
  # This carries out the Viterbi algorithm.
  # Adapted from "Applied Statistics for Bioinformatics using R" by Wim P. Krijnen,
↪ page 209
  # ( cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf )
  {
     # Get the names of the states in the HMM:
     states <- rownames(theemissionmatrix)

     # Make the Viterbi matrix v:
     v <- makeViterbimat(sequence, transitionmatrix, emissionmatrix)

     # Go through each of the rows of the matrix v (where each row represents
     # a position in the DNA sequence), and find out which column has the
     # maximum value for that row (where each column represents one state of
     # the HMM):
     mostprobablestatepath <- apply(v, 1, function(x) which.max(x))

     # Print out the most probable state path:
     prevnucleotide <- sequence[1]
     prevmostprobablestate <- mostprobablestatepath[1]
     prevmostprobablestatename <- states[prevmostprobablestate]
     startpos <- 1
     for (i in 2:length(sequence))
     {
        nucleotide <- sequence[i]
        mostprobablestate <- mostprobablestatepath[i]
        mostprobablestatename <- states[mostprobablestate]
        if (mostprobablestatename != prevmostprobablestatename)
        {
           print(paste("Positions",startpos,"-",(i-1), "Most probable state = ",
↪prevmostprobablestatename))
           startpos <- i
        }
        prevnucleotide <- nucleotide
        prevmostprobablestatename <- mostprobablestatename
     }
     print(paste("Positions",startpos,"-",i, "Most probable state = ",
↪prevmostprobablestatename))
   }
```

The viterbi() function requires a second function makeViterbimat():

```r
> makeViterbimat <- function(sequence, transitionmatrix, emissionmatrix)
  # This makes the matrix v using the Viterbi algorithm.
  # Adapted from "Applied Statistics for Bioinformatics using R" by Wim P. Krijnen,
↪ page 209
  # ( cran.r-project.org/doc/contrib/Krijnen-IntroBioInfStatistics.pdf )
  {
     # Change the sequence to uppercase
     sequence <- toupper(sequence)
     # Find out how many states are in the HMM
     numstates <- dim(transitionmatrix)[1]
```

```
    # Make a matrix with as many rows as positions in the sequence, and as many
    # columns as states in the HMM
    v <- matrix(NA, nrow = length(sequence), ncol = dim(transitionmatrix)[1])
    # Set the values in the first row of matrix v (representing the first
↪position of the sequence) to 0
    v[1, ] <- 0
    # Set the value in the first row of matrix v, first column to 1
    v[1,1] <- 1
    # Fill in the matrix v:
    for (i in 2:length(sequence)) # For each position in the DNA sequence:
    {
        for (l in 1:numstates) # For each of the states of in the HMM:
        {
            # Find the probabilility, if we are in state l, of choosing the
↪nucleotide at position in the sequence
            statelprobnucleotidei <- emissionmatrix[l,sequence[i]]

            # v[(i-1),] gives the values of v for the (i-1)th row of v, ie. the (i-
↪1)th position in the sequence.
            # In v[(i-1),] there are values of v at the (i-1)th row of the sequence
↪for each possible state k.
            # v[(i-1),k] gives the value of v at the (i-1)th row of the sequence
↪for a particular state k.

            # transitionmatrix[l,] gives the values in the lth row of the
↪transition matrix, xx should not be transitionmatrix[,l]?
            # probabilities of changing from a previous state k to a current state
↪l.

            # max(v[(i-1),] * transitionmatrix[l,]) is the maximum probability for
↪the nucleotide observed
            # at the previous position in the sequence in state k, followed by a
↪transition from previous
            # state k to current state l at the current nucleotide position.

            # Set the value in matrix v for row i (nucleotide position i), column l
↪(state l) to be:
            v[i,l] <-  statelprobnucleotidei * max(v[(i-1),] * transitionmatrix[,l])
        }
    }
    return(v)
}
```

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:

```
> myseq <- c("A", "A", "G", "C", "G", "T", "G", "G", "G", "G", "C", "C", "C", "C",
↪"G", "G", "C", "G", "A", "C", "A", "T", "G", "G", "G", "G", "T", "G", "T", "C")
> 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 - 23 Most probable state =  GC-rich"
```

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 p_A, p_R, p_C... 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 p_A, p_R, p_C... 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. p_A of the "hydrophobic" state will be higher than the p_A 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.

Summary

In this practical, you will have learnt 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.

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).

Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the Matlab case studies on the Bacteriophage lambda genome (www.computational-genomics.net/case_studies//lambdaphage_demo.html) and on the olfactory receptors (www.computational-genomics.net/case_studies/olfactoryreceptors_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/).

Several of the examples and and sample code used in this practical were inspired by the examples and code in the great 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).

Thank you to Noel O'Boyle for his nice suggestion of using roulette wheels to explain multinomial models, Markov models and HMMs.

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.

Q1. 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 p_A= 0.27, p_C= 0.2084, p_G= 0.198, and p_T= 0.3236. For the GC-rich state, set p_A= 0.2462, p_C= 0.2476, p_G= 0.2985, and p_T= 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?

Q2. 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 p_A= 0.3236, p_C= 0.2084, p_G= 0.198, and p_T= 0.27. For the C-rich state, set p_A= 0.2462, p_C= 0.2985, p_G= 0.2476, and p_T= 0.2077. For the G-rich state, set p_A= 0.2462, p_C= 0.2476, p_G= 0.2985, and p_T= 0.2077. For the T-rich state, set p_A= 0.27, p_C= 0.2084, p_G= 0.198, and p_T= 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)?

Q3. Make a two-state HMM to model protein sequence evolution, with "hydrophilic" and "hydrophobic" states. For the hydrophilic state, set p_A= 0.02, p_R= 0.068, p_N= 0.068, p_D= 0.068, p_C= 0.02, p_Q= 0.068, p_E= 0.068, p_G= 0.068, p_H= 0.068, p_I= 0.012, p_L= 0.012, p_K= 0.068, p_M= 0.02, p_F= 0.02, p_P= 0.068, p_S= 0.068, p_T= 0.068, p_W= 0.068, p_Y= 0.068, and p_V= 0.012. For the hydrophobic state, set p_A= 0.114, p_R= 0.007, p_N= 0.007, p_D= 0.007, p_C= 0.114, p_Q= 0.007, p_E= 0.007, p_G= 0.025, p_H= 0.007, p_I= 0.114, p_L= 0.114, p_K= 0.007, p_M= 0.114, p_F= 0.114, p_P= 0.025, p_S= 0.026, p_T= 0.026, p_W= 0.025, p_Y= 0.026, and p_V= 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?

Answers to the End-of-chapter Exercises

DNA Sequence Statistics (1)

Q1.

What are the last twenty nucleotides of the DEN-1 Dengue virus genome sequence?

To answer this, you first need to install the "SeqinR" R package, and download the DEN-1 Dengue genome sequence from the NCBI database and save it as a file "den1.fasta" in the "My Documents" folder.

Then to find the length of the DEN-1 Dengue virus genome sequence, type in the R console:

```
> library("seqinr")
> dengue <- read.fasta(file="den1.fasta")
> dengueseq <- dengue[[1]]
> length(dengueseq)
[1] 10735
```

This tells us that the length of the sequence is 10735 nucleotides. Therefore, the last 20 nucleotides are from 10716 to 10735. You can extract the sequence of these nucleotides by typing:

```
> dengueseq[10716:10735]
[1] "c" "t" "g" "t" "t" "g" "a" "a" "t" "c" "a" "a" "c" "a" "g" "g" "t" "t" "c"
[20] "t"
```

Q2.

What is the length in nucleotides of the genome sequence for the bacterium Mycobacterium leprae strain TN (accession NC_002677)?

To answer this question, you first need to retrieve the *Mycobacterium leprae* TN genome sequence from the NCBI database. You can use this by going to the NCBI website and searching for it via the NCBI website, or alternatively by using the getncbiseq() function in R.

To get the *Mycobacterium leprae* TN genome via the NCBI website, its necessary to first go to the NCBI website (www.ncbi.nlm.nih.gov) and search for NC_002677 and download it as a fasta format file (eg. "leprae.fasta") and save it in the "My Documents" folder. You can then read the sequence into R from the file by typing:

Then in R type:

```
> leprae <- read.fasta(file="leprae.fasta")
> lepraeseq <- leprae[[1]]
```

Alternatively, to get the *Mycobacterium leprae* TN genome using the getncbiseq() function in R, you first need to copy the getncbiseq() function and paste it into R, and then you can retrieve the sequence (accession **NC**_002677) by typing in R:

```
> lepraeseq <-  getncbiseq("NC_002677")
```

Now we have the *Mycobacterium leprae* TN genome sequence stored in the vector *lepraseq* in R. We can get the length of the sequence by getting the length of the vector:

```
> length(lepraeseq)
[1] 3268203
```

Q3.

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?

Type:

```
> table(lepraeseq)
lepraeseq
     a      c      g      t
687041 938713 950202 692247
```

Q4.

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?

Find out how the GC function deals with non-A/C/T/G nucleotides, type:

```
> help("GC")
```

Type:

```
> GC(lepraeseq)
[1] 0.5779675
> GC(lepraeseq, exact=FALSE)
[1] 0.5779675
```

This gives 0.5779675 or 57.79675%. This is the GC content when non-A/C/T/G nucleotides are not taken into account.

The length of the *M. leprae* sequence is 3268203 bp, and it has 938713 Cs and 950202 Gs, and 687041 As and 692247 Ts. So to calculating the GC content when only considering As, Cs, Ts and Gs, we can also type:

```
> (938713+950202)/(938713+950202+687041+692247)
[1] 0.5779675
```

To take non-A/C/T/G nucleotides into account when calculating GC, type:

```
> GC(lepraeseq, exact=TRUE)
[1] 0.5779675
```

We get the same answer as when we ignored non-A/C/G/T nucleotides. This is actually because the *M. leprae* TN sequence does not have any non-A/C/G/T nucleotides.

However, many other genome sequences do contain non-A/C/G/T nucleotides. Note that under 'Details' in the box that appears when you type 'help('GC')', it says : "When exact is set to TRUE the G+C content is estimated with ambiguous bases taken into account. Note that this is time expensive. A first pass is made on non-ambiguous bases to estimate the probabilities of the four bases in the sequence. They are then used to weight the contributions of ambiguous bases to the G+C content."

Q5.

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?

First you need to search for a function to calculate reverse complement, eg. by typing:

```
> help.search("complement")
```

You will find that there is a function seqinr::comp that complements a nucleic acid sequence. This means it is a function in the SeqinR package.

Find out how to use this function by typing:

```
> help("comp")
```

The help says "Undefined values are returned as NA". This means that the complement of non-A/C/T/G symbols will be returned as NA.

To find the number of A, C, T, and G in the reverse complement type:

```
> complepraeseq <- comp(lepraeseq)
> table(complepraeseq)
 complepraeseq
      a        c        g        t
 692247 950202 938713 687041
```

Note that in the *M. leprae* sequence we had 687041 As, in the complement have 687041 Ts. In the *M. leprae* sequence we had 938713 Cs, in the complement have 938713 Gs. In the *M. leprae* sequence we had 950202 Gs, in the complement have 950202 Cs. In the *M. leprae* sequence we had 692247 Ts, in the complement have 692247 As.

Q6.

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

```
> count(lepraeseq, 2)
    aa     ac     ag     at     ca     cc     cg     ct     ga     gc     gg
 149718 206961 170846 159516 224666 236971 306986 170089 203397 293261 243071
    gt     ta     tc     tg     tt
 210473 109259 201520 229299 152169
```

Get count for CC is 236,971; count for CG is 306,986; count for GC is 293,261.

Q7.

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?

Type:

```
> length(lepraeseq)
[1] 3268203
```

to find the length of the *M. leprae* genome sequence. It is 3,268,203 bp. Therefore the first 1000 nucleotides will have indices 1-1000, and the last thousand nucleotides will have indices 3267204-3268203. We find the count of DNA words of length 2 by typing:

```
> count(lepraeseq[1:1000],2)
 aa ac ag at ca cc cg ct ga gc gg gt ta tc tg tt
 78 95 51 49 85 82 92 54 68 63 39 43 42 73 31 54
> count(lepraeseq[3267204:3268203],2)
 aa ac ag at ca cc cg ct ga gc gg gt ta tc tg tt
 70 85 44 55 94 81 87 50 53 75 49 51 36 72 48 49
```

To check that the subsequences that you looked at are 1000 nucleotides long, you can type:

```
> length(lepraeseq[1:1000])
[1] 1000
> length(lepraeseq[3267204:3268203])
[1] 1000
```

DNA Sequence Statistics (2)

Q1.

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)?

To do this, you first need to download the DEN-1 Dengue virus sequence from the NCBI database. To do this follow the steps in the chapter DNA Sequence Statistics (1).

Then read the sequence into R using the SeqinR package:

```
> library("seqinr")
> dengue <- read.fasta(file = "den1.fasta")
> dengueseq <- dengue[[1]]
```

Then write a function to make a sliding window plot:

```
> slidingwindowplot <- function(windowsize, inputseq)
  {
     starts <- seq(1, length(inputseq)-windowsize, by = windowsize)
     n <- length(starts)
     chunkGCs <- numeric(n)
     for (i in 1:n) {
        chunk <- inputseq[starts[i]:(starts[i]+windowsize-1)]
        chunkGC <- GC(chunk)
        chunkGCs[i] <- chunkGC
     }
     plot(starts,chunkGCs,type="b",xlab="Nucleotide start position",ylab="GC␣
→content")
  }
```

Then make a sliding window plot with a window size of 200 nucleotides:

```
> slidingwindowplot(200, dengueseq)
```
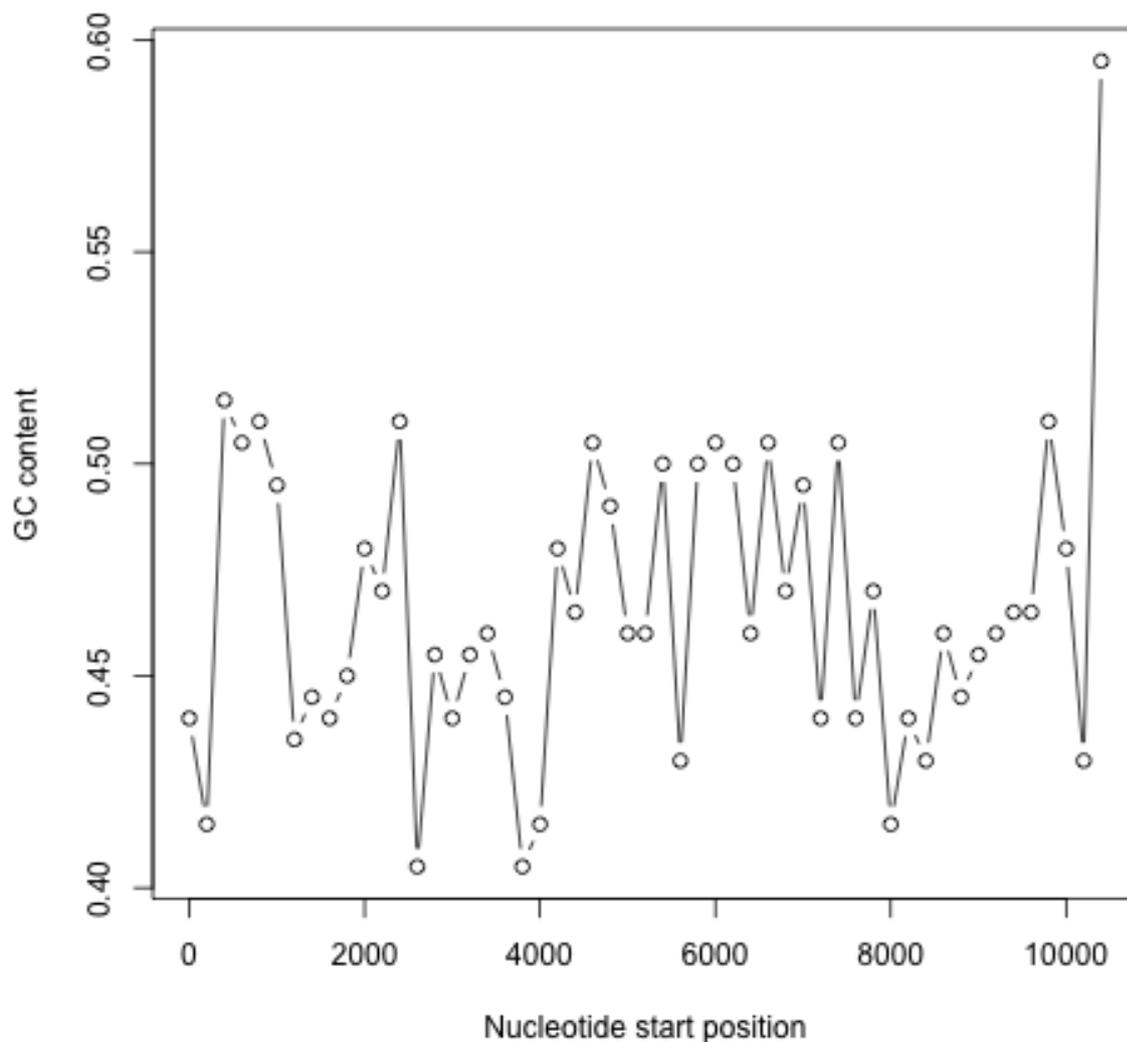

The GC content varies from about 45% to about 50% throughout the DEN-1 Dengue virus genome, with some noticeable troughs at about 2500 bases and at about 4000 bases along the sequence, where the GC content drops to about 40%. There is no strong difference between the start and end of the genome, although from around bases

4000-7000 the GC content is quite high (about 50%), and from about 2500-3500 and 7000-9000 bases the GC content is relatively low (about 43-47%).

We can also make a sliding window plot of GC content using a window size of 2000 nucleotides:

```
> slidingwindowplot(2000, dengueseq)
```
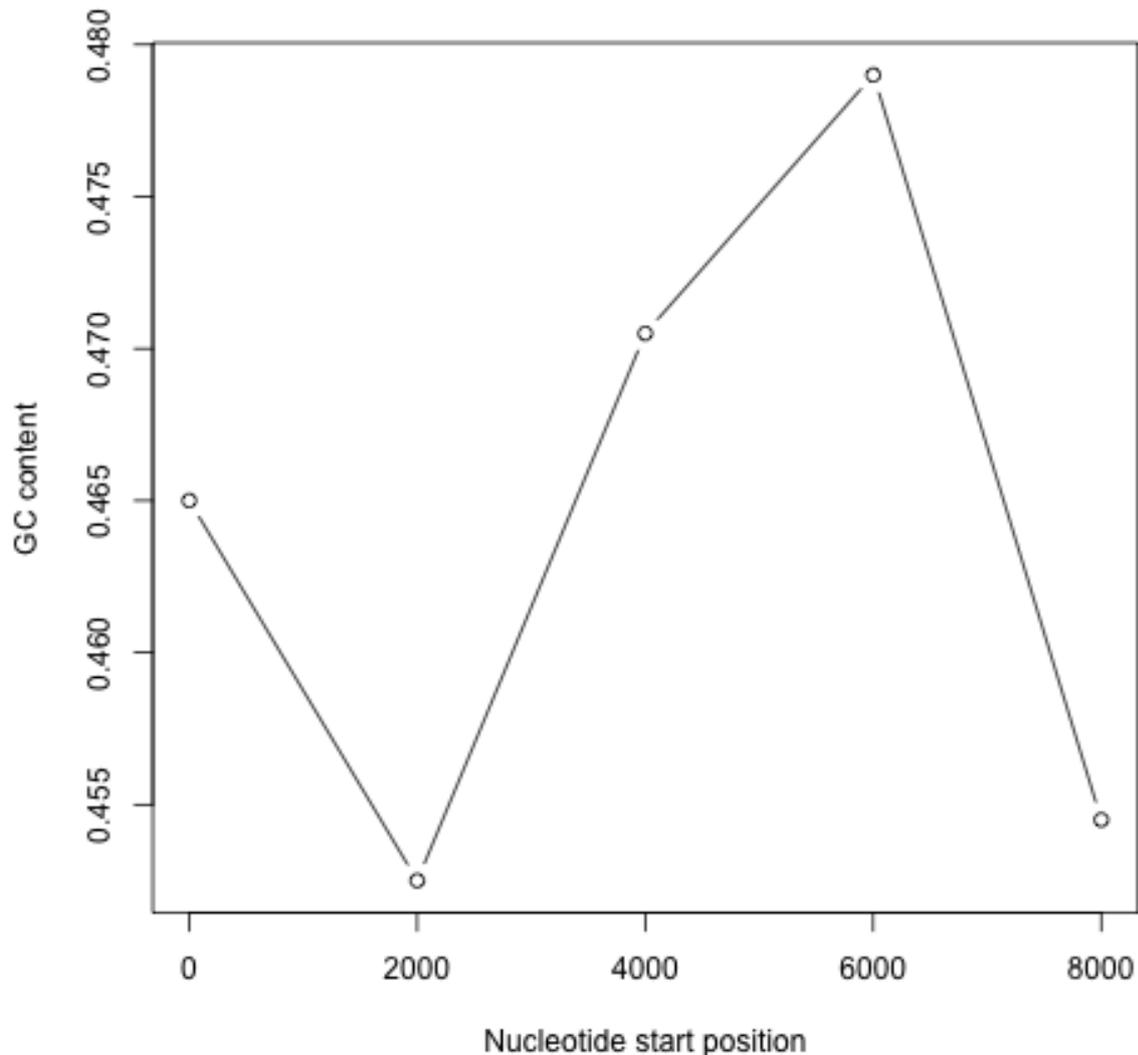

In this picture it is much more noticeable that the GC content is relatively high from around 4000-7000 bases, and lower on either side (from 2500-3500 and 7000-9000 bases).

Q2.

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)?

To do this, you first need to download the *Mycobacterium leprae* sequence from the NCBI database. To do this follow the steps in the chapter DNA Sequence Statistics (1).

Then read the sequence into R using the SeqinR package:

```
> leprae <- read.fasta(file = "leprae.fasta")
> lepraeseq <- leprae[[1]]
```

Then make a sliding window plot with a window size of 20000 nucleotides:

```
> slidingwindowplot(20000, lepraeseq)
```
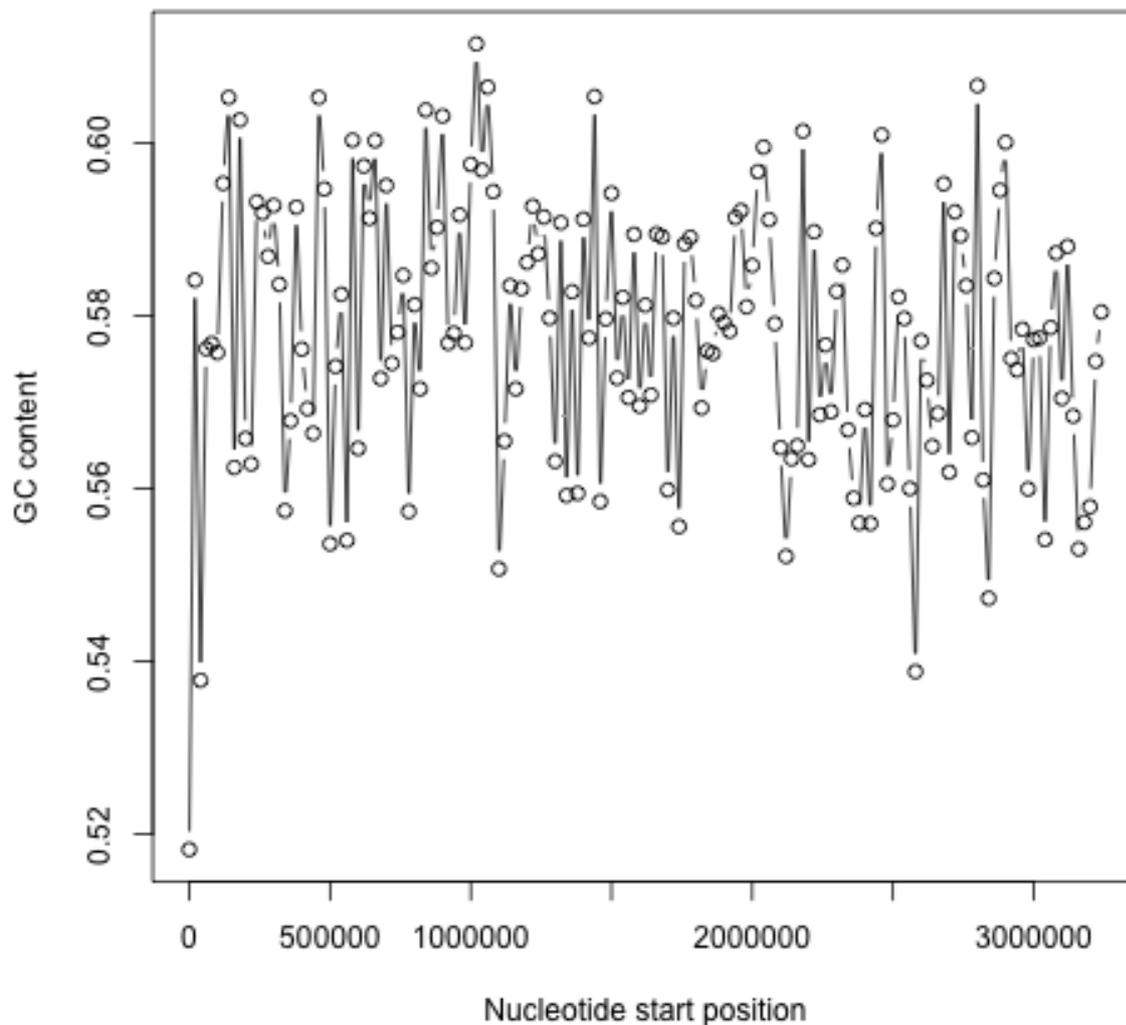

We see the highest peak in GC content at about 1 Mb into the *M. leprae* genome. We also see troughs in GC content at about 1.1 Mb, and at about 2.6 Mb.

With a window size of 200 nucleotides, the plot is very messy, and we cannot see the peaks and troughs in GC content so easily:

```
> slidingwindowplot(200, lepraeseq)
```
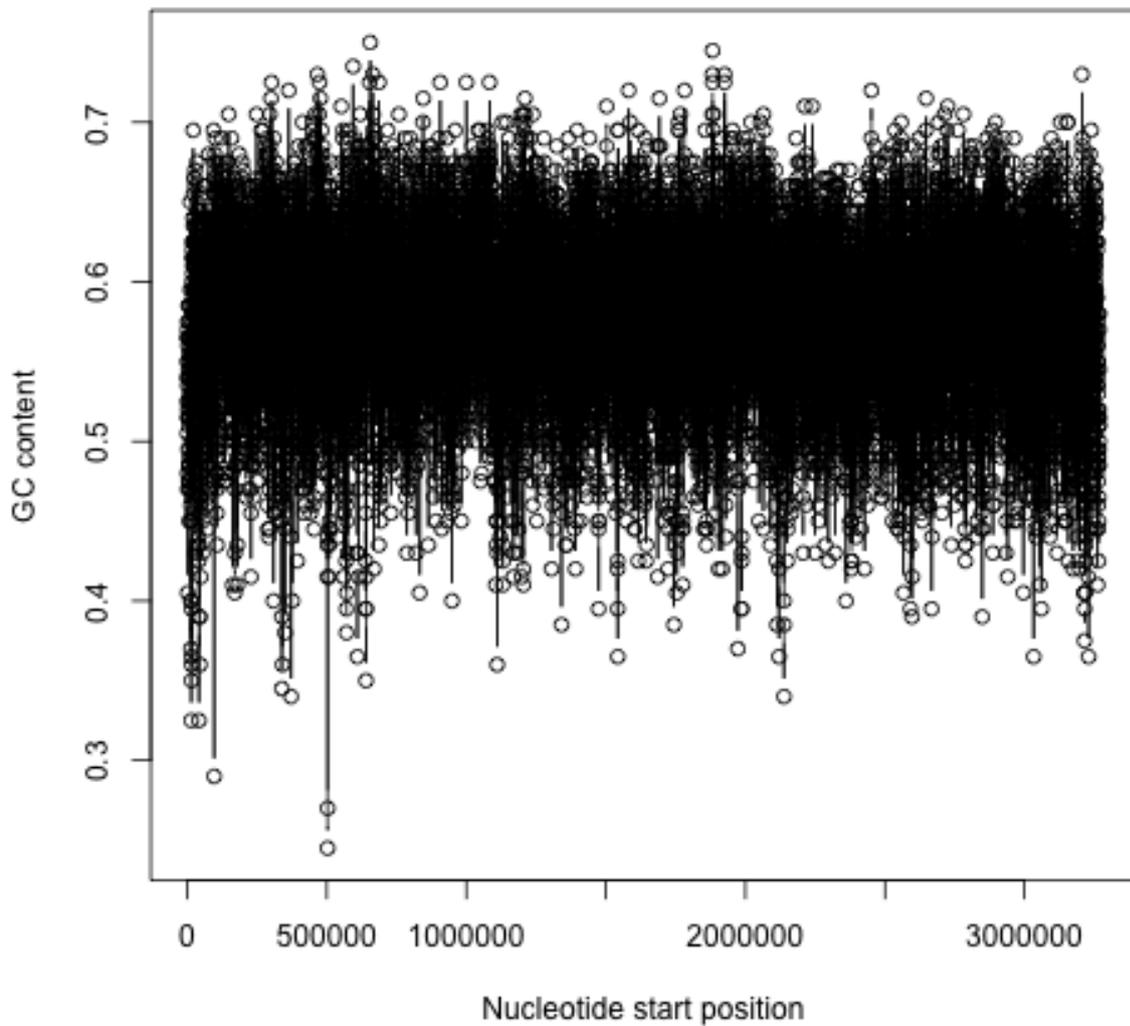
With a window size of 200,000 nucleotides, the plot is very smooth, and we cannot see the peaks and troughs in GC content very easily:

```
> slidingwindowplot(200000, lepraeseq)
```

Q3.

Write a function to calculate the AT content of a DNA sequence (ie. the fraction of the nucleotides in the sequence that are As or Ts). What is the AT content of the Mycobacterium leprae TN genome?

Here is a function to calculate the AT content of a genome sequence:

```
> AT <- function(inputseq)
  {
     mytable <- count(inputseq, 1) # make a table with the count of As, Cs, Ts,␣
→and Gs
     mylength <- length(inputseq) # find the length of the whole sequence
     myAs <- mytable[[1]] # number of As in the sequence
     myTs <- mytable[[4]] # number of Ts in the sequence
     myAT <- (myAs + myTs)/mylength
     return(myAT)
  }
```
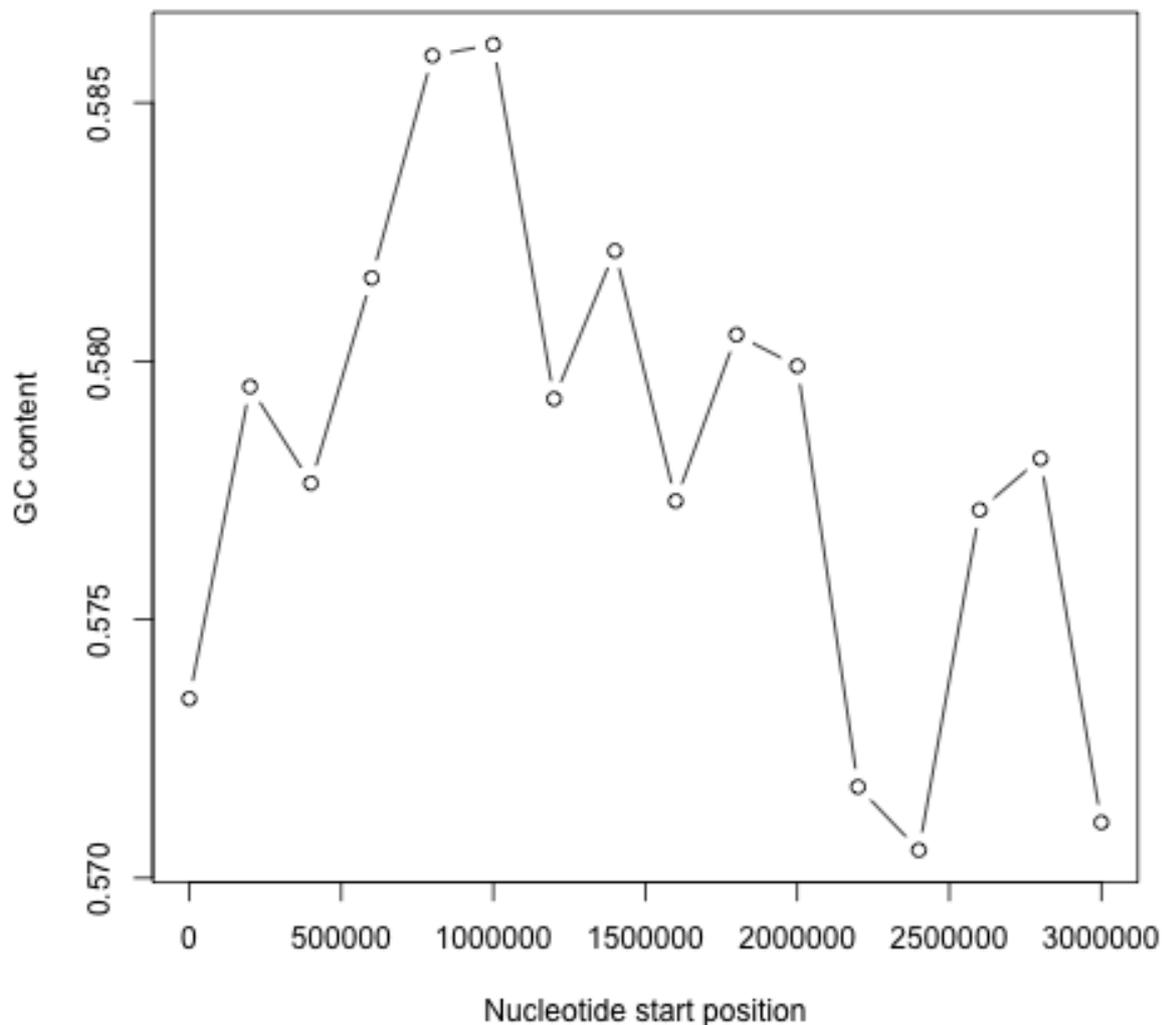
We can then use the function to calculate the AT content of the *M. leprae* genome:

```
> AT(lepraeseq)
[1] 0.4220325
```

You should notice that the AT content is (1 minus GC content), ie. (AT content + GC content = 1):

```
> GC(lepraeseq)
[1] 0.5779675
> 0.4220325 + 0.5779675
[1] 1
```

Q4.

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 windowsize 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?

We can write a function to write a sliding window plot of AT content:

```
> slidingwindowplotAT <- function(windowsize, inputseq)
  {
     starts <- seq(1, length(inputseq)-windowsize, by = windowsize)
     n <- length(starts)
     chunkATs <- numeric(n)
     for (i in 1:n) {
        chunk <- inputseq[starts[i]:(starts[i]+windowsize-1)]
        chunkAT <- AT(chunk)
        chunkATs[i] <- chunkAT
     }
     plot(starts,chunkATs,type="b",xlab="Nucleotide start position",ylab="AT
→content")
  }
```

We can then use this function to make a sliding window plot with a window size of 20000 nucleotides:

```
> slidingwindowplotAT(20000, lepraeseq)
```
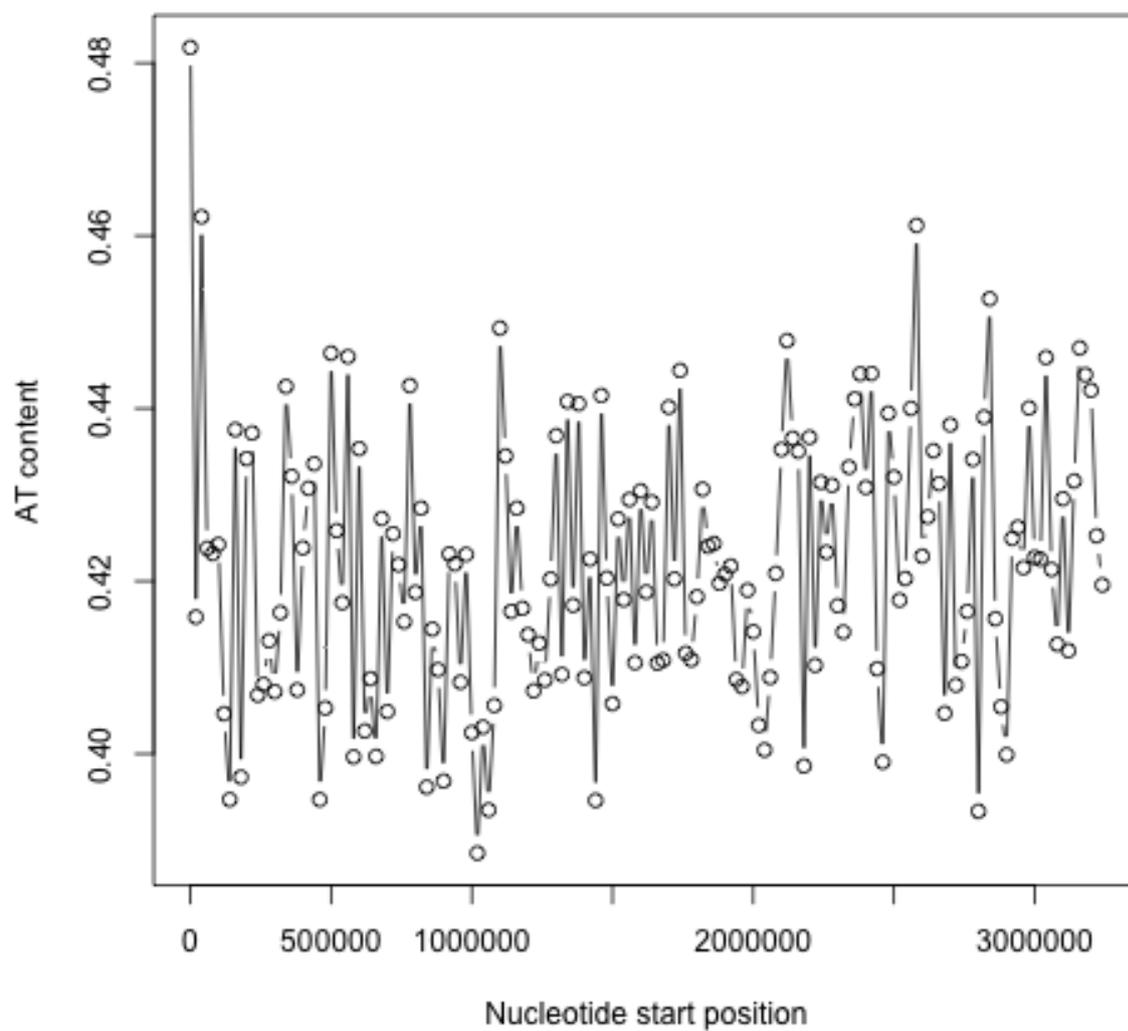
This is the mirror image of the plot of GC content (because AT equals 1 minus GC):

```
> slidingwindowplot(20000, lepraeseq)
```
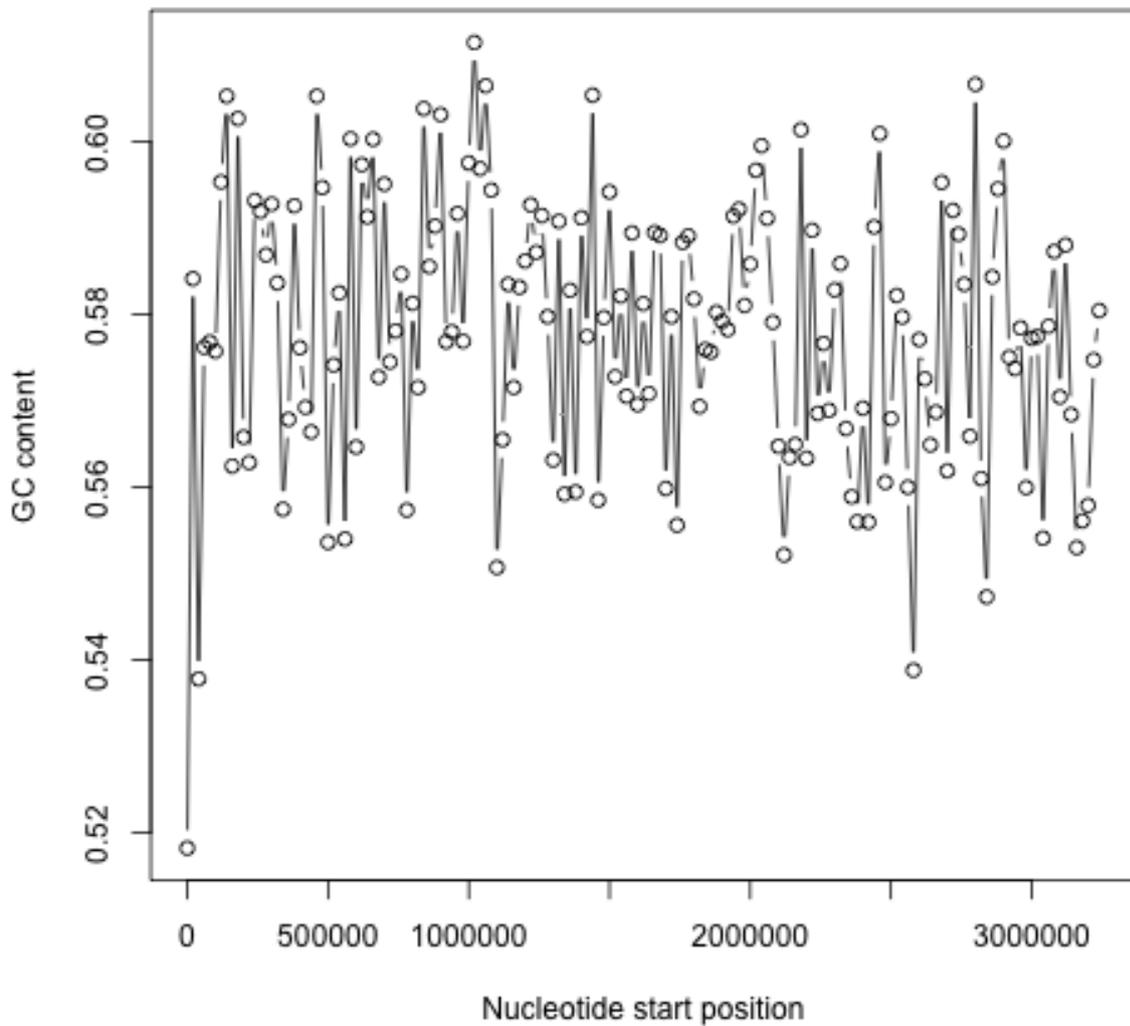
Q5.

Is the 3-nucleotide word GAC GC over-represented or under-represented in the Mycobacterium leprae TN genome sequence?

We can get the number of counts of each of the 3-nucleotide words by typing:

```
> count(lepraeseq, 3)
   aaa    aac    aag    aat    aca    acc    acg    act    aga    agc    agg    agt    ata ␣
→atc    atg
 32093 48714 36319 32592 44777 67449 57326 37409 31957 62473 38946 37470 25030␣
→57245 44268
   att    caa    cac    cag    cat    cca    ccc    ccg    cct    cga    cgc    cgg    cgt ␣
→cta    ctc
 32973 52381 64102 64345 43838 64869 46037 87560 38504 78120 82057 89358 57451␣
→29004 39954
   ctg    ctt    gaa    gac    gag    gat    gca    gcc    gcg    gct    gga    ggc    ggg ␣
→ggt    gta
 64730 36401 43486 61174 40728 58009 66775 80319 83415 62752 44002 81461 47651␣
→69957 33139
```

```
   gtc   gtg   gtt   taa   tac   tag   tat   tca   tcc   tcg   tct   tga   tgc ␣
→tgg   tgt
 60958 65955 50421 21758 32971 29454 25076 48245 43166 78685 31424 49318 67270␣
→67116 45595
   tta   ttc   ttg   ttt
 22086 43363 54346 32374
```

There are 61,174 GACs in the sequence.

The total number of 3-nucleotide words is calculated by typing:

```
> sum(count(lepraeseq,3))
[1] 3268201
```

Therefore, the observed frequency of GAC is 61174/3268201 = 0.01871794.

To calculate the expected frequency of GAC, first we need to get the number of counts of 1-nucleotide words by typing:

```
> count(lepraeseq, 1)
     a      c      g      t
 687041 938713 950202 692247
```

The sequence length is 3268203 bp. The frequency of G is 950202/3268203 = 0.2907414. The frequency of A is 687041/3268203 = 0.2102198. The frequency of C is 938713/3268203 = 0.2872260. The expected frequency of GAC is therefore 0.2907414*0.2102198*0.2872260 = 0.01755514.

The value of Rho is therefore the observed frequency/expected frequency = 0.01871794/0.01755514 = 1.066237. That, is there are about 1.1 times as many GACs as expected. This means that GAC is slightly over-represented in this sequence. The difference from 1 is so little however that it might not be statistically significant.

We can search for a function to calculate rho by typing:

```
> help.search("rho")
 base::getHook                          Functions to Get and Set Hooks for Load,␣
→Attach, Detach and Unload
 seqinr::rho                            Statistical over- and under-␣
→representation of dinucleotides in a sequence
 stats::cor.test                        Test for Association/Correlation Between␣
→Paired Samples
 survival::pbc                          Mayo Clinic Primary Biliary Cirrhosis Dat
```

There is a function rho in the SeqinR package. For example, we can use it to calculate Rho for words of length 3 in the *M. leprae* genome by typing:

```
> rho(lepraeseq, wordsize=3)
      aaa       aac       aag       aat       aca       acc       acg       act ␣
→    aga
 1.0570138 1.1742862 0.8649101 1.0653761 1.0793820 1.1899960 0.9991680 0.8949893␣
→0.7610323
      agc       agg       agt       ata       atc       atg       att       caa ␣
→    cac
 1.0888781 0.6706048 0.8856096 0.8181874 1.3695545 1.0462815 1.0697245 1.2626819␣
→1.1309452
      cag       cat       cca       ccc       ccg       cct       cga       cgc ␣
→    cgg
 1.1215062 1.0487995 1.1444773 0.5944657 1.1169725 0.6742135 1.3615987 1.0467726␣
→1.1261261
      cgt       cta       ctc       ctg       ctt       gaa       gac       gag ␣
→    gat
 0.9938162 0.6939044 0.6996033 1.1197319 0.8643241 1.0355868 1.0662370 0.7012887␣
→1.3710523
      gca       gcc       gcg       gct       gga       ggc       ggg       ggt ␣
→    gta
```

```
  1.1638601 1.0246015 1.0512300 1.0855155 0.7576632 1.0266049 0.5932565 1.1955191␣
→0.7832457
       gtc       gtg       gtt       taa       tac       tag       tat       tca ␣
→    tcc
  1.0544820 1.1271276 1.1827465 0.7112314 0.7888126 0.6961501 0.8135266 1.1542345␣
→0.7558461
       tcg       tct       tga       tgc       tgg       tgt       tta       ttc ␣
→    ttg
  1.3611325 0.7461477 1.1656391 1.1636701 1.1469683 1.0695410 0.7165237 1.0296334␣
→1.2748168
       ttt
  1.0423929
```

The Rho value for GAC is given as 1.0662370, in agreement with our calculation above.

Sequence Databases

Q1.

What information about the rabies virus sequence (NCBI accession NC_001542) can you obtain from its annotations in the NCBI Sequence Database?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and type the rabies virus genome sequence accession (NC_001542) in the search box, and press 'Search'.

On the search results page, you should see '1' beside the word 'Nucleotide', meaning that there was one hit to a sequence record in the NCBI Nucleotide database, which contains DNA and RNA sequences. If you click on the word 'Nucleotide', it will bring you to the sequence record, which should be the NCBI sequence record for the rabies virus' genome (ie. for accession NC_001542):

Rabies virus, complete genome

NCBI Reference Sequence: NC_001542.1

FASTA   Graphics

Go to: ⌄

```
LOCUS       NC_001542              11932 bp ss-RNA     linear   VRL 08-DEC-2008
DEFINITION  Rabies virus, complete genome.
ACCESSION   NC_001542
VERSION     NC_001542.1  GI:9627197
DBLINK      Project: 15144
KEYWORDS    .
SOURCE      Rabies virus
  ORGANISM  Rabies virus
            Viruses; ssRNA negative-strand viruses; Mononegavirales;
            Rhabdoviridae; Lyssavirus.
REFERENCE   1  (bases 5388 to 11932)
  AUTHORS   Tordo,N., Poch,O., Ermine,A., Keith,G. and Rougeon,F.
  TITLE     Completion of the rabies virus genome sequence determination:
            highly conserved domains among the L (polymerase) proteins of
            unsegmented negative-strand RNA viruses
  JOURNAL   Virology 165 (2), 565-576 (1988)
   PUBMED   3407152
REFERENCE   2  (bases 1 to 5500)
  AUTHORS   Tordo,N., Poch,O., Ermine,A., Keith,G. and Rougeon,F.
  TITLE     Walking along the rabies genome: is the large G-L intergenic region
            a remnant gene?
  JOURNAL   Proc. Natl. Acad. Sci. U.S.A. 83 (11), 3914-3918 (1986)
   PUBMED   3459163
```

On the webpage (above), you can see the DEFINITION, ORGANISM and REFERENCE fields of the NCBI record:

DEFINITION: Rabies virus, complete genome.

ORGANISM: Rabies virus

REFERENCE: There are several papers (the first is): AUTHORS: Tordo,N., Poch,O., Ermine,A., Keith,G. and Rougeon,F.

TITLE: Completion of the rabies virus genome sequence determination: highly conserved domains among the L (polymerase) proteins of unsegmented negative-strand RNA viruses

JOURNAL: Virology 165 (2), 565-576 (1988)

There are also some other references, for papers published about the rabies virus genome sequence.

An alternative way of retrieving the annotations for the rabies virus sequence is to use the SeqinR R package. As the rabies virus is a virus, its genome sequence should be in the "refseqViruses" ACNUC sub-database. Therefore, we can perform the following query to retrieve the annotations for the rabies virus genome sequence (accession NC_001542):

```
> library("seqinr")                              # load the SeqinR R package
> choosebank("refseqViruses")                    # select the ACNUC sub-
↪database to be searched
> query("rabies", "AC=NC_001542")                # specify the query
> annots <- getAnnot(rabies$req[[1]])            # retrieve the annotations
> annots[1:20]                                   # print out the first 20 lines␣
↪of the annotations
 [1] "LOCUS       NC_001542               11932 bp ss-RNA     linear   VRL 08-DEC-
↪2008"
 [2] "DEFINITION  Rabies virus, complete genome."
 [3] "ACCESSION   NC_001542"
 [4] "VERSION     NC_001542.1  GI:9627197"
 [5] "DBLINK      Project: 15144"
 [6] "KEYWORDS    ."
 [7] "SOURCE      Rabies virus"
 [8] "  ORGANISM  Rabies virus"
 [9] "            Viruses; ssRNA negative-strand viruses; Mononegavirales;"
[10] "            Rhabdoviridae; Lyssavirus."
[11] "REFERENCE   1  (bases 5388 to 11932)"
[12] "  AUTHORS   Tordo,N., Poch,O., Ermine,A., Keith,G. and Rougeon,F."
[13] "  TITLE     Completion of the rabies virus genome sequence determination:"
[14] "            highly conserved domains among the L (polymerase) proteins of"
[15] "            unsegmented negative-strand RNA viruses"
[16] "  JOURNAL   Virology 165 (2), 565-576 (1988)"
[17] "   PUBMED   3407152"
[18] "REFERENCE   2  (bases 1 to 5500)"
[19] "  AUTHORS   Tordo,N., Poch,O., Ermine,A., Keith,G. and Rougeon,F."
[20] "  TITLE     Walking along the rabies genome: is the large G-L intergenic␣
↪region"
> closebank()
```

Q2.

How many nucleotide sequences are there from the bacterium Chlamydia trachomatis in the NCBI Sequence Database?

To answer this, you need to go to www.ncbi.nlm.nih.gov and select "Nucleotide" from the drop-down list at the top of the webpage, as you want to search for nucleotide (DNA or RNA) sequences.

Then in the search box, type "Chlamydia trachomatis"[ORGN] and press 'Search':

Here [ORGN] specifies the organism you are interested in, that is, the species name in Latin.

The results page should give you a list of the hits to sequence records in the NCBI Nucleotide database:

It will say "Found 35577 nucleotide sequences. Nucleotide (35429) GSS (148)". This means that 35,577 sequences were found, of which 35429 are DNA or RNA sequences, and 148 are DNA sequences from the Genome Sequence Surveys (GSS), that is, from genome sequencing projects [as of 15-Jun-2011]. Note that there are new sequences being added to the database continuously, so if you check this again in a couple of months, you will probably find a higher number of sequences (eg. 36,000 sequences).

Note: if you just go to the www.ncbi.nlm.nih.gov database, and search for "Chlamydia trachomatis"[ORGN] (without choosing "Nucleotide" from the drop-down list), you will see 35429 hits to the Nucleotide database and 148 to the GSS (Genome Sequence Survey) database:

Note also that if you search for "Chlamydia trachomatis", without using [ORGN] to specify the organism, you will get 56032 hits to the Nucleotide database and 149 to the GSS database, but some of these might not be *Chlamydia trachomatis* sequences - some could be sequences from other species for which the NCBI sequence record contains the phrase "Chlamydia trachomatis" somewhere.

An alternative way to search for nucleotide sequences from the bacterium *Chlamydia trachomatis* is to use the SeqinR package. We want to find nucleotide sequences, so the correct ACNUC sub-database to search is the "genbank" sub-database. Thus, we can carry out our search by typing:

```
> library("seqinr")                                # load the SeqinR R package
> choosebank("genbank")                            # select the ACNUC sub-
↪database to be searched
> query("Ctrachomatis", "SP=Chlamydia trachomatis") # specify the query
> Ctrachomatis$nelem                               # print out the number of␣
↪matching sequences
  [1] 35471
> closebank()
```

We find 35,471 nucleotide sequences from *Chlamydia trachomatis*. We do not get exactly the same number of sequences as we got when we searched via the NCBI website (35,577 sequences), but the numbers are very close. The likely reasons for the differences could be that the ACNUC "genbank" sub-database excludes some sequences from whole genome sequencing projects from the NCBI Nucleotide database, and in addition, the ACNUC databases are updated very regularly, but may be missing a few sequences that were added to the NCBI database in the last day or two.

Q3.

How many nucleotide sequences are there from the bacterium Chlamydia trachomatis in the RefSeq part of the NCBI Sequence Database?

To answer this, you need to go to www.ncbi.nlm.nih.gov and select "Nucleotide" from the drop-down list at the top of the webpage, as you want to search for nucleotide sequences.

Then in the search box, type "Chlamydia trachomatis"[ORGN] AND srcdb_refseq[PROP] and press 'Search':
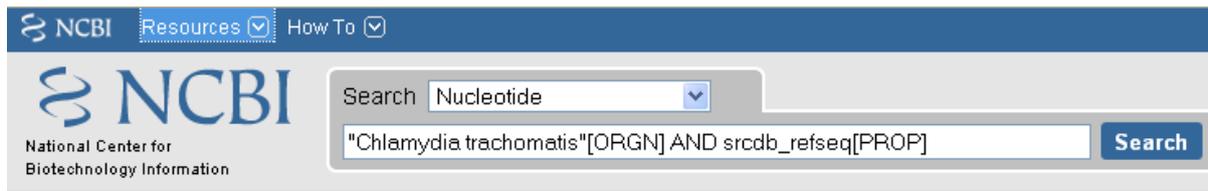
Here [ORGN] specifies the organism, and [PROP] specifies a property of the sequences (in this case that they belong to the RefSeq subsection of the NCBI database).

At the top of the results page, it should say "Results: 1 to 20 of 29 sequences", so there were 29 matching sequences [as of 15-Jun-2011]. As for Q2, if you try this again in a couple of months, the number will probably be higher, due to extra sequences added to the database.

Note that the sequences in Q2 are all *Chlamydia trachomatis* DNA and RNA sequences in the NCBI database. The sequences in Q3 gives the *Chlamydia trachomatis* DNA and RNA sequences in the RefSeq part of the NCBI database, which is a subsection of the database for high-quality manually-curated data.

The number of sequences in RefSeq is much fewer than the total number of *C. trachomatis* sequences, partly because low quality sequences are never added to RefSeq, but also because RefSeq curators have probably not had time to add all high-quality sequences to RefSeq (this is a time-consuming process, as the curators add additional information to the NCBI Sequence records in RefSeq, such as references to papers that discuss a particular sequence).

An alternative way to search for nucleotide sequences from the bacterium *Chlamydia trachomatis* in RefSeq use the SeqinR package. We want to find RefSeq sequences, so the correct ACNUC sub-database to search is the "refseq" sub-database. Thus, we can carry out our search by typing:

```
> library("seqinr")                             # load the SeqinR R package
> choosebank("refseq")                          # select the ACNUC sub-
↪database to be searched
> query("Ctrachomatis2", "SP=Chlamydia trachomatis") # specify the query
> Ctrachomatis2$nelem                           # print out the number of␣
↪matching sequences
  [1] 1
> closebank()
```

We find 1 RefSeq sequence from *Chlamydia trachomatis*. We do not get exactly the same number of sequences as we got when we searched via the NCBI website (29 sequences). This is because the 29 sequences found via the NCBI website include whole genome sequences, but the whole genome sequences from bacteria are stored in the ACNUC "bacterial" sub-database, and so are not in the ACNUC "refseq" sub-database.

Q4.

How many nucleotide sequences were submitted to NCBI by Matthew Berriman?

To answer this, you need to go to www.ncbi.nlm.nih.gov, and select "Nucleotide" from the drop-down list, as you want to search for nucleotide sequences.

Then in the search box, type "Berriman M"[AU] and press 'Search'.

Here [AU] specifies the name of the person who either submitted the sequence to the NCBI database, or wrote a paper describing the sequence.

The results page should look like this:

On the top of the results page, it says [as of 15-Jun-2011]: "Found 487270 nucleotide sequences. Nucleotide (277546) EST (121075) GSS (88649)". This means that 487270 DNA/RNA sequences were either submitted to the NCBI database by someone called M. Berriman, or were described in a paper by someone called M. Berriman. Of these, 277546 were DNA/RNA sequences, 121075 were EST sequences (part of mRNAs), and 88649 were DNA sequences from genome sequencing projects (GSS or Genome Sequence Survey sequences).

Note that unfortunately the NCBI website does not allow us to search for "Berriman Matthew"[AU] so we cannot be sure that all of these sequences were submitted by Matthew Berriman.

Note also that the search above will find sequences that were either submitted to the NCBI database by M. Berriman, or described in a paper on which M. Berriman was an author. Therefore, not all of the sequences found were necessarily submitted by M. Berriman.

An alternative way to search for nucleotide sequences submitted by M. Berriman is to use the SeqinR package. We want to find nucleotide sequences, so the appropriate ACNUC sub-database to search is "genbank". Therefore, we type:

```
> library("seqinr")                       # load the SeqinR R package
> choosebank("genbank")                   # select the ACNUC sub-database to be searched
> query("mberriman", "AU=Berriman")       # specify the query
> mberriman$nelem                         # print out the number of matching sequences
 [1] 169701
> closebank()
```

We find 169,701 matching sequences. This is less than the number found by searching via the NCBI website (487,270 sequences). The difference is probably due to the fact that the "genbank" ACNUC sub-database excludes some sequences from the NCBI Nucleotide database (eg. short sequences from genome sequencing projects).

Note that the "AU=Berriman" query will find sequences submitted or published by someone called Berriman. We are not able to specify the initial of the first name of this person using the "query()" command, so we cannot specify that the person is called "M. Berriman".

Q5.

How many nucleotide sequences from the nematode worms are there in the RefSeq Database?

To answer this, you need to go to www.ncbi.nlm.nih.gov and select "Nucleotide" from the drop-down list, as you want to search for nucleotide sequences.

Then in the search box, type Nematoda[ORGN] AND srcdb_refseq[PROP] and press 'Search'.

Here [ORGN] specifies the group of species that you want to search for sequences from. In Q3, [ORGN] was used to specify the name of one organism (*Chlamydia trachomatis*). However, you can also use [ORGN] to specify the name of a group of organisms, for example, Fungi[ORGN] would search for fungal sequences or Mammalia[ORGN] would search for mammalian sequences. The name of the group of species that you want to search for must be given in Latin, so to search for sequences from nematode worms we use the Latin name Nematoda.

The search page should say at the top 'Results: 1 to 20 of 145355' [as of 15-Jun-2011]. This means that 145,355 DNA or RNA sequences were found from nematode worm species in the RefSeq database. These sequences are probably from a wide range of nematode worm species, including the model nematode worm *Caenorhabditis elegans*, as well as parasitic nematode species.

An alternative way to search for RefSeq nucleotide sequences from nematode worms is to use the SeqinR package. We want to find nucleotide sequences that are in RefSeq, so the appropriate ACNUC sub-database to search is "refseq". Therefore, we type:

```
> library("seqinr")                    # load the SeqinR R package
> choosebank("refseq")                 # select the ACNUC sub-database to be searched
> query("nematodes", "SP=Nematoda")    # specify the query
> nematodes$nelem                      # print out the number of matching sequences
 [1] 55241
> closebank()
```

That is, using SeqinR, we find 55,241 DNA or RNA sequences from nematode worms in the RefSeq database. This is less than the number of sequences found by searching via the NCBI website (145,355 sequences). This is because the "refseq" ACNUC sub-database does not contain all of the sequences in the NCBI RefSeq database, for various reasons, for example, some of the sequences in the NCBI RefSeq database (eg. whole genome sequences) are in other ACNUC sub-databases.

Q6.

How many nucleotide sequences for collagen genes from nematode worms are there in the NCBI Database?

To answer this, you need to go to www.ncbi.nlm.nih.gov and select "Nucleotide" from the drop-down list, as you want to search for nucleotide sequences.

Then in the search box, type Nematoda[ORGN] AND collagen.

Here [ORGN] specifies that we want sequences from nematode worms. The phrase "AND collagen" means that the word collagen must appear somewhere in the NCBI entries for those sequences, for example, in the sequence name, or in a description of the sequence, or in the title of a paper describing the sequence, etc.

On the results page, you should see 'Found 8437 nucleotide sequences. Nucleotide (1642) EST (6795)' [as of 15-Jun-2011]. This means that 8437 DNA or RNA sequences for collagen genes from nematode worms were found, of which 6795 are EST sequences (parts of mRNAs). Note that these 8437 nucleotide sequences may not all necessarily be for collagen genes, as some of the NCBI records found may be for other genes but contain the word "collagen" somewhere in the NCBI record (for example, in the title of a cited paper). However, a good number of them are probably collagen sequences from nematodes.

An alternative way to search for collagen nucleotide sequences from nematode worms is to use the SeqinR package. We want to find nucleotide sequences, so the appropriate ACNUC sub-database to search is "genbank". To search for collagen genes, we can specify "collagen" as a keyword by using "K=collagen" in our query. Therefore, we type:

```
> library("seqinr")                                  # load the SeqinR R package
> choosebank("genbank")                              # select the ACNUC sub-database
↪to be searched
> query("collagen", "SP=Nematoda AND K=collagen")    # specify the query
> collagen$nelem                                     # print out the number of
↪matching sequences
 [1] 60
> closebank()
```

That is, using SeqinR, we find 60 DNA or RNA sequences with the keyword "collagen" from nematode worms. This is less than the number of sequences found by searching via the NCBI website (8437 sequences). This is probably partly because the ACNUC "genbank" sub-database excludes some sequences that are in the NCBI Nucleotide database (eg. short sequences from genome sequencing projects), but also partly because the method used to assign keywords to sequences in ACNUC is quite conservative and relatively few sequences seem to be assigned the keyword "collagen". However, presumably most of the sequences tagged with the keyword "collagen" are collagen genes (while the search via the NCBI website may have picked up many non-collagen genes, as explained above).

Q7.

How many mRNA sequences for collagen genes from nematode worms are there in the NCBI Database?

To answer this, you need to go to www.ncbi.nlm.nih.gov, and select "Nucleotide" from the drop-down sequences, as you want to search for nucleotide sequences (nucleotide sequences include DNA sequences and RNA sequences, such as mRNAs).

Then in the search box, type Nematoda[ORGN] AND collagen AND "biomol mRNA"[PROP].

Here [ORGN] specifies the name of the group of species, collagen specifies that we want to find NCBI entries that include the word collagen, and [PROP] specifies a property of those sequences (that they are mRNAs, in this case).

The search page should say 'Found 7751 nucleotide sequences. Nucleotide (956) EST (6795)' [as of 15-Jun-2011]. This means that 7751 mRNA sequences from nematodes were found that contain the word 'collagen' in the NCBI record. Of the 7751, 6795 are EST sequences (parts of mRNAs).

Note that in Q6 we found 8437 nucleotide (DNA or RNA) sequences from nematode worms. In this question, we found out that only 7751 of those sequences are mRNA sequences. This means that the other (8437-7751=) 686 sequences must be DNA sequences, or other types of RNA sequences (not mRNAs) such as tRNAs or rRNAs.

An alternative way to search for collagen mRNA sequences from nematode worms is to use the SeqinR package. mRNA sequences are nucleotide sequences, so the appropriate ACNUC sub-database to search is "genbank". To search for mRNAs, we can specify "M=mRNA" in our query. Therefore, we type:

```
> library("seqinr")                                           # load the SeqinR R
↪package
> choosebank("genbank")                                       # select the ACNUC
↪sub-database to be searched
> query("collagen2", "SP=Nematoda AND K=collagen AND M=mRNA")  # specify the query
> collagen2$nelem                                             # print out the
↪number of matching sequences
 [1] 14
> closebank()
```

We find 14 nematode mRNA sequences labelled with the keyword "collagen". Again, we find less sequences than found when searching via the NCBI website (7751 sequences), but as in Q6, the search using the keyword "collagen" in the SeqinR package may be more likely to pick up true collagen sequences (rather than other sequences that just happen to contain the word "collagen" somewhere in their NCBI entries).

Q8.

How many protein sequences for collagen proteins from nematode worms are there in the NCBI database?

To answer this, you need to go to www.ncbi.nlm.nih.gov, and select "Protein" from the drop-down list, as you want to search for protein sequences.

Then type in the search box: Nematoda[ORGN] AND collagen and press 'Search':

On the results page, you should see '1 to 20 of 1982'. This means that 1982 protein sequences from nematode worms were found that include the word collagen in the NCBI sequence entries [as of 15-Jun-2011].

As far as I know, there is not an ACNUC sub-database that contains all the protein sequences from the NCBI Protein database, and therefore it is not currently possible to carry out the same query using SeqinR.

Q9.

What is the accession number for the Trypanosoma cruzi genome in NCBI?

There are two ways that you can answer this.

The first method is to go to www.ncbi.nlm.nih.gov and select "Genome" from the drop-down list, as you want to search for genome sequences.

Then type in the search box: "Trypanosoma cruzi"[ORGN] and press 'Search':

This will search the NCBI Genome database, which contains fully sequenced genome sequences.

The results page says 'All:1', and lists just one NCBI record, the genome sequence for *Trypanosoma cruzi* strain CL Brener, which has accession NZ_AAHK00000000:

The second method of answering the question is to go directly to the NCBI Genomes webpage.

Click on the 'Eukaryota' link at the middle the page, as *Trypanosoma cruzi* is a eukaryotic species.

This will give you a complete list of all the eukaryotic genomes that have been fully sequenced.

Go to the 'Edit' menu of your web browser, and choose 'Find', and search for 'Trypanosoma cruzi':

11756	Trypanosoma brucei TREU927	Protists	Kinetoplasts	5691	37.798	11	Assembly	WGS		08/09/2003	Trypanosoma brucei consortium [more]
40697	Trypanosoma brucei gambiense DAL972 Dal 927 clone 1 (MHOM/CI/86/DAL972)	Protists	Kinetoplasts	679716	22.14	11	Complete	WGS		10/14/2009	Wellcome Trust Sanger Institute [more]
12958	Trypanosoma congolense IL3000	Protists	Kinetoplasts	5692			In Progress	WGS			Wellcome Trust Sanger Institute
11755	Trypanosoma cruzi CL Brener	Protists	Kinetoplasts	5693	89.6124		Assembly	WGS		07/14/2005	Trypanosoma cruzi consortium [more]
59941	Trypanosoma cruzi JR cl. 4	Protists	Kinetoplasts	914063			In Progress	WGS			Genome Sequencing Center (GSC) at Washington University (WashU) School of Medicine [more]
40815	Trypanosoma cruzi Sylvio X10/1	Protists	Kinetoplasts	5693	46		Assembly	WGS	11X	02/09/2011	Karolinska Institutet
62515	Trypanosoma cruzi Y	Protists	Kinetoplasts	5693	30		In Progress		50X		J. Craig Venter Institute
50493	Trypanosoma cruzi strain Esmeraldo Esmeraldo cl. 3	Protists	Kinetoplasts	366581			In Progress				Genome Sequencing Center (GSC) at Washington University (WashU) School of Medicine [more]
18959	Trypanosoma rangeli Choachi	Protists	Kinetoplasts	429132			In Progress				BiowebDB [more]

You should find *Trypanosoma cruzi* strain CL Brener. You will also find that there are several ongoing genome sequencing projects listed for other strains of *Trypanosoma cruzi*: strains JR cl. 4, Sylvio X10/1, Y, and Esmeraldo Esmeraldo cl. 3.

If you look 7th column of the table, you will see that it says "Assembly" for strains CL Brener and Sylvio X10/1, meaning that genome assemblies are available for these two strains. Presumably the other strains are still being sequenced, and genome assemblies are not yet available.

The link 'GB' (in green) at the far right of the webpage gives a link to the NCBI record for the sequence. In this case, the link for *Trypanosoma cruzi* strain CL Brener leads us to the NCBI record for accession AAHK01000000. This is actually an accession for the *T. cruzi* strain CL Brener sequencing project, rather than for the genome sequence itself. On the top right of the page, you will see a link "Genome", and if you click on it, it will bring you to the NCBI accession NZ_AAHK00000000, the genome sequence for *Trypanosoma cruzi* strain CL Brener.

Of the other *T. cruzi* strains listed, there is only a 'GB' link for one other strain, Sylvio X10/1. If you click on the link for *Trypanosoma cruzi* strain Sylvio X10/1, it will bring you to the NCBI record for accession ADWP01000000, the accession for the *T. cruzi* strain Sylvio X10/1 sequencing project.

Note that the answer is slightly different for the answer from the first method above, which did not find the information on the genome projects for strains JR cl. 4, Sylvio X10/1, Y, and Esmeraldo Esmeraldo cl. 3, because the sequencing projects for these species are still ongoing.

Q10.

How many fully sequenced nematode worm species are represented in the NCBI Genome database?

To answer this question, you need to go to the NCBI Genome webpage.

In the search box at the top of the page, type Nematoda[ORGN] to search for genome sequences from nematode worms, using the Latin name for the nematode worms.

On the results page, you will see 'Items 1 - 20 of 63', indicating that 63 genome sequences from nematode worms have been found. If you look down the page, you will see however that many of these are mitochondrial genome sequences, rather than chromosomal genome sequences.

If you are just interested in chromosomal genome sequences, you can type 'Nematoda[ORGN] NOT mitochondrion' in the search box, to search for non-mitochondrial sequences. This should give you 16 sequences, which are all chromosomal genome sequences for nematode worms, including the species *Caenorhabditis elegans*, *Caenorhabditis remanei*, *Caenorhabditis briggsae*, *Loa loa* (which causes subcutaneous filariasis), and *Brugia malayi* (which causes lymphatic filariasis).

Thus, there are nematode genome sequences from five different species that have been fully sequenced (as of 15-Jun-2011). Because nematode worms are multi-chromosomal species, there may be several chromosomal sequences for each species.

Note that when you search the NCBI Genome database, you will find the NCBI records for completely sequenced genomes (completely sequenced nematode genomes in this case).

If you are interested in partially sequenced genomes, that is sequences from genome sequencing projects that are still in progress, you can go to the NCBI Genome Projects website. If you search the NCBI Genome Projects database for Nematoda[ORGN], you will find that genome sequencing projects for many other nematode species are ongoing, including for the species *Onchocerca volvulus* (which causes onchocerciasis), *Wuchereria bancrofti* (which causes lymphatic filariasis), and *Necator americanus* (which causes soil-transmitted helminthiasis).

Sequence Alignment

Q1.

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.

We can use SeqinR to retrieve these sequences by typing:

```
> library("seqinr")                          # load the SeqinR package
> choosebank("swissprot")                    # select the ACNUC sub-database to
↪be searched
> query("brugia", "AC=A8PZ80")               # search for the Brugia sequence
> brugiaseq <- getSequence(brugia$req[[1]])  # get the Brugia sequence
> query("loa", "AC=E1FTG0")                  # search for the Loa sequence
> loaseq <- getSequence(loa$req[[1]])        # get the Loa sequence
> closebank()                                # close the connection to the ACNUC
↪sub-database
```

Q2.

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?

We can use the Biostrings R package to answer this, by typing:

```
> library("Biostrings")                      # load the Biostrings package
> data(BLOSUM50)                             # load the BLOSUM50 scoring matrix
> brugiaseqstring <- c2s(brugiaseq)          # convert the Brugia sequence to a
↪string
> loaseqstring <- c2s(loaseq)                # convert the Loa sequence to a
↪string
> brugiaseqstring <- toupper(brugiaseqstring) # convert the Brugia sequence to
↪uppercase
> loaseqstring <- toupper(loaseqstring)      # convert the Loa sequence to a
↪string
> myglobalAlign <- pairwiseAlignment(brugiaseqstring, loaseqstring,
↪substitutionMatrix = "BLOSUM50",
  gapOpening = -9.5, gapExtension = -0.5, scoreOnly = FALSE) # align the two
↪sequences
> myglobalAlign
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [1] MK--LIVDSGHTGVNQLGGVFVNGRPLPDSTRQKI...
↪IESYKREQPSIFAWEIRDKLLHEKVCSPDTIPSA
  subject: [1] SSSNLFADSGHTGVNQLGGVFVNGRPLPDSTRQKI...
↪IESYKREQPSIFAWEIRDKLLHEKVCSPDTIPSV
  score: 777.5
```

The alignment score is 777.5.

Q3.

*Use the printPairwiseAlignment() 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.*

To do this, first you must copy and paste the printPairwiseAlignment() function into R.

Then you can use it to view the alignment that you obtained in Q2:

```
> printPairwiseAlignment(myglobalAlign)
  [1] "MK--LIVDSGHTGVNQLGGVFVNGRPLPDSTRQKIVDLAHQGARPCDISRILQVSNGCVS 58"
  [1] "SSSNLFADSGHTGVNQLGGVFVNGRPLPDSTRQKIVDLAHQGARPCDISRILQVSNGCVS 60"
  [1] " "
  [1] "KILCRYYESGTIRPRAIGGSKPRVATVSVCDKIESYKREQPSIFAWEIRDKLLHEKVCSP 118"
  [1] "KILCRYYESGTIRPRAIGGSKPRVATVSVCDKIESYKREQPSIFAWEIRDKLLHEKVCSP 120"
  [1] " "
  [1] "DTIPSA 178"
  [1] "DTIPSV 180"
  [1] " "
```

The two proteins are very similar over their whole lengths, with few gaps and mostly identities (few mismatches).

Q4.

*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?*

Again, we can use the Biostrings R package to answer this, by typing:

```
> data(BLOSUM62)                                  # load the BLOSUM62 scoring matrix
> myglobalAlign2 <- pairwiseAlignment(brugiaseqstring, loaseqstring,
↪substitutionMatrix = "BLOSUM62",
  gapOpening = -9.5, gapExtension = -0.5, scoreOnly = FALSE) # align the two
↪sequences
> myglobalAlign2
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [1] MK--LIVDSGHTGVNQLGGVFVNGRPLPDSTRQKI...
↪IESYKREQPSIFAWEIRDKLLHEKVCSPDTIPSA
  subject: [1] SSSNLFADSGHTGVNQLGGVFVNGRPLPDSTRQKI...
↪IESYKREQPSIFAWEIRDKLLHEKVCSPDTIPSV
  score: 593.5
```

The alignment score when BLOSUM62 is used is 593.5, while the score when BLOSUM50 is used is 777.5 (from
Q2).

We can print out the alignment and see if the alignment made using BLOSUM62 is different from that when
BLOSUM50 is used:

```
> printPairwiseAlignment(myglobalAlign2)
  [1] "MK--LIVDSGHTGVNQLGGVFVNGRPLPDSTRQKIVDLAHQGARPCDISRILQVSNGCVS 58"
  [1] "SSSNLFADSGHTGVNQLGGVFVNGRPLPDSTRQKIVDLAHQGARPCDISRILQVSNGCVS 60"
  [1] " "
  [1] "KILCRYYESGTIRPRAIGGSKPRVATVSVCDKIESYKREQPSIFAWEIRDKLLHEKVCSP 118"
  [1] "KILCRYYESGTIRPRAIGGSKPRVATVSVCDKIESYKREQPSIFAWEIRDKLLHEKVCSP 120"
  [1] " "
  [1] "DTIPSA 178"
```

```
[1] "DTIPSV 180"
[1] " "
```

The alignment made using BLOSUM62 is actually the same as that made using BLOSUM50, so it doesn't matter which scoring matrix we use in this case.

Q5.

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?

To answer this, we can first make 1000 random sequences using a multinomial model in which the probabilities of the 20 amino acids are set equal to their frequencies in the *Brugia malayi* Vab-3 protein.

First you need to first copy and paste the generateSeqsWithMultinomialModel() function into R, and then you can use it as follows:

```
> randomseqs <- generateSeqsWithMultinomialModel(brugiaseqstring,1000)
```

This makes a vector *randomseqs*, containing 1000 random sequences, each of the same length as the *Brugia malayi* Vab-3 protein.

We can then align each of the 1000 random sequences to the *Loa loa* Vab-3 protein, and store the scores for each of the 1000 alignments in a vector *randomscores*:

```
> randomscores <- double(1000) # Create a numeric vector with 1000 elements
> for (i in 1:1000)
  {
     score <- pairwiseAlignment(loaseqstring, randomseqs[i], substitutionMatrix =
↪"BLOSUM50",
        gapOpening = -9.5, gapExtension = -0.5, scoreOnly = TRUE)
     randomscores[i] <- score
  }
```

The score for aligning the *Brugia malayi* and *Loa loa* Vab-3 proteins using BLOSUM50 with a gap opening penalty of -10 and gap extension penalty of -0.5 was 777.5 (from Q2).

We can see what fraction of the 1000 alignments between the random sequences (of the same composition as *Brugia malayi* Vab-3) and *Loa loa* Vab-3 had scores equal to or higher than 777.5:

```
> sum(randomscores >= 777.5)
[1] 0
```

We see that none of the 1000 alignments had scores equal to or higher than 777.5.

Thus, the *p*-value for the alignment of *Brugia malayi* and *Loa loa* Vab-3 proteins is 0, and we can therefore conclude that the alignment score is statistically significant (as it is less than 0.05). Therefore, it is very likely that the *Brugia malayi* Vab-3 and *Loa loa* Vab-3 proteins are homologous (related).

Q6.

What is the optimal global alignment score between the Brugia malayi Vab-6 protein and the Mycobacterium leprae chorismate lyase protein?

To calculate the optimal global alignment score, we must first retrieve the *M. leprae* chorismate lyase sequence:

```
> choosebank("swissprot")
> query("leprae", "AC=Q9CD83")
> lepraeseq <- getSequence(leprae$req[[1]])
> closebank()
```

```
> lepraeseqstring <- c2s(lepraeseq)
> lepraeseqstring <- toupper(lepraeseqstring)
```

We can then align the *Brugia malayi* Vab-3 protein sequence to the *M. leprae* chorismate lyase sequence:

```
> myglobalAlign3 <- pairwiseAlignment(brugiaseqstring, lepraeseqstring,
↪substitutionMatrix = "BLOSUM50",
  gapOpening = -9.5, gapExtension = -0.5, scoreOnly = FALSE) # align the two
↪sequences
> myglobalAlign3
  Global PairwiseAlignedFixedSubject (1 of 1)
  pattern: [1] M----------------KLIVDSGHTGVNQLGGV...------INYAKQNNNLL----DRFILP---
↪FSKL
  subject: [1] MTNRTLSREEIRKLDRDLRILVATNGT-LTRVLNV...
↪DTPREELDRCQYSNDIDTRSGDRFVLHGRVFKNL
  score: 67.5
```

The alignment score is 67.5.

We can print out the alignment as follows:

```
> printPairwiseAlignment(myglobalAlign3)
  [1] "M----------------KLIVDSGHTGVNQLGGVFVNGRPLPDSTRQKIVDLAHQGARP 43"
  [1] "MTNRTLSREEIRKLDRDLRILVATNGT-LTRVLNVVANEEIVVDIINQQLLDVA-----P 54"
  [1] " "
  [1] "-------CDISRILQ---VSNGCVSKILCRYYESGTI---RPRAIGG-----SKPRVATV 85"
  [1] "KIPELENLKIGRILQRDILLKGQKSGILFVAAESLIVIDLLPTAITTYLTKTHHP-IGEI 113"
  [1] " "
  [1] "SVCDKIESYKREQ-------PSIFA----WEIRDKLLHEKVCSPDTIPSAVV-------- 126"
  [1] "MAASRIETYKEDAQVWIGDLPCWLADYGYWDL--------------PKRAVGRRYRIIA 158"
  [1] " "
  [1] "--EAIIV----------------INYAKQNNNLL----DRFILP---FSKL 160"
  [1] "GGQPVIITTEYFLRSVFQDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKNL 218"
  [1] " "
```

The alignment does not look very good, it contains many gaps and mismatches and few matches.

In Q5, we made a vector *randomseqs* that contains 1000 random sequences generated using a multinomial model in which the probabilities of the 20 amino acids are set equal to their frequencies in the *Brugia malayi* Vab-3 protein.

To calculate a statistical significance for the alignment between *Brugia malayi* Vab-3 and *M. leprae* chorismate lyase, we can calculate the alignment scores for the 1000 random sequences to *M. leprae* chorismate lyase:

```
> randomscores <- double(1000) # Create a numeric vector with 1000 elements
> for (i in 1:1000)
  {
     score <- pairwiseAlignment(lepraeseqstring, randomseqs[i], substitutionMatrix
↪= "BLOSUM50",
       gapOpening = -9.5, gapExtension = -0.5, scoreOnly = TRUE)
     randomscores[i] <- score
  }
```

We can then see how many of the 1000 alignment score exceed the actual alignment score for *B. malayi* Vab-3 and *M. leprae* chorismate lyase (67.5):

```
> sum(randomscores >= 67.5)
[1] 22
```

We see that 22 of the 1000 scores for the 1000 random sequences to *M. leprae* chorismate lyase are higher than the actual alignment score of 67.5. Therefore the *P-value* for the alignment score is 22/1000 = 0.022. This is just under 0.05, and so is quite near to the general cutoff for statistical significance (0.05). However, in fact it is close enough to 0.05 that we should have some doubt about whether the alignment is statistically significant.

In fact, the *B. malayi* Vab-3 and *M. leprae* chorismate lyase proteins are not known to be homologous (related), and so it is likely that the relatively high alignment score (67.5) is just due to chance alone.

Multiple Alignment and Phylogenetic Trees

Q1.

Calculate the genetic distances between the following NS1 proteins from different Dengue virus strains: Dengue virus 1 NS1 protein (Uniprot Q9YRR4), Dengue virus 2 NS1 protein (UniProt Q9YP96), Dengue virus 3 NS1 protein (UniProt B0LSS3), and Dengue virus 4 NS1 protein (UniProt Q6TFL5). Which are the most closely related proteins, and which are the least closely related, based on the genetic distances?

To retrieve the sequences of the four proteins, we can use the retrieveseqs() function in R, to retrieve the sequences from the "swissprot" ACNUC sub-database:

```
> library("seqinr")                                    # Load the SeqinR package
> seqnames <- c("Q9YRR4", "Q9YP96", "B0LSS3", "Q6TFL5")  # Make a vector
→containing the names of the sequences
> seqs <- retrieveseqs(seqnames,"swissprot")           # Retrieve the sequences
→and store them in list variable "seqs"
```

We then can write out the sequences to a FASTA-format file called "NS1.fasta", by typing:

```
> write.fasta(seqs, seqnames, file="NS1.fasta")
```

We can then use the CLUSTAL software to make a multiple alignment of the protein sequences in NS1.fasta, and store it in a PHYLIP-format alignment file called "NS1.phy", as described in the chapter.

The next step is to read the PHYLIP-format alignment into R, and calculate the genetic distances between the protein sequences, by typing:

```
> NS1aln  <- read.alignment(file = "NS1.phy", format = "phylip")
> NS1dist <- dist.alignment(NS1aln)
> NS1dist
              Q9YRR4     Q9YP96     B0LSS3
  Q9YP96      0.2544567
  B0LSS3      0.2302831  0.2268713
  Q6TFL5      0.3058189  0.3328595  0.2970443
```

We see that the two sequences with the greatest genetic distance are Q6TFL5 (Dengue virus 4 NS1) and Q9YP96 (Dengue virus 2 NS1), which have a genetic distance of about 0.33. The two sequences with the smallest genetic distance are Q9YRR4 (Dengue virus 1 NS1) and B0LSS3 (Dengue virus 3 NS1), which have a genetic distance of about 0.23.

Q2.

Build an unrooted phylogenetic tree of the NS1 proteins from Dengue virus 1, Dengue virus 2, Dengue virus 3 and Dengue virus 4, using the neighbour-joining algorithm. Which are the most closely related proteins, based on the tree? Based on the bootstrap values in the tree, how confident are you of this?

We can build an unrooted phylogenetic tree of the NS1 proteins using the neighbour-joining algorithm by typing:

```
> NS1alntree <- unrootedNJtree(NS1aln,type="protein")
```
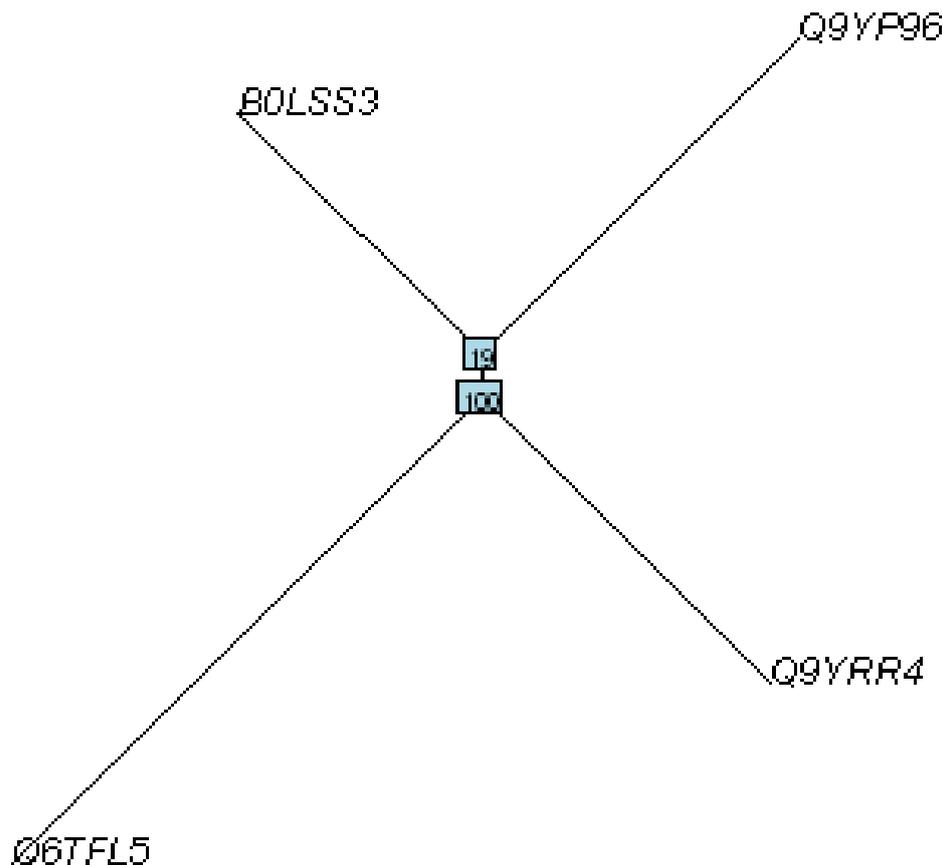
We see in the tree that Q6TFL5 (Dengue virus 4 NS1) and Q9YRR4 (Dengue virus 1 NS1) are grouped together, with a bootstrap value of 100%, which is a high bootstrap value, so we are reasonably confident of this grouping.

The other two proteins, B0LSS3 (Dengue virus 3 NS1) and Q9YP96 (Dengue virus 2 NS1) are grouped together, but the bootstrap value for the node representing the ancestor of this clade is just 19%.

One thing that is surprising is that Q6TFL5 and Q9YRR4 were not the two closest proteins when we calculated the genetic distance (in Q1), and we should bear this in mind, as it should make us a little bit cautious in trusting this phylogenetic tree.

Q3.

Build an unrooted phylogenetic tree of the NS1 proteins from Dengue viruses 1-4, based on a filtered alignment of the four proteins (keeping alignment columns in which at least 30% of letters are not gaps, and in which at least 30% of pairs of letters are identical). Does this differ from the tree based on the unfiltered alignment (in Q2)? Can you explain why?

To filter the alignment of the NS1 proteins, we can use the "cleanAlignment()" function:

```
> cleanedNS1aln <- cleanAlignment(NS1aln, 30, 30)
```

We can then build an unrooted tree based on the filtered alignment:

```
> cleanedNS1alntree <- unrootedNJtree(cleanedNS1aln,type="protein")
```
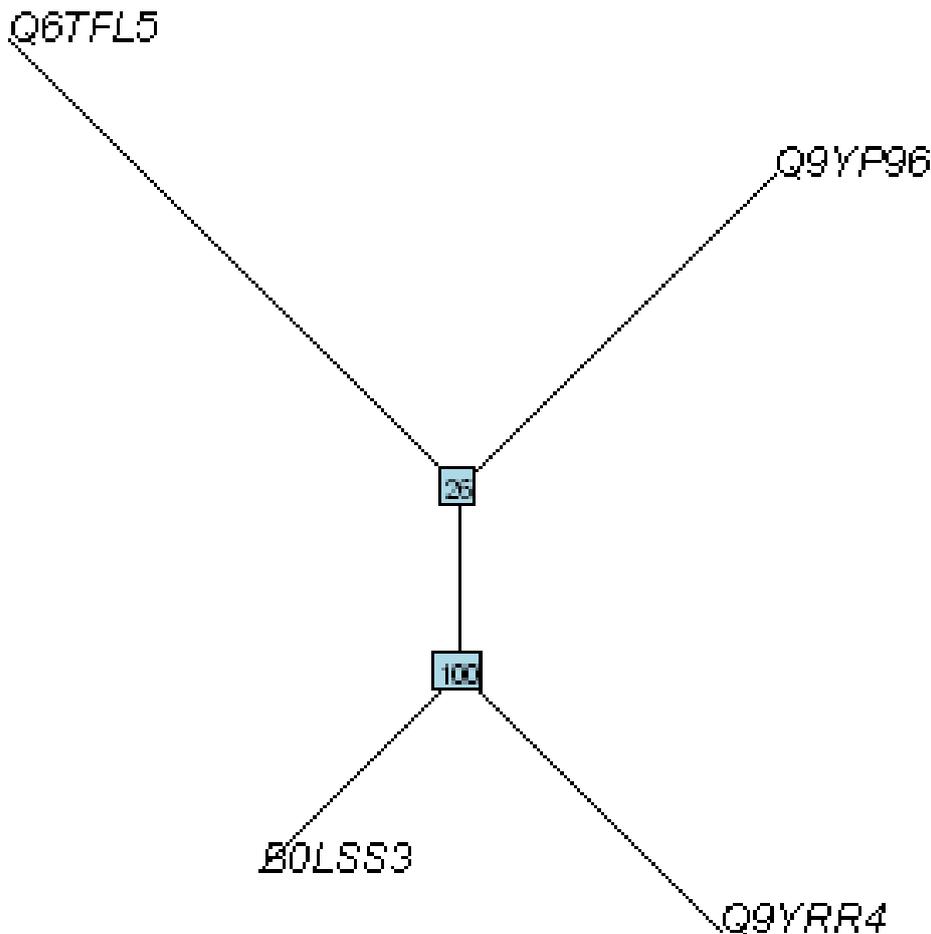
We find that B0LSS3 (Dengue virus 3 NS1) and Q9YRR4 (Dengue virus 1 NS1) are grouped together with bootstrap of 100%. This disagrees with what we found in the phylogenetic tree based on the unfiltered alignment (in Q2), in which B0LSS3 was grouped with Q9YP96. However, it agrees with what we found when we calculated the genetic distance matrix (in Q1), which suggested that B0LSS3 is most closely related to Q9YRR4.

Why do the filtered and unfiltered alignments disagree? To find out, it is a good idea to print out both alignments:

```
> printMultipleAlignment(NS1aln)
  [1] "------------------------------------------------------------ 0"
  [1] "DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAQEEGICGIRSV 60"
  [1] "------------------------------------------------------------ 0"
  [1] "DMGCVVSWNGKELKCGSGIFVIDNVHTRTEQYKFQPESPARLASAILNAHKDGVCGVRST 60"
  [1] " "
  [1] "------------------------------------------------------------ 0"
  [1] "TRLENLMWKQITPELNHILSENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKAWGK 120"
  [1] "------------------------------------------------------------ 0"
  [1] "TRLENVMWKQITNELNYVLWEGGHDLTVVAGDVKGVLTEGKRALTPPVNDLKYSWKTWGK 120"
  [1] " "
  [1] "------------------------------------------------------------ 0"
  [1] "AKMLSTESHNQTFLIDGPETAECPNTNRAWNSLEVEDYGFGVFTTNIWLKLKEKQDAFCD 180"
  [1] "------------------------------------------------------------ 0"
  [1] "AKIFTLEARNSTFLIDGPDTSECPNERRAWNFLEVEDYGFGMFTTNIWMKFREGSSEVCD 180"
  [1] " "
  [1] "----------------DMGYWIESEKNETWKLARASFIEVKTCIWPKSHTLWSNGVWESE 44"
  [1] "SKLMSAAIKDNRAVHADMGYWIESALNDTWKIEKASFIEVKNCHWPKSHTLWSNGVLESE 240"
  [1] "-----------ASHADMGYWIESQKNGSWKLEKASLIEVKTCTWPKSHTLWSNGVLESD 48"
```

```
[1] "HRLMSAAIKDQKAVHADMGYWIESSKNQTWQIEKASLIEVKTCLWPKTHTLWSNGVLESQ 240"
[1] " "
[1] "MIIPKIYGGPISQHNYRPGYFTQTAGPWHLGKLELDFDLCEGTTVVVDEHCGNRGPSLRT 104"
[1] "MIIPKNFAGPVSQHNYRPGYHTQIAGPWHLGKLEMDFDFCDGTTVVVTEDCGNRGPSLRT 300"
[1] "MIIPKSLAGPISQHNYRPGYHTQTAGPWHLGKLELDFNYCEGTTVVITENCGTRGPSLRT 108"
[1] "MLIPRSYAGPFSQHNYRQGYATQTMGPWHLGKLEINFGECPGTTVAIQEDCGHRGPSLRT 300"
[1] " "
[1] "TTVTGKIIHEWCCRFCTLPPLRFRGEDGCWYGMEI---------------- 147"
[1] "TTASGKLITEWCCRSCTLPPLRYRGEDGCWYGMEIRPLKEKEENLVNSLVTA 360"
[1] "TTVSGKLIHEWCCRSCTLPPLRYMGEDG--------------------- 144"
[1] "TTASGKLVTQWCCRSCAMPPLRFLGEDGCWYGMEIRPLSEKEENMVKSQVTA 360"
[1] " "
```

We can see that the unfiltered (original) alignment (above) contains a lot of columns with gaps in them. This could possibly be adding noise to the phylogenetic analysis.

Let's print out the filtered alignment now:

```
> printMultipleAlignment(cleanedNS1aln)
  [1] "----------------------------------------------------------- 0"
  [1] "DGCVVSWKELKCGSGIFDNVHTTEQYKFQPESPLASAIAGCGRSTRLENMWKQITELNLE 60"
  [1] "----------------------------------------------------------- 0"
  [1] "DGCVVSWKELKCGSGIFDNVHTTEQYKFQPESPLASAIAGCGRSTRLENMWKQITELNLE 60"
  [1] " "
  [1] "----------------------------------------------------------- 0"
  [1] "LTGDKGGKRLPLKYSWKWGKAKENTFLIDGPTECPNRAWNLEVEDYGFGFTTNIWKECDL 120"
  [1] "----------------------------------------------------------- 0"
  [1] "LTGDKGGKRLPLKYSWKWGKAKENTFLIDGPTECPNRAWNLEVEDYGFGFTTNIWKECDL 120"
  [1] " "
  [1] "----------DMGYWIESKNTWKLARASFIEVKTCWPKSHTLWSNGVWESMIIPKGGPS 49"
  [1] "MSAAIKDAVHADMGYWIESLNTWKIEKASFIEVKNCWPKSHTLWSNGVLESMIIPKAGPS 180"
  [1] "-------ASHADMGYWIESKNSWKLEKASLIEVKTCWPKSHTLWSNGVLESMIIPKAGPS 53"
  [1] "MSAAIKDAVHADMGYWIESKNTWQIEKASLIEVKTCWPKTHTLWSNGVLESMLIPRAGPS 180"
  [1] " "
  [1] "QHNYRPGYTQTAGPWHLGKLEDFCGTTVVVECGRGPSLRTTTVTGKIIHEWCCRFCTLPP 109"
  [1] "QHNYRPGYTQIAGPWHLGKLEDFCGTTVVVECGRGPSLRTTTASGKLITEWCCRSCTLPP 240"
  [1] "QHNYRPGYTQTAGPWHLGKLEDFCGTTVVIECGRGPSLRTTTVSGKLIHEWCCRSCTLPP 113"
  [1] "QHNYRQGYTQTMGPWHLGKLENFCGTTVAIECGRGPSLRTTTASGKLVTQWCCRSCAMPP 240"
  [1] " "
  [1] "LRFGEDGCWYGMEI------------ 156"
  [1] "LRYGEDGCWYGMEIRPLEKEENVSVTA 300"
  [1] "LRYGEDG------------------ 153"
  [1] "LRFGEDGCWYGMEIRPLEKEENVSVTA 300"
  [1] " "
```

The unfiltered alignment contains far fewer "gappy" columns (columns where two or more sequences have gaps) compared to the original unfiltered alignment. It is likely that the gappy columns in the original unfiltered alignment were adding noise to the phylogenetic analysis, and that the phylogenetic tree based on the filtered alignment is more reliable in this case.

Q4. *Build a rooted phylogenetic tree of the Dengue NS1 proteins based on a filtered alignment, using the Zika virus protein as the outgroup. Which are the most closely related Dengue virus proteins, based on the tree? What extra information does this tree tell you, compared to the unrooted tree in Q2?*

First we need to obtain the Zika virus protein (UniProt accession Q32ZE1):

```
> seqnames <- c("Q9YRR4", "Q9YP96", "B0LSS3", "Q6TFL5", "Q32ZE1")  # Make a vector
→containing the names of the sequences
> seqs <- retrieveseqs(seqnames,"swissprot")                       # Retrieve the
→sequences and store them in list variable "seqs"
```

We then write out the sequences to a FASTA-format file called "NS1b.fasta":

```
> write.fasta(seqs, seqnames, file="NS1b.fasta")
```

We then use CLUSTAL to make a PHYLIP-format alignment, and save it as "NS1b.phy".

We then read the alignment into R:

```
> NS1baln  <- read.alignment(file = "NS1b.phy", format = "phylip")
```

We then discard unreliable columns from the alignment:

```
> cleanedNS1baln <- cleanAlignment(NS1baln, 30, 30)
```

We then can build a rooted phylogenetic tree using the Zika virus protein (accession Q32ZE1) as the outgroup, by using the rootedNJtree() function:

```
> cleanedNS1balntree <- rootedNJtree(cleanedNS1baln, "Q32ZE1",type="protein")
```
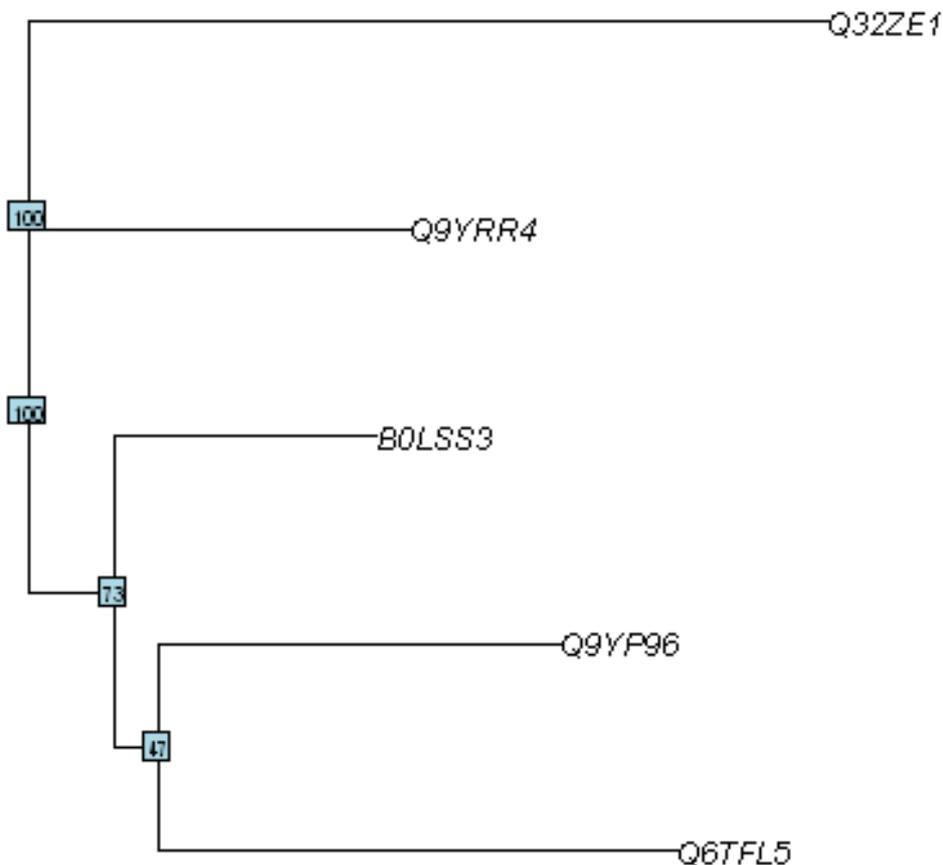

We see in this tree that Q9YP96 (Dengue virus 2 NS1) and Q6TFL5 (Dengue virus 4 NS1) are grouped together with bootstrap 47%. The next closest sequence is B0LSS3 (Dengue virus 3 NS1). The Q9YRR4 sequence (Dengue virus 1 NS1) diverged earliest of the four Dengue virus NS1 proteins, as it is grouped with the outgroup.

Note that in Q3, we found that B0LSS3 (Dengue virus 3 NS1) and Q9YRR4 (Dengue virus 1 NS1) were grouped together in an unrooted tree. The current rooted tree is consistent with this; it has B0LSS3 and Q9YRR4 as the two earliest diverging Dengue NS1 proteins, as they are nearest to the outgroup in the tree.

Thus, the rooted tree tells you which of the Dengue virus NS1 proteins branched off the earliest from the ancestors of the other proteins, and which branched off next, and so on... We were not able to tell this from the unrooted tree.

Computational Gene-finding

Q1.

How many ORFs are there on the forward strand of the DEN-1 Dengue virus genome (NCBI accession NC_001477)?

To answer this, we can use the findORFsinSeq() function to find ORFs on the forward strand of the DEN-1 Dengue virus sequence. This function requires a string of characters as its input, so we first use "c2s()" to convert the Dengue virus sequence to a string of characters:

```
> dengueseqstring <- c2s(dengueseq)              # Convert the Dengue sequence to a
↪string of characters
> mylist <- findORFsinSeq(dengueseqstring)       # Find ORFs in "dengueseqstring"
> orflengths <- mylist[[3]]                      # Find the lengths of ORFs in
↪"dengueseqstring"
> length(orflengths)                             # Find the number of ORFs that were
↪found
  [1] 116
```

We find that there are 116 ORFs on the forward strand of the DEN-1 Dengue virus genome.

Q2.

What are the coordinates of the rightmost (most 3', or last) ORF in the forward strand of the DEN-1 Dengue virus genome?

To answer this, we need to get the coordinates of the ORFs in the DEN-1 Dengue virus genome, as follows:

```
> dengueseqstring <- c2s(dengueseq)              # Convert the Dengue sequence to a
↪string of characters
> mylist <- findORFsinSeq(dengueseqstring)       # Find ORFs in "dengueseqstring"
> starts <- mylist[[1]]                          # Start positions of ORFs
> stops <- mylist[[2]]                           # Stop positions of ORFs
```

The vector *starts* contains the start coordinates of the predicted start codons, and the vector *stops* contains the end coordinates of the predicted stop codons. We know there are 116 ORFs on the forward strand (from Q1), and we want the coordinates of the 116th ORF. Thus, we type:

```
> starts[116]
  [1] 10705
> stops[116]
  [1] 10722
```

This tells us that the most 3' ORF has a predicted start codon from 10705-10707 and a predicted stop codon from 10720-10722. Thus, the coordinates of the 3'-most ORF are 10705-10722.

Q3.

What is the predicted protein sequence for the rightmost (most 3', or last) ORF in the forward strand of the DEN-1 Dengue virus genome?

To get the predicted protein sequence of the 5'-most ORF (from 10705-10722), we type:

```
> myorfvector <- dengueseq[10705:10722] # Get the DNA sequence of the ORF
> seqinr::translate(myorfvector)
  [1] "M" "E" "W" "C" "C" "*"
```

The sequence of the ORF is "MEWCC".

Q4.

How many ORFs are there of 30 nucleotides or longer in the forward strand of the DEN-1 Dengue virus genome sequence?

The findORFsinSeq() function returns a list variable, the third element of which is a vector containing the lengths of the ORFs found. Thus we can type:

```
> dengueseqstring <- c2s(dengueseq)              # Convert the Dengue sequence to a␣
↪string of characters
> mylist <- findORFsinSeq(dengueseqstring)       # Find ORFs in "dengueseqstring"
> orflengths <- mylist[[3]]                      # Find the lengths of ORFs in
↪"dengueseqstring"
> summary(orflengths >= 30)
    Mode    FALSE     TRUE     NA's
  logical     54       62        0
```

This tells us that 62 ORFs on the forward strand of the DEN-1 Dengue virus are 30 nucleotides or longer.

Q5.

How many ORFs longer than 248 nucleotides are there in the forward strand of the DEN-1 Dengue genome sequence?

To answer this, we type:

```
> summary(orflengths >= 248)
    Mode    FALSE     TRUE     NA's
  logical    114        2        0
```

This tells us that there are 2 ORFs of 248 nucleotides or longer on the forward strand.

Q6.

If an ORF is 248 nucleotides long, what length in amino acids will its predicted protein sequence be?

If we include the predicted stop codon in the length of the ORF, it means that the last three bases of the ORF are not coding for any amino acid. Therefore, the length of the ORF that is coding for amino acids is 245 bp. Each amino acid is coded for by 3 bp, so there can be 245/3 = 81 amino acids. Thus, the predicted protein sequence will be 81 amino acids long.

Q7.

How many ORFs are there on the forward strand of the rabies virus genome (NCBI accession NC_001542)?

We first retrieve the rabies virus sequence by copying and pasting the "getncbiseq()" function into R, and then typing:

```
> rabiesseq <- getncbiseq("NC_001542")
```

We then find the ORFs in the forward strand by typing:

```
> rabiesseqstring <- c2s(rabiesseq)              # Convert the rabies sequence to a␣
↪string of characters
> rabieslist <- findORFsinSeq(rabiesseqstring)   # Find ORFs in "rabiesseqstring"
> rabiesorflengths <- rabieslist[[3]]            # Find the lengths of ORFs in
↪"rabiesseqstring"
> length(rabiesorflengths)                       # Find the number of ORFs that were␣
↪found
  [1] 111
```

There were 111 ORFs on the forward strand.

Q8.

What is the length of the longest ORF among the 99% of longest ORFs in 10 random sequences of the same lengths and composition as the rabies virus genome sequence?

We generate 10 random sequences using a multinomial model in which the probabilities of the 4 bases are set equal to their frequencies in the rabies sequence:

```
> randseqs <- generateSeqsWithMultinomialModel(rabiesseqstring, 10) # Generate 10␣
↪random sequences using the multinomial model
> randseqorflengths <- numeric()                # Tell R that we want to make a new␣
↪vector of numbers
> for (i in 1:10)
  {
    print(i)
    randseq <- randseqs[i]                       # Get the ith random sequence
    mylist <- findORFsinSeq(randseq)             # Find ORFs in "randseq"
    lengths <- mylist[[3]]                        # Find the lengths of ORFs in
↪"randseq"
    randseqorflengths <- append(randseqorflengths, lengths,␣
↪after=length(randseqorflengths))
  }
```

To find the length of the longest ORF among the 99% of the longest ORFs in the 10 random sequences, we find the 99th quantile of *randomseqorflengths*:

```
> quantile(randseqorflengths, probs=c(0.99))
  99%
  259.83
```

That is, the longest of the longest 99% of ORFs in the random sequences is 260 nucleotides.

Q9.

How many ORFs are there in the rabies virus genome that are longer than the threshold length that you found in Q8?

To answer this, we type:

```
> summary(rabiesorflengths > 260)
     Mode    FALSE    TRUE    NA's
  logical     105       6       0
```

There are 6 ORFs in the rabies virus genome that are longer than the threshold length found in Q8 (260 nucleotides).

Comparative Genomics

Q1.

How many Mycobacterium ulcerans genes are there in the current version of the Ensembl Bacteria database?

Q2.

How many of the Mycobacterium ulcerans Ensembl genes are protein-coding genes?

Q3.

How many Mycobacterium ulcerans protein-coding genes have Mycobacterium leprae orthologues?

Q4.

How many of theMycobacterium ulcerans protein-coding genes have one-to-one orthologues in Mycobacterium leprae?

Q5.

How many Mycobacterium ulcerans genes have Pfam domains?

Q6.

What are the top 5 most common Pfam domains in Mycobacterium ulcerans genes?

Q7.

How many copies of each of the top 5 domains found in Q6 are there in the Mycobacterium ulcerans protein set?

Q8.

How many of copies are there in the Mycobacterium lepraae protein set, of each of the top 5 Mycobacterium ulcerans Pfam protein domains?

Q9.

Are the numbers of copies of some domains different in the two species?

Q10.

Of the differences found in Q9, are any of the differencess statistically significant?

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Answers to Revision Exercises

Revision Exercises 1

Q1.

What is the length of (total number of base-pairs in) the Schistosoma mansoni mitochondrial genome (NCBI accession NC_002545), and how many As, Cs, Gs and Ts does it contain?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and type the *S. mansoni* mitochondrial genome (accession NC_002545) in the search box, and press 'Search'.

On the search results page, you should see '1' beside the word 'Nucleotide', meaning that there was one hit to a sequence record in the NCBI Nucleotide database, which contains DNA and RNA sequences. If you click on the word 'Nucleotide', it will bring you to the sequence record, which should be the NCBI sequence record for the *S.mansoni* mitochondrial genome (ie. for accession NC_002545).

To save the sequence as FASTA-format file, click on 'Send' at the top right of the page, and choose 'File', then select 'FASTA' from the drop-down list labelled 'Format', then click 'Create File'. Save the file with a name that you will remember (eg. "smansoni.fasta") in your "My Documents" folder.

You can then read the sequence into R by typing:

```
> library("seqinr")                              # load the SeqinR R package
> smansoni <- read.fasta(file="smansoni.fasta")  # read in the sequence file
> smansoniseq <- smansoni[[1]]                    # get the sequence
> length(smansoniseq)                             # get the length of the
↪sequence
  [1] 14415
> table(smansoniseq)                              # get the number of As, Cs, Gs,
↪ Ts
  smansoniseq
    a    c    g    t
  3654 1228 3307 6226
```

Thus, the mitochondrial genome is 14415 bases long, and consists of 3654 As, 1228 Cs, 3307 Gs and 6226 Ts.

Note that, as far as I know, it is not possible to retrieve the sequence for accession NC_002545 directly using the "query()" function in SeqinR, because the *S. mansoni* mitochondrial genome sequence does not seem to be stored in any of the ACNUC sub-databases.

Q2.

What is the length of the Brugia malayi mitochondrial genome (NCBI accession NC_004298), and how many As, Cs, Gs and Ts does it contain?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and type the *B. malayi* mitochondrial genome (accession NC_004298) in the search box, and press 'Search'.

As in Q1, go to the NCBI record for the sequence, and save the sequence in a FASTA format file, for example, called "bmalayi.fasta".

Then read the sequence into R, and get its length and composition by typing:

```
> bmalayi <- read.fasta(file="bmalayi.fasta")    # read in the sequence file
> bmalayiseq <- bmalayi[[1]]                      # get the sequence
> length(bmalayiseq)                              # get the length of the
↪sequence
  [1] 13657
> table(bmalayiseq)                               # get the number of As, Cs, Gs,
↪ Ts
  bmalayiseq
```

```
   a    c    g    t
 2950 1054 2297 7356
```

The sequence is 13657 bases long, and consists of 2950 As, 1054 Cs, 2297 Gs and 7356 Ts.

Note that, as far as I know, it is not possible to retrieve the sequence for accession NC_004298 directly using the "query()" function in SeqinR, because the *B. malayi* mitochondrial genome sequence does not seem to be stored in any of the ACNUC sub-databases.

Q3.

What is the probability of the Brugia malayi mitochondrial genome sequence (NCBI accession NC_004298), according to a multinomial model in which the probabilities of As, Cs, Gs and Ts (pA, pC, pG, and pT) are set equal to the fraction of As, Cs, Gs and Ts in the Schistosoma mansoni mitochondrial genome?

First we can calculate the frequencies of A, C, G and T in the *S. mansoni* mitochondrial sequence. We can do this by making a table of the counts of As, Cs, Gs and Ts, and dividing the counts of the bases by the total sequence length to get frequencies:

```
> mytable <- table(smansoniseq)
> mytable
    a    c    g    t
 3654 1228 3307 6226
> mytable <- mytable/length(smansoniseq) # Divide the counts by the sequence
→length, to get frequencies
> mytable
         a          c          g          t
 0.25348595 0.08518904 0.22941381 0.43191120
> freqA <- mytable[["a"]]                 # Get the frequency of As
> freqC <- mytable[["c"]]                 # Get the frequency of Cs
> freqG <- mytable[["g"]]                 # Get the frequency of Gs
> freqT <- mytable[["t"]]                 # Get the frequency of Ts
> probabilities <- c(freqA,freqC,freqG,freqT) # Make a vector containing the
→frequencies of As,Cs,Gs,Ts
> probabilities
  [1] 0.25348595 0.08518904 0.22941381 0.43191120
```

First we need to make a function to calculate the probability of a sequence, given a particular multinomial model (with a certain *pA*, *pC*, *pG*, *pT*). To do this, we can write the following R function "multinomialprob()":

```
> multinomialprob <- function(mysequence, probabilities)
  {
     nucleotides  <- c("A", "C", "G", "T") # Define the alphabet of nucleotides
     names(probabilities) <- nucleotides
     mysequence   <- toupper(mysequence)# Convert the sequence to uppercase
→letters
     seqlength    <- length(mysequence) # Get the length of the input sequence
     seqprob      <- numeric()          # Make a variable to hold to probability
→of the whole sequence
     for (i in 1:seqlength)             # For each letter in the input sequence
     {
        nucleotide <- mysequence[i]     # Find the ith nucleotide in the sequence
        # Calculate the probability of the ith nucleotide in the sequence
        nucleotideprob <- probabilities[nucleotide]
        # The probability of the whole sequence is calculated by multiplying
→together
        # the probabilities of the nucleotides at each sequence position
        if (i == 1) { seqprob <- nucleotideprob[[1]]          }
        else        { seqprob <- seqprob * nucleotideprob[[1]] }
     }
     # Return the value of the probability of the whole sequence
```

```
    return(seqprob)
  }
```

The function multinomialprob() takes as its arguments (inputs) a vector that contains the DNA sequence, and a vector containing the probabilities *pA*, *pC*, *pG*, and *pT*.

You will need to copy and paste this function into R to use it. You can then use it to calculate the probability of the *B. malayi* mitochondrial sequence, using a multinomial model where *pA*, *pC*, *pG*, *pT* are set equal to the fraction of As, Cs, Gs, and Ts in the *S. mansoni* mitohondrial sequence (which we have already stored in the vector *probabilities*, see above):

```
> multinomialprob(bmalayiseq, probabilities)
  0
```

In this case, the probability is so small that it is effectively zero.

Q4.

What are the top three most frequent 4-bp words (4-mers) in the genome of the bacterium Chlamydia trachomatis strain D/UW-3/CX (NCBI accession NC_000117), and how many times do they occur in its sequence?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and type the *C. trachomatis* D/UW-3/CX genome (accession NC_000117) in the search box, and press 'Search'.

As in Q1, go to the NCBI record for the sequence, and save the sequence in a FASTA format file, for example, called "ctrachomatis.fasta".

Alternatively, you can retrieve the sequence using the SeqinR package. The sequence is a fully sequenced bacterial genome, so is in the ACNUC sub-database called "bacterial". Thus, we type in R:

```
> choosebank("bacterial")                              # select the ACNUC sub-
↪database to search
> query("ctrachomatis", "AC=NC_000117")                # specify the query
> ctrachomatisseq <- getSequence(ctrachomatis$req[[1]]) # get the sequence
> closebank()                                          # close the connection to
↪the ACNUC sub-database
```

We can now find the most frequent 4-bp words in the sequence by using the "count()" function from SeqinR:

```
> mytable <- count(ctrachomatisseq, 4)                 # get the count for each
↪4-bp word
> sort(mytable)                                        # sort the 4-bp words, by
↪the number of occurrences of each word
  ccgg cggg ggcc cccg cgcg cggc gccg cgcc ggcg cggt gccc cacg gggc
  1180 1198 1206 1215 1287 1321 1334 1407 1435 1481 1512 1520 1537
  cgtg accg ggtc gacc cgac gtcg gcgg ccgc acgg gacg cgtc ccgt gtac
  1541 1545 1558 1567 1606 1647 1658 1678 1716 1750 1786 1802 1802
  ...
  agag agct ctct tatt cttc tttg caaa gaag ttta taaa attt aaat tttc
  6836 6860 6937 6946 7234 7280 7289 7353 7671 7731 8100 8144 8462
  gaaa aaag cttt tctt aaga ttct agaa tttt aaaa
  8563 9099 9199 10060 10069 10492 10581 14021 14122
```

The three most frequent 4-bp words are "aaaa" (14122 occurrences), "tttt" (14021 occurrences) and "agaa" (10581 occurrences).

Q5.

Write an R function to generate a random DNA sequence that is n letters long (that is, n bases long) using a multinomial model in which the probabilities pA, pC, pG, and pT are set equal to the fraction of As, Cs, Gs and Ts in the Schistosoma mansoni mitochondrial genome.

In Q3 above, we stored the frequencies of A, C, G and T in the *S. mansoni* mitochondrial genome in a vector called *probabiltiies*:

```
> probabilities
  [1] 0.25348595 0.08518904 0.22941381 0.43191120
```

The R function "generateSeqWithMultinomialModel()" below is an R function for generating a random sequence with a multinomial model, where the probabilities of the different letters are set equal to the fraction of As, Cs, Gs, and Ts in the *S. mansoni* mitochondrial genome (ie. with vector *probabilities* as its input):

```
> generateSeqWithMultinomialModel <- function(n, probabilities)
  {
     # Define the letters in the alphabet
     letters <- c("A", "C", "G", "T")
     # Make a random sequence of length n letters, using the multinomial model␣
→with probabilities "probabilities"
     seq <- sample(letters, n, rep=TRUE, prob=probabilities) # Sample with␣
→replacement
     # Return the sequence
     return(seq)
  }
```

To use this function to generate a 10-bp random sequence, using vector *probabilities* as input, we would type:

```
> generateSeqWithMultinomialModel(10, probabilities)
  [1] "T" "A" "T" "G" "T" "G" "G" "A" "G" "G"
```

Each time we call the function, it will create a slightly different 10-bp sequence:

```
> generateSeqWithMultinomialModel(10, probabilities)
  [1] "A" "G" "T" "A" "G" "G" "T" "T" "T" "T"
> generateSeqWithMultinomialModel(10, probabilities)
  [1] "C" "G" "A" "T" "A" "T" "G" "T" "T" "A"
```

Q6.

Give an example of using your function from Q5 to calculate a random sequence that is 20 letters long, using a multinomial model with pA =0.28, pC =0.21, pG =0.22, and pT =0.29.

First we need to define a vector *myprobabilities* containing the probabilities of A, C, G, and T:

```
> myprobabilities <- c(0.28, 0.21, 0.22, 0.29)
```

Then we can use the function "generateSeqWithMultinomialModel()" to calculate a 20-bp random sequence, using the vector *myprobabilities* as its input:

```
> generateSeqWithMultinomialModel(20, myprobabilities)
  [1] "C" "C" "G" "A" "T" "A" "T" "C" "C" "G" "C" "C" "T" "G" "A" "G" "T" "T" "T"
  [20] "C"
```

Q7.

How many protein sequences from rabies virus are there in the NCBI Protein database?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and select 'Protein' from the drop-down box above the search box.

Then type "rabies virus"[ORGN] in the search box, and press 'Search'.

On the results page, it should say "Results: 1 to 20 of 11768", meaning that there are 11768 protein sequences from rabies virus in the database [as of 16-Jun-2011]. Note that if you carry out this search at a later date, you may find more sequences, as the database is growing all the time.

Q8.

What is the NCBI accession for the Mokola virus genome?

To do this, you need to go to the www.ncbi.nlm.nih.gov website and select 'Genome' from the drop-down box above the search box.

Then type "Mokola virus"[ORGN] in the search box, and press 'Search'.

You should get a hit to accession NC_006429, the Mokola virus genome sequence.

Note that alternatively you can go to the www.ncbi.nlm.nih.gov website, and type "Mokola virus"[ORGN] in the search box, and press 'Search'. On the results page, you will see lots of hits to the Nucleotide and Protein databases, and 1 hit to the Genome database. If you click on the 1 hit beside "Genome", it will bring you to accession NC_006429, the Mokola virus genome sequence.

Revision Exercises 2

Q1.

Use the dotPlot() function in the SeqinR R package to make a dotplot of the rabies virus phosphoprotein and Mokkola virus phosphoprotein, using a windowsize of 10 and threshold of 5.

First we need to retrieve the rabies virus phosphoprotein (UniProt P06747) and Mokola virus phosphoprotein (UniProt P0C569) sequences from UniProt, which we can do using SeqinR:

```
> library("seqinr")                                    # load the SeqinR package
> choosebank("swissprot")                              # select the ACNUC sub-
↪database to search
> query("rabies", "AC=P06747")                         # specify the query
> rabiesseq <- getSequence(rabies$req[[1]])            # get the sequence
> query("mokola", "AC=P0C569")                         # specify the query
> mokolaseq <- getSequence(mokola$req[[1]])            # get the sequence
> closebank()                                          # close the connection to␣
↪the ACNUC sub-database
```

If you look at the help page of the dotPlot function (by typing "help(dotPlot)"), you will see that the windowsize can be specified using the "wsize" argument and the threshold can be specified using the "nmatch" argument.

We can therefore use dotPlot() to make a dotplot of the two proteins, using a windowsize of 10 and a threshold of 5, by typing:

```
> dotPlot(mokolaseq,rabiesseq,wsize=10,nmatch=5)
```
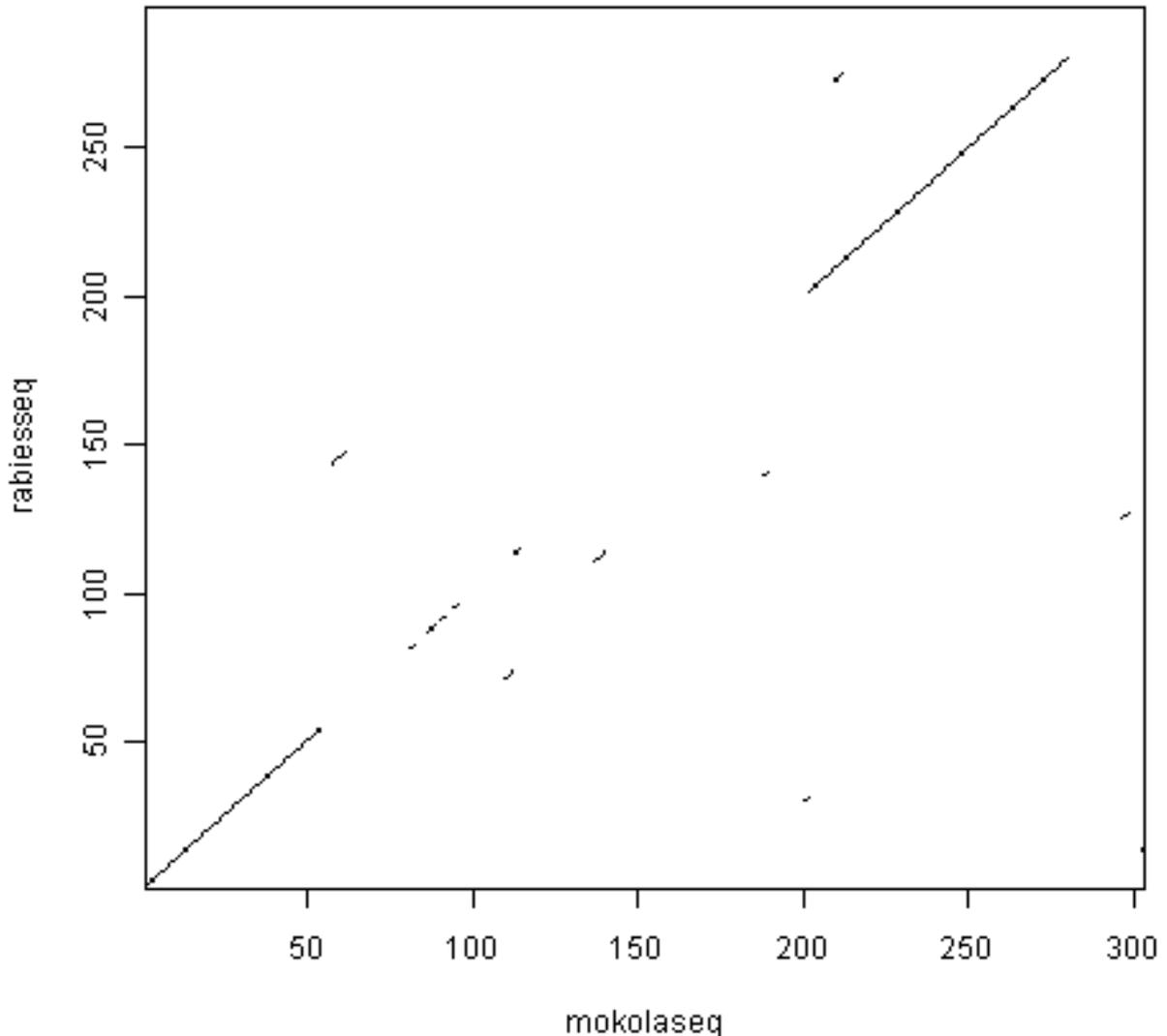

You can see that there is a region of similarity that covers about 60-70 amino acids at the start of the two proteins, then there is a region of similarity from about 210-280 in each of the two proteins. There is also a weak amount of similarity in a region from about 85-100 in the two proteins.
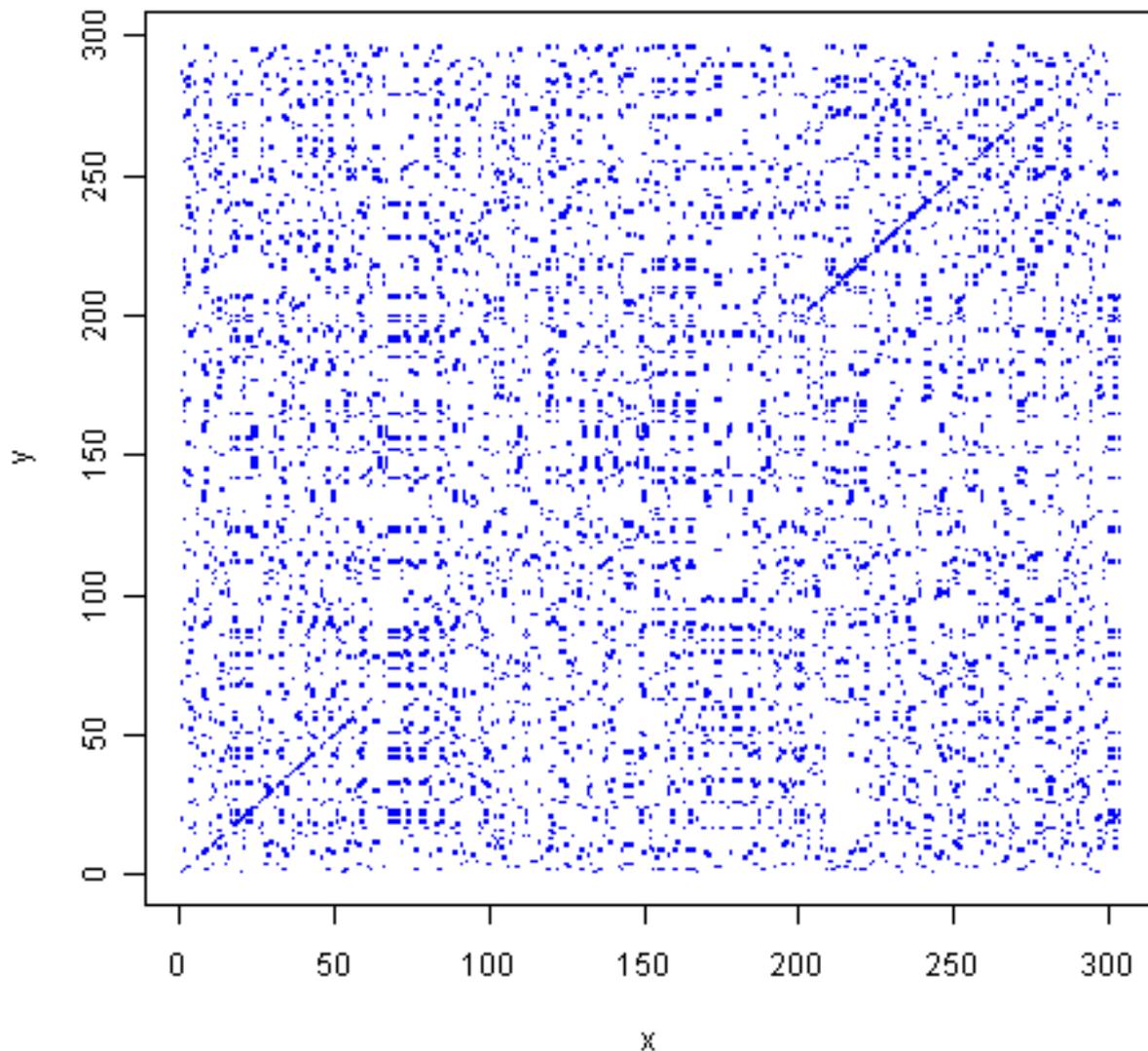
Q2.

Use the function makeDotPlot1 to make a dotplot of the rabies virus phosphoprotein and the Mokola virus phosphoprotein, setting the argument "dotsize" to 0.1.

To use the function makeDotPlot1(), we first need to copy and paste it into R.

We can then use it to make a dotplot, setting "dotsize" to 0.1, by typing:

```
> makeDotPlot1(mokolaseq,rabiesseq,dotsize=0.1)
```


As in Q1, you can see that there is a region of similarity that covers about 60-70 amino acids at the start of the two proteins, then there is a region of similarity from about 210-280 in each of the two proteins.

There are a lot of off-diagonal dots in this picture, because a dot is plotted at every position where the two sequences are identical in one letter (while in Q1, we only plotted a dot at the start of a 10-letter window, where 5 or more out of 10 positions in the window were identical).

The fact that there are so many dots in the picture makes it hard to see the weak region of similarity seen in Q1, from about 85-100 in the two proteins.

Q3.

Adapt the R code in Q2 to write a function that makes a dotplot using a window of size x letters, where a dot is plotted in the first cell of the window if y or more letters compared in that window are identical in the two sequences.

Here is an R function that will do this:

```
> makeDotPlot3 <- function(seq1,seq2,windowsize,threshold,dotsize=1)
  {
      length1 <- length(seq1)
```

```
    length2 <- length(seq2)
    # make a plot:
    x <- 1
    y <- 1
    plot(x,y,ylim=c(1,length2),xlim=c(1,length1),col="white")
    for (i in 1:(length1-windowsize+1))
    {
       word1 <- seq1[i:(i+windowsize)]
       word1b <- c2s(word1)
       for (j in 1:(length2-windowsize+1))
       {
          word2 <- seq2[j:(j+windowsize)]
          word2b <- c2s(word2)
          # count how many identities there are:
          identities <- 0
          for (k in 1:windowsize)
          {
             letter1 <- seq1[(i+k-1)]
             letter2 <- seq2[(j+k-1)]
             if (letter1 == letter2)
             {
                identities <- identities + 1
             }
          }
          if (identities >= threshold)
          {
             # add a point to the plot at the position
             for (k in 1:1)
             {
                points(x=(i+k-1),(y=j+k-1),cex=dotsize,col="blue",pch=7)
             }
          }
       }
    }
    print(paste("FINISHED NOW"))
}
```

Q4.

Use the dotPlot() function in the SeqinR R package to make a dotplot of rabies virus phosphoprotein and Mokola virus phosphoprotein, using a window size of 3 and a threshold of 3. Use your own R function from Q3 to make a dotplot of rabies virus phosphoprotein and Mokola virus phosphoprotein, using a windowsize (x) of 3 and a threshold (y) of 3. Are the two plots similar or different, and can you explain why?

We can use the dotPlot() function from SeqinR to make a dotplot of the rabies and Mokola virus phosphoproteins, using a window size of 3 and a threshold of 3, by typing:

```
> dotPlot(mokolaseq,rabiesseq,wsize=3,nmatch=3)
```
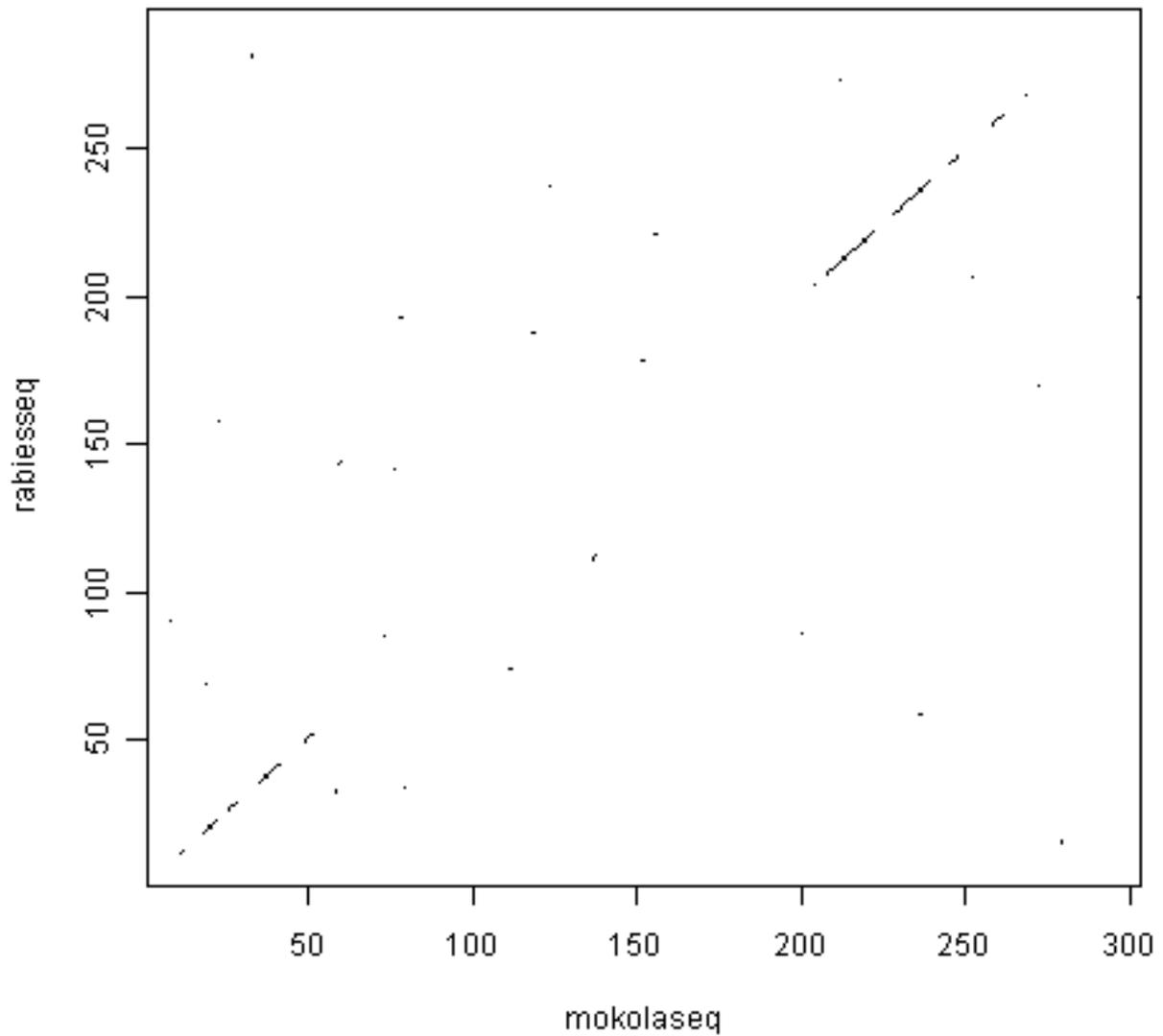
We can also use our function makeDotPlot3 to make a dotplot of the rabies and Mokola virus proteins, using a window size of 3 and a threshold of 3:

```
> makeDotPlot3(mokolaseq,rabiesseq,windowsize=3,threshold=3,dotsize=0.1)
```
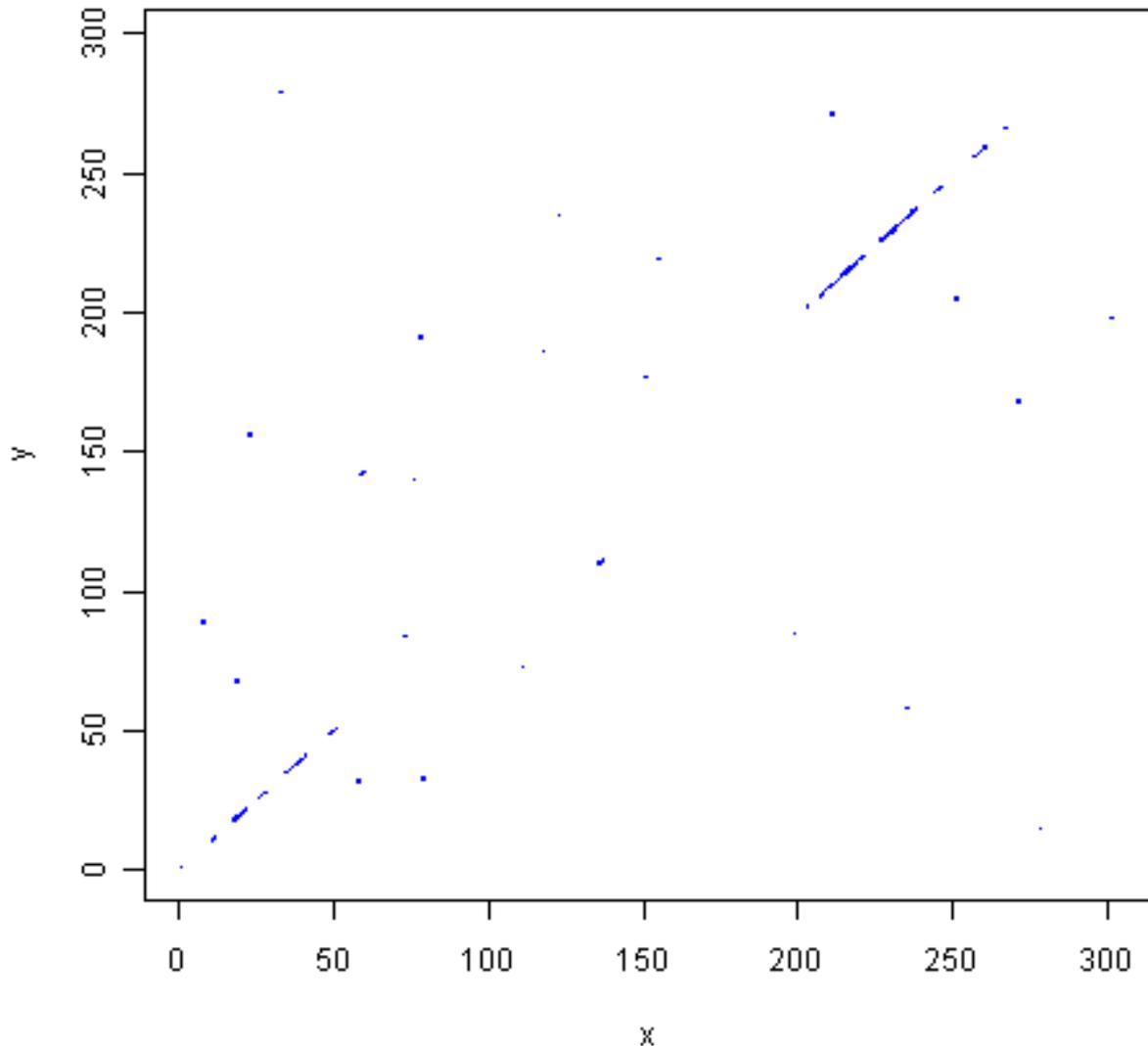
The two pictures are the same, as they should be, as both are plotting a dot in the first position of a 3-letter window if all 3 letters in that window are identical in the two sequences.

Q5.

Write an R function to calculate an unrooted phylogenetic tree with bootstraps, using the minimum evolution method.

We can adjust the function unrootedNJtree, which uses the neighbour-joining method, as it calls the function "nj()" to build a tree.

You can search for R functions that build a tree using minimum evolution method by typing:

```
> help.search("evolution")
  ape::fastme                 Tree Estimation Based on the Minimum Evolution
                              Algorithm
```

We find that there is a function "fastme()" in the Ape package to build a tree using the minimum evolution method.

You can view the help page for this function by typing 'help("fastme")'. If you do this, you will see that it can be run by typing fastme.bal() or fastme.ols(), which are two different versions of the minimum evolution function.

Thus, we can adapt the unrootedNJtree to make a function that builds a tree using minimum evolution, by using "fastme.bal()" instead of "nj()":

```
> unrootedMEtree <- function(alignment,type)
  {
     # load the ape and seqinR packages:
     require("ape")
     require("seqinr")
     # define a function for making a tree:
     makemytree <- function(alignmentmat)
     {
        alignment <- ape::as.alignment(alignmentmat)
        if       (type == "protein")
        {
           mydist <- dist.alignment(alignment)
        }
        else if (type == "DNA")
        {
           alignmentbin <- as.DNAbin(alignment)
           mydist <- dist.dna(alignmentbin)
        }
        mytree <- fastme.bal(mydist)
        mytree <- makeLabel(mytree, space="") # get rid of spaces in tip names.
        return(mytree)
     }
     # infer a tree
     mymat  <- as.matrix.alignment(alignment)
     mytree <- makemytree(mymat)
     # bootstrap the tree
     myboot <- boot.phylo(mytree, mymat, makemytree)
     # plot the tree:
     plot.phylo(mytree,type="u")    # plot the unrooted phylogenetic tree
     nodelabels(myboot,cex=0.7)     # plot the bootstrap values
     mytree$node.label <- myboot    # make the bootstrap values be the node labels
     return(mytree)
  }
```

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

Protein-Protein Interaction Graphs

More about R

In previous practicals you have have used the R statistics software for analysing many different types of data. In this practical, you will use R for analysing protein-protein interaction data. However, first we will discuss some features of R that will be useful in this practical.

One thing that is useful to know about R is that many R packages come with example data sets, which can be used to familiarise yourself with the functions in the particular package. To list the data sets that come with a particular package, you can use the data() function in R. For example, to find the data sets that come with the "graph" package, type:

```
> library("graph")
> data(package="graph")
Data sets in package 'graph':

IMCAAttrs (integrinMediatedCellAdhesion)
                                KEGG Integrin Mediated Cell Adhesion graph
IMCAGraph (integrinMediatedCellAdhesion)
                                KEGG Integrin Mediated Cell Adhesion graph
MAPKsig                         A graph encoding parts of the MAPK signaling pathway
MAPKsig (defunctGraph)          A graph encoding parts of the MAPK signaling pathway
apopGraph                       KEGG apoptosis pathway graph
biocRepos                       A graph representing the Bioconductor package␣
→repository
graphExamples                   A List Of Example Graphs
pancrCaIni                      A graph encoding parts of the pancreatic cancer␣
→initiation
```

You can then load any of these data sets into R by typing, for example, to load the apopGraph data set:

```
> data("apopGraph")
```

In this practical, you will also be using the table() function to make tables of data stored in vectors. If you have a vector containing numeric values, the table() function is useful for making a table saying how many elements in the vector have each of the values. For example:

```
> y <- c(10, 10, 20, 20, 20, 20, 20, 30) # Make a numeric vector "y"
> table(y)
y
10 20 30
 2  5  1
```

The results from the table() function tell us that two of the elements in vector *y* have values of 10, five elements have values of 20, and one element has a value of thirty.

Another use of the table() function is to tell us how many elements in a vector of numbers have a particular numeric value. For example, if we want to know how many elements in vector *y* have a value of 20, we can type:

```
> table(y == 20)
FALSE  TRUE
    3     5
```

This tells us that five elements of vector *y* have values of 20.

In this practical you will be using functions from several different packages. It's important to remember that sometimes functions in different packages have the same name. For example, there is a function called degree() in both the "igraph" and "graph" packages. Therefore, you need to specify which degree() function you want to use, by putting the package name, followed by ":", before the name of the function. For example, to use the degree() function from the "graph" package, you can type graph::degree(), while to use the degree() function from the "igraph" package, you can type igraph::degree().

Graphs for protein-protein interaction data in R

Protein-protein interaction data can be described in terms of graphs. In this practical, we will explore a curated data set of protein-protein interactions, by using R packages for analysing and visualising graphs.

We will use three main R packages that have been written for handling biological graphs: the "graph" package, the "RBGL" package, and the "Rgraphviz" package. The Rgraphviz package is part of the Bioconductor set of R packages, so needs to be installed as part of that set of package (see the Bioconductor webpage at www.bioconductor.org/docs/install/ for details).

We will first analyse a curated data set of protein-protein interactions in the yeast *Saccharomyces cerevisiae* extracted from published papers. This data set comes from with an R package called "yeastExpData", which

calls the data set "litG". This data was first described in a paper by Ge *et al* (2001) in *Nature Genetics* (http://www.nature.com/ng/journal/v29/n4/full/ng776.html).

To read the litG data set into R, we first need to load the yeastExpData package, and then we can use the R data() function to read in the litG data set:

```
> library("yeastExpData") # Load the yeastExpData package
> data("litG")            # Read the litG data set
```

When you read in the litG data set using the data() function, it is stored as a graph in R. In this graph, the vertices (nodes) are proteins, and edges between vertices indicate that two proteins interact. There are 2885 different vertices in the graph, representing 2885 different proteins.

You can print out the number of vertices and edges in a graph in R by just typing the name of the graph, for example:

```
> litG
A graphNEL graph with undirected edges
Number of Nodes = 2885
Number of Edges = 315
```

This tells us that the litG graph has 2885 vertices, and 315 edges. The 315 edges in the graph represent 315 protein-protein interactions between 315 pairs of proteins.

Finding the names of vertices in graphs for protein-protein interaction data in R

The "graph" R package contains many functions for analysing graph data in R. For example, the nodes() function from the graph package can be used to retrieve the names of the vertices (nodes) in the graph. For example, we can retrive the names of the vertices in the litG graph, and store them in a vector "mynodes", by typing:

```
> library("graph")          # Load the graph package
> mynodes <- nodes(litG)  # Retrieve the names of the vertices in the litG graph
```

We can then print the names of the first 10 vertices in the litG graph, by typing:

```
> mynodes[1:10]               # Print the names of the first 10 vertices in the litG
↪graph
 [1] "YBL072C" "YBL083C" "YBR009C" "YBR010W" "YBR031W" "YBR093C" "YBR106W"
 [8] "YBR118W" "YBR188C" "YBR191W"
```

This gives the names of the yeast proteins corresponding to the first 10 vertices in the litG graph. Note that the order that the proteins are stored in the graph does not have any meaning; these 10 proteins just happen to be the first 10 stored in the litG graph. As *mynodes* is a vector that contains one element for each vertex in the litG graph, the number of elements *mynodes* should be equal to the number of vertices in the litG graph:

```
> length(mynodes)             # Find the number of vertices in the litG graph
[1] 2885
```

As expected, we find that the litG graph contains 2885 vertices, which represent 2885 different yeast proteins.

Finding the names of proteins that a particular protein interacts with

If you are particularly interested in a particular protein in a protein-protein interaction graph, you may want to print out the list of the proteins that that protein interacts with. To do this, you can use the adj() function in the R "graph" package. For example, to print out the proteins that yeast protein YBR009C interacts with in the litG graph, you can type:

```
> adj(litG, "YBR009C")
$YBR009C
[1] "YBR010W" "YNL031C" "YDR227W"
```

This tells us that protein YBR009C interacts with three other protein in the litG graph, that is, YBR010W, YNL031C and YDR227W.

Calculating the degree distribution for a graph in R

The *degree* of a vertex (node) in a graph is the number of connections that it has to other vertices in the graph.

The *degree distribution* for a graph is the distribution of degree values for all the vertices in the graph, that is, the number of vertices in the graph that have degrees of 0, 1, 2, 3, etc.

In terms of a protein-protein interaction graph, each vertex in the graph represents a protein, and the degree of a particular vertex is the number of interactions that that protein has with other proteins.

You can calculate the degrees of all the vertices in a graph by using the degree() function in the R "graph" package. The degree() function returns a vector containing the degrees of each of the vertices in the graph. Remember that there is a degree() function in both the "graph" and "igraph" packages, so if you have loaded both packages, you will need to specify that you want to use the degree() function in the "graph" package, by writing graph::degree().

For example, to calculate the degrees of vertices in the litG graph, we type:

```
> mydegrees <- graph::degree(litG)
> mydegrees # Print out the values in the vector "mydegrees"
  YBL072C   YBL083C   YBR009C   YBR010W   YBR031W   YBR093C   YBR106W   YBR118W
        0         0         3         3         0         0         0         2
```

For example, we see from the above results that the yeast protein YBL072C does not interact with any other protein, while the yeast protein YBR118W interacts with two other yeast proteins. Only the first line of the results is shown, as there are 2885 vertices in the litG graph.

You can sort the vector *mydegrees* in order of the number of degrees, by using the sort() function:

```
> sort(mydegrees)
  YBL072C   YBL083C   YBR031W   YBR093C   YBR106W   YBR188C   YBR191W   YBR206W   ↵
→YCL007C   YCL018W
        0         0         0         0         0         0         0         0   ↵
→       0         0
...
  YBR198C   YDR392W   YDR448W   YBR160W   YFL039C
        8         8         9        10        12
```

Only the first and last lines of the output are shown above. You can see from the last line of the output that there are some vertices that have high degrees. For example, the vertex corresponding to the protein YFL039C is 12. This means that the protein YFL039C interacts with 12 other proteins. Such highly connected proteins in a protein-protein interaction graph are sometimes called *hub proteins*.

We can calculate the *degree distribution* for a graph by using the table() function to make a table of how many vertices in the graph have degrees of 0, 1, 2, 3, etc. For example, to calculate the degree distribution for the litG graph, you can type:

```
> table(mydegrees)
mydegrees
   0    1    2    3    4    5    6    7    8    9   10   12
2587  159   58   38   19    7    3    7    4    1    1    1
```

This tells us that 2587 vertices in the litG graph are not connected to any other vertices, 159 vertices are connected to one other vertex, 58 vertices are connected to two other vertices, and so on. You can calculate the mean degree of the vertices using the mean() function in R:

```
> mean(mydegrees)
[1] 0.2183709
```

The mean degree is only about 0.22 for the litG graph, as most of the proteins do not interact with any other protein.

It is nice to visualise the degree distribution for a graph by plotting it as a histogram (using the hist() R function):

```
> hist(mydegrees, col="red")
```

src/../../_static/MB6300_P1_image5.png
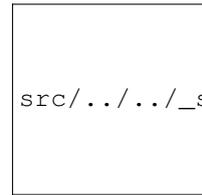
Finding connected components in graphs for protein-protein interaction data in R

If you are analysing a very large graph, it may contain several subgraphs, where the vertices within each subgraph are connected to each other by edges, but there are no edges connecting the vertices in different subgraphs. In this case, the subgraphs are known as *connected components* (also called *maximally connected subgraphs*).

For example, the graph below contains three connected components:
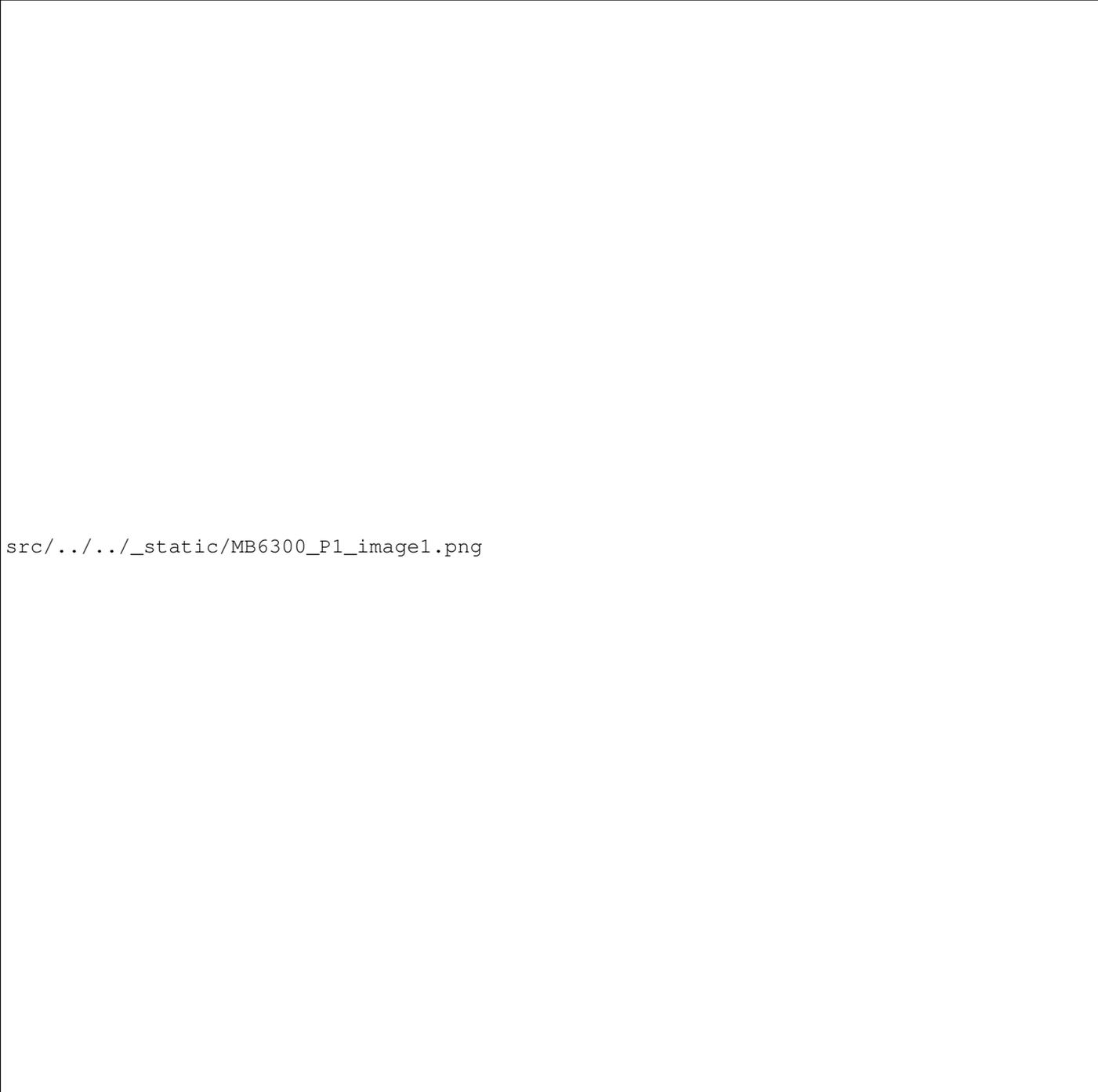
Image source: http://en.wikipedia.org/wiki/Connected_component_(graph_theory)

You can find connected components of a graph in R, by using the connectedComp function in the "RBGL" package. For example, to find connected components in the litG graph, we type:

```
> library("RBGL")
> myconnectedcomponents <- connectedComp(litG)
```

The commands above store the connected components in the litG graph in a list *myconnectedcomponents*. Each connected component is stored in one element of the list variable *myconnectedcomponents*. That is, each element of the list *myconnectedcomponents* is a vector containing the names of the proteins in a particular connected component.

We can print out the yeast proteins that are the vertices of the first three connected components by printing out the first three elements in the list *myconnectedcomponents*. Remember that you need to use double square brackets to

access the elements of a list variable in R:

```
> myconnectedcomponents[[1]]
[1] "YBL072C"
> myconnectedcomponents[[2]]
[1] "YBL083C"
> myconnectedcomponents[[3]]
 [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W"
 [8] "YER095W" "YJL173C" "YNL312W" "YBL084C" "YDR146C" "YLR127C" "YNL172W"
[15] "YLR134W" "YMR284W" "YER179W" "YIL144W" "YML104C" "YOR191W" "YDL008W"
[22] "YDL030W" "YDL042C" "YDR004W" "YGR162W" "YMR117C" "YDR386W" "YDR485C"
[29] "YDL043C" "YDR118W" "YMR106C" "YML032C" "YDR076W" "YDR180W" "YDL013W"
[36] "YDR227W"
```

That is, the first two connected components only contain one protein each. These two proteins must not have interactions with any of the other yeast proteins in the litG graph. The third connected component contains 36 proteins. These 36 proteins are not necessarily all connected to each other, but each of the 36 proteins must be connected to at least one of the other 35 proteins in the connected component. Note that the connected components are not stored in the list *myconnectedcomponents* in any particular order; these just happen to be the first three connected components stored in the list.

To find the total number of connected components in the litG graph, we can just find the length of the list *myconnectedcomponents*:

```
> length(myconnectedcomponents)
[1] 2642
```

That is, there are 2642 different connected components in the litG graph. These are 2642 subgraphs of the graph, where there are edges between the vertices within a subgraph, but no edges between the 2642 subgraphs.

It is interesting to know what is the largest connected component in a graph. How can we calculate this for the litG graph? Well, each element of the litG graph contains a vector storing the proteins in a particular connected component, and the length of this vector is the number of proteins in that connected component. Thus, to calculate the sizes of all connected components in the litG graph, we can use a "for loop" to calculate the length of each of the vectors in *myconnectedcomponents* in turn:

```
> componentsizes <- numeric(2642) # Make a vector for storing the sizes of the
↪2642 connected components
> for (i in 1:2642) {
    component <- myconnectedcomponents[[i]] # Store the connected component in a
↪vector "component"
    componentsize <- length(component)      # Find the number of vertices in this
↪connected component
    componentsizes[i] <- componentsize      # Store the size of this component
}
```

In the code above, the line componentsizes <- numeric(2642) makes a new vector *componentsizes* which has the same number of elements as the number of connected components in the litG graph (2642). This vector *componentsizes* is then used within the for loop for storing the size of each connected component. We can now find the size of the largest connected component in the litG graph by using the max() R function to find the largest value in the vector *componentsizes*:

```
> max(componentsizes)
[1] 88
```

That is, the largest connected component in the litG graph has 88 different proteins.

We can also use the table() function in R to make a table of the number of connected components of different sizes:

```
> table(componentsizes)
componentsizes
```

```
     1     2     3     4     5     6     7     8    12    13    36    88
  2587    29    10     7     1     1     2     1     1     1     1     1
```

This tells us that there is just one connected component with 88 proteins. Furthermore, we see that there are 2587 connected components that contain just 1 protein each. These proteins presumably do not have any known interactions with with any other protein in the litG data set.

To find the connected component that a particular protein belongs to, you can use the findcomponent function:

```r
> findcomponent <- function(graph,vertex)
  {
     # Function to find the connected component that contains a particular vertex
     require("RBGL")
     found <- 0
     myconnectedcomponents <- connectedComp(graph)
     numconnectedcomponents <- length(myconnectedcomponents)
     for (i in 1:numconnectedcomponents)
     {
        componenti <- myconnectedcomponents[[i]]
        numvertices <- length(componenti)
        for (j in 1:numvertices)
        {
           vertexj <- componenti[j]
           if (vertexj == vertex)
           {
              found <- 1
              return(componenti)
           }
        }
     }
     print("ERROR: did not find vertex in the graph")
  }
```

The function findcompontent() returns a vector containing the names of the proteins in the connected component. For example, to find the connected component containing the protein YBR009C, you can type:

```r
> mycomponent <- findcomponent(litG, "YBR009C")
> mycomponent # Print out the members of this connected component.
  [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W"
↪"YER095W" "YJL173C" "YNL312W"
 [11] "YBL084C" "YDR146C" "YLR127C" "YNL172W" "YLR134W" "YMR284W" "YER179W"
↪"YIL144W" "YML104C" "YOR191W"
 [21] "YDL008W" "YDL030W" "YDL042C" "YDR004W" "YGR162W" "YMR117C" "YDR386W"
↪"YDR485C" "YDL043C" "YDR118W"
 [31] "YMR106C" "YML032C" "YDR076W" "YDR180W" "YDL013W" "YDR227W"
```

Extracting a subgraph from a graph in R

If you want to extract a particular subgraph of a graph (that is, part of a graph), you can use the subGraph function in the "graph" package. As its arguments (inputs), the subGraph function contains a vector containing the vertices (nodes) in the subgraph that we're interested in, and the graph that the subgraph belongs to.

For example, if we want to extract the subgraph (of graph litG) that contains the third connected component in the vector *myconnectedcomponents*, we type:

```r
> myconnectedcomponents <- connectedComp(litG)
> component3 <- myconnectedcomponents[[3]]
> component3 # Print out the proteins in connected component 3
 [1] "YBR009C" "YBR010W" "YNL030W" "YNL031C" "YOL139C" "YAR007C" "YBR073W"
 [8] "YER095W" "YJL173C" "YNL312W" "YBL084C" "YDR146C" "YLR127C" "YNL172W"
[15] "YLR134W" "YMR284W" "YER179W" "YIL144W" "YML104C" "YOR191W" "YDL008W"
```

```
[22] "YDL030W" "YDL042C" "YDR004W" "YGR162W" "YMR117C" "YDR386W" "YDR485C"
[29] "YDL043C" "YDR118W" "YMR106C" "YML032C" "YDR076W" "YDR180W" "YDL013W"
[36] "YDR227W"
> mysubgraph <- subGraph(component3, litG)
> mysubgraph
A graphNEL graph with undirected edges
Number of Nodes = 36
Number of Edges = 48
```

The commands above store the subgraph corresponding to *component3* in a graph object *mysubgraph* that contains 36 vertices and 48 edges.

Plotting graphs for protein-protein interaction data in R

The "Rgraphviz" R package contains useful functions for plotting graphs, or plotting parts of graphs ("subgraphs").

The layoutGraph and renderGraph functions in the Rgraphviz package can be used to make a nice plot of a subgraph. There are lots of options for the colours to use for plotting vertices and edges.

For example, if we want to make a plot of the subgraph corresponding to the third connected component in the vector *myconnectedcomponents*, we can type:

```
> library("Rgraphviz")
> mysubgraph <- subGraph(component3, litG)
> mygraphplot <- layoutGraph(mysubgraph, layoutType="neato")
> renderGraph(mygraphplot)
```

src/../../_static/MB6300_P1_image2.png

The plot above shows a plot of the third connected component in the graph litG. There are 36 vertices in this subgraph, corresponding to 36 different yeast proteins. The names of the proteins are shown in the circles that represent the vertices. The edges between vertices represent interactions between pairs of proteins.

Detecting communities in a protein-protein interaction graph using R

A property common to many types of graphs, including protein-protein interaction graphs, is *community structure*. A *community* is often defined as a subset of the vertices in the graph such that connections btween the vertices are denser than connections with the rest of the graph. For example, the graph in the picture below consists of one connected component. However, within that connected component, we can see three densely connected subgraphs; these could be said to be three different *communities* within the graph:

src/../../_static/MB6300_P1_image7.png

Image source: http://en.wikipedia.org/wiki/Community_structure

In terms of protein-protein interaction networks, if there are several communities within a connected component (for example, three communities, as in the picture above), these could represent three different groups of proteins, where the proteins within one community interact much more with each other than with proteins in the other communities.

By detecting communities within a protein-protein interaction graph, we can detect putative *protein complexes*, that is, groups of associated proteins that are probably fairly stable over time. In other words, protein complexes can be detected by looking for groups of proteins among which there are many interactions, and where the members of the complex have few interactions with other proteins that do not belong to the complex.

There are lots of different methods available for detecting communities in a graph, and each method will give slightly different results. That is, the particular method used for detecting communities will decide how you split a connected component into one or more communities.

The function findcommunities() below identifies communities within a graph (or subgraph of a graph). It requires a second function, findcommunities2(), which is also below:

```r
> findcommunities <- function(mygraph,minsize)
  {
     # Function to find network communities in a graph
     # Load up the igraph library:
     require("igraph")
     # Set the counter for the number of communities:
     cnt <- 0
     # First find the connected components in the graph:
     myconnectedcomponents <- connectedComp(mygraph)
     # For each connected component, find the communities within that connected
→component:
     numconnectedcomponents <- length(myconnectedcomponents)
     for (i in 1:numconnectedcomponents)
     {
        component <- myconnectedcomponents[[i]]
        # Find the number of nodes in this connected component:
        numnodes <- length(component)
        if (numnodes > 1) # We can only find communities if there is more than one
→node
        {
           mysubgraph <- subGraph(component, mygraph)
           # Find the communities within this connected component:
           # print(component)
           myvector <- vector()
           mylist <- findcommunities2(mysubgraph,cnt,"FALSE",myvector,minsize)
           cnt <- mylist[[1]]
           myvector <- mylist[[2]]
        }
     }
     print(paste("There were",cnt,"communities in the input graph"))
  }
> findcommunities2 <- function(mygraph,cnt,plot,myvector,minsize)
  {
     # Function to find network communities in a connected component of a graph
     # Find the number of nodes in the input graph
     nodes <- nodes(mygraph)
     numnodes <- length(nodes)
     # Record the vertex number for each vertex name
     myvector <- vector()
     for (i in 1:numnodes)
     {
        node <- nodes[i] # "node" is the vertex name, i is the vertex number
        myvector[`node`] <- i  # Add named element to myvector
     }
     # Create a graph in the "igraph" library format, with numnodes nodes:
     newgraph <- graph.empty(n=numnodes,directed=FALSE)
     # First record which edges we have seen already in the "mymatrix" matrix,
     # so that we don't add any edge twice:
     mymatrix <- matrix(nrow=numnodes,ncol=numnodes)
     for (i in 1:numnodes)
     {
```

```
         for (j in 1:numnodes)
         {
            mymatrix[i,j] = 0
            mymatrix[j,i] = 0
         }
      }
      # Now add edges to the graph "newgraph":
      for (i in 1:numnodes)
      {
         node <- nodes[i] # "node" is the vertex name, i is the vertex number
         # Find the nodes that this node is joined to:
         neighbours <- adj(mygraph, node)
         neighbours <- neighbours[[1]] # Get the list of neighbours
         numneighbours <- length(neighbours)
         if (numneighbours >= 1) # If this node "node" has some edges to other nodes
         {
            for (j in 1:numneighbours)
            {
               neighbour <- neighbours[j]
               # Get the vertex number
               neighbourindex <- myvector[neighbour]
               neighbourindex <- neighbourindex[[1]]
               # Add a node in the new graph "newgraph" between vertices i and␣
→neighbourindex
               # In graph "newgraph", the vertices are counted from 0 upwards.
               indexi <- i
               indexj <- neighbourindex
               # If we have not seen this edge already:
               if (mymatrix[indexi,indexj] == 0 && mymatrix[indexj,indexi] == 0)
               {
                  mymatrix[indexi,indexj] <- 1
                  mymatrix[indexj,indexi] <- 1
                  # Add edges to the graph "newgraph"
                  newgraph <- add.edges(newgraph, c(i, neighbourindex))
               }
            }
         }
      }
      # Set the names of the vertices in graph "newgraph":
      newgraph <- set.vertex.attribute(newgraph, "name", value=nodes)
      # Now find communities in the graph:
      communities <- spinglass.community(newgraph)
      # Find how many communities there are:
      sizecommunities <- communities$csize
      numcommunities <- length(sizecommunities)
      # Find which vertices belong to which communities:
      membership <- communities$membership
      # Get the names of vertices in the graph "newgraph":
      vertexnames <- get.vertex.attribute(newgraph, "name")
      # Print out the vertices belonging to each community:
      for (i in 1:numcommunities)
      {
         cnt <- cnt + 1
         nummembers <- 0
         printout <- paste("Community",cnt,":")
         for (j in 1:length(membership))
         {
            community <- membership[j]
            if (community == i) # If vertex j belongs to the ith community
            {
               vertexname <- vertexnames[j]
               if (plot == FALSE)
               {
```

```
                nummembers <- nummembers + 1
                # Print out the vertices belonging to the community
                printout <- paste(printout,vertexname)
             }
             else
             {
                # Colour in the vertices belonging to the community
                myvector[`vertexname`] <- cnt
             }
          }
       }
       if (plot == FALSE && nummembers >= minsize)
       {
          print(printout)
       }
    }
    return(list(cnt,myvector))
 }
```

The function findcommunities() uses the function spinglass.community() from the "igraph" package to identify communities in a graph or subgraph. As its arguments (inputs), the findcommunities() function takes the graph/subgraph that we want to find communities in, and the minimum number of vertices that a community must have to be reported.

For example, to find communities within the subgraph corresponding to the third connected component of the litG graph, we can type:

```
> mysubgraph <- subGraph(component3, litG)
> findcommunities(mysubgraph, 1)
[1] "Community 1 : YML104C YOR191W YDL030W YDR485C YDL013W"
[1] "Community 2 : YBR073W YDR146C YLR134W YER179W YIL144W"
[1] "Community 3 : YOL139C YGR162W YMR117C YDR386W YDL043C YDR180W"
[1] "Community 4 : YBL084C YLR127C YNL172W YDL008W YDR118W"
[1] "Community 5 : YAR007C YER095W YJL173C YNL312W YDR004W YML032C YDR076W"
[1] "Community 6 : YBR009C YBR010W YNL030W YNL031C YMR284W YDL042C YMR106C YDR227W"
[1] "There were 6 communities in the input graph"
```

This tells us that there are six different communities in the subgraph corresponding to the third connected component of the litG graph.

Note that if you run findcommunities() again and again on the same input graph, it might find slightly different sets of communities each time. This is because it uses a random number generator in the method that it uses for identifying communities, and the random number used will be different each time you run the findcommunities() function, which means that you will get slightly different answers each time. The answers should be very similar, however, but you might see a small difference, for example, a large community might be split into two smaller communities.

You can make a plot of the communities in a graph or subgraph by using the plotcommunities() function:

```
> plotcommunities <- function(mygraph)
  {
     # Function to plot network communities in a graph
     # Load the "igraph" package:
     require("igraph")
     # Make a plot of the graph
     graphplot <- layoutGraph(mygraph, layoutType="neato")
     renderGraph(graphplot)
     # Get the names of the nodes in the graph:
     vertices <- nodes(mygraph)
     numvertices <- length(vertices)
     # Now record the colour of each vertex in a vector "myvector":
     myvector <- vector()
     colour <- "red"
```

```
    for (i in 1:numvertices)
    {
       vertex <- vertices[i]
       myvector[`vertex`] <- colour   # Add named element to myvector
    }
    # Set the counter for the number of communities:
    cnt <- 0
    # First find the connected components in the graph:
    myconnectedcomponents <- connectedComp(mygraph)
    # For each connected component, find the communities within that connected␣
↪component:
    numconnectedcomponents <- length(myconnectedcomponents)
    for (i in 1:numconnectedcomponents)
    {
       component <- myconnectedcomponents[[i]]
       # Find the number of nodes in this connected component:
       numnodes <- length(component)
       if (numnodes > 1) # We can only find communities if there is more than one␣
↪node
       {
          mysubgraph <- subGraph(component, mygraph)
          # Find the communities within this connected component:
          mylist <- findcommunities2(mysubgraph,cnt,"TRUE",myvector,0)
          cnt <- mylist[[1]]
          myvector <- mylist[[2]]
       }
    }
    # Get a set of cnt colours, where cnt is equal to the number of communities␣
↪found:
    mycolours <- rainbow(cnt)
    # Set the colour of the vertices, so that vertices in each community are of␣
↪the same colour,
    # and vertices in different communities are different colours:
    myvector2 <- vector()
    for (i in 1:numvertices)
    {
       vertex <- vertices[i]
       community <- myvector[vertex]
       mycolour <- mycolours[community]
       myvector2[`vertex`] <- mycolour
    }
    nodeRenderInfo(graphplot) = list(fill=myvector2)
    renderGraph(graphplot)
 }
```
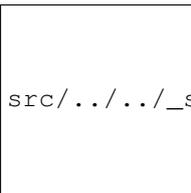
For example, to make a plot of the communities in the third connected component of the litG graph using the plotcommunities() function, you need to type:

```
> mysubgraph <- subGraph(component3, litG)
> plotcommunities(mysubgraph)
```


src/../../_static/MB6300_P1_image8.png

In the graph above, the six communities in the third connected component of the litG graph are coloured with six different colours.

Reading in protein-protein interaction data in R

In the above example, you looked at the litG data set of protein-protein interactions, which is a data set that comes with the "yeastExpData" R package. But what if you want to look at a data set of protein-protein interactions that does not come from R?

It is common to store data on protein-protein interactions in a text file with two columns, where each line of the file contains a pair of proteins that interact with each other. For example, such a file may look like this: YKL166C YIL033C YCR002C YHR107C YCR002C YJR076C YCR002C YLR314C YJR076C YHR107C This indicates that there are 5 protein-protein interactions, between protein YKL166C and protein YIL033C, between YCR002C and YHR107C, between YCR002C and YJR076C, between YCR002C and YLR314C, and between YJR076C and YHR107C.

The function makeproteingraph() makes a graph based on an input file of protein-protein interactions, where the first two columns of the input file indiciate the pairs of proteins that interact:

```
> makeproteingraph <- function(myfile)
  {
     # Function to make a graph based on protein-protein interaction data in an
→input file
     require("graph")
     mytable <- read.table(file(myfile)) # Store the data in a data frame
     proteins1 <- mytable$V1
     proteins2 <- mytable$V2
     protnames <- c(levels(proteins1),levels(proteins2))
     # Find out how many pairs of proteins there are
     numpairs <- length(proteins1)
     # Find the unique protein names:
     uniquenames <- unique(protnames)
     # Make a graph for these proteins with no edges:
     mygraph <- new("graphNEL", nodes = uniquenames)
     # Add edges to the graph:
     # See http://rss.acs.unt.edu/Rdoc/library/graph/doc/graph.pdf for more
→examples
     weights <- rep(1,numpairs)
     mygraph2 <- addEdge(as.vector(proteins1),as.vector(proteins2),mygraph,weights)
     return(mygraph2)
  }
```

For example, the example file http://www.maths.tcd.ie/~avrillee/littlebookofr/ExampleInteractionData contains the five pairs of interacting proteins listed above. You can read it in and make a graph for these interacting proteins by typing:

```
> thegraph <- makeproteingraph("http://www.maths.tcd.ie/~avrillee/littlebookofr/
→ExampleInteractionData")
```

You can then make a plot of this graph as before:

```
> mygraphplot <- layoutGraph(thegraph, layoutType="neato")
> renderGraph(mygraphplot)
```

src/../../_static/MB6300_P1_image4.png
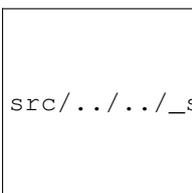
Creating random graphs in R

A *random graph* is a graph that is generated by a random process, where you start off with a certain number of vertices (nodes), and edges are added by randomly choosing pairs of vertices and making an edge between the two members of each of those pairs of vertices. (This is known as the *Erdös-Renyi* model for random graphs). In a random graph, vertices with lots of connections are equally likely as vertices with very few connections. That is, if you calculate the average degree of the vertices in a random graph, you will find that the degrees of most of the vertices in the graph is near to the average.

It is often useful and interesting to compare the properties of biological graphs to random graphs. In order to do this, you need to be able to generate some random graphs. The function makerandomgraph() in R makes a random graph with a certain number of edges:

```
> makerandomgraph <- function(numvertices,numedges)
  {
     # Function to make a random graph
     require("graph")
     # Make a vector with the names of the vertices
     mynames <- sapply(seq(1,numvertices),toString)
     myrandomgraph <- randomEGraph(mynames, edges = numedges)
     return(myrandomgraph)
  }
```

This function takes as its argument (input) the number of vertices and edges that you want the random graph to have to have. For example, to create a random graph that has 15 vertices and 43 edges, we type:

```
> myrandomgraph <- makerandomgraph(15, 43)
> myrandomgraph # Print out the number of vertices and edges in the graph
A graphNEL graph with undirected edges
Number of Nodes = 15
Number of Edges = 43
```

In the R code above, we tell R to give the vertices the labels 1 to 15. We can of course plot the random graph:

```
> myrandomgraphplot <- layoutGraph(myrandomgraph, layoutType="neato")
> renderGraph(myrandomgraphplot)
```

src/../../_static/MB6300_P1_image6.png
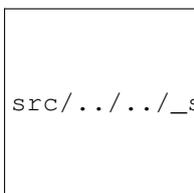
Summary

In this practical, you will have learnt to use the following R functions:

1. data() to load a data set that comes with a package into R

2. table() for making a table of the data in a vector, or finding out how many elements in a vector have a particular value

3. sort() for sorting a vector

All of these functions belong to the standard installation of R.

You have also learnt the following R functions that belong to the additional R packages:

1. nodes() from the "graph" package for getting a list of the names of vertices in a graph

2. adj() from the "graph" package for getting a list of the vertices that a particular vertex is connected to in a graph

3. degree() from the "graph" package for calculating the degree of each of the vertices in a graph

4. connectedComp() from the "RBGL" package for identifying connected components in a graph

5. subGraph() from the "graph" package for extracting a subgraph from a graph

6. layoutGraph() and renderGraph() from the "Rgraphviz" package for plotting a graph or subgraph

Links and Further Reading

Some links are included here for further reading.

For background reading on graphs and protein-protein interaction graphs, it is recommended to read Chapters 1, 2 and 4 of *Principles of Computational Cell Biology: from protein complexes to cellular networks* by Volkhard Helms (Wiley-VCH; http://www.wiley-vch.de/publish/en/books/bySubjectLS00/ISBN3-527-31555-1).

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 also a useful introduction to R in Appendix A ("A Brief Introduction to R") of the book *Computational genome analysis: an introduction* by Deonier, Tavaré and Waterman (Springer).

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 in-depth information and more examples on using the "graph" package for analysing graphs, look at the "graph" package documentation, www.cran.r-project.org/web/packages/graph/index.html.

More information and examples on using the "RBGL" package is available in the RBGL documentation at www.cran.r-project.org/web/packages/RBGL/index.html.

More information and examples on using the "Rgraphviz" package is available in the Rgraphviz documentation at www.bioconductor.org/packages/release/bioc/html/Rgraphviz.html.

More information and examples on using the "igraph" package is available in the "igraph" documentation at www.cran.r-project.org/web/packages/igraph/index.html.

There are also very useful chapters on "Using Graphs for Interactome Data" and "Graph Layout" in the book *Bioconductor Case Studies* by Florian Hahne, Wolfgang Huber, Robert Gentleman and Seth Falcon (http://www.bioconductor.org/pub/biocases/).

Acknowledgements

Many of the ideas for the examples and exercises for this practical were inspired by the book *Principles of Computational Cell Biology: from protein complexes to cellular networks* by Volkhard Helms (Wiley-VCH; http://www.wiley-vch.de/publish/en/books/bySubjectLS00/ISBN3-527-31555-1).

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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

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.

Q1. de Lichtenberg *et al* **identified protein-protein complexes in the yeast** *Saccharomyces cerevisiae* **that form during the yea**
There are about 6600 predicted genes in the *S. cerevisiae* genome. Is this the same as the number of vertices in the graph of de Lichtenberg *et al*'s data? If not, can you explain why? Note: the full paper by de Lichtenberg *et al* is available at http://www.sciencemag.org/content/307/5710/724.abstract

Q2. What is the minimum, maximum and mean number of interactions for the proteins in the graph of de Lichtenberg *et al*
Can you find an example of a *hub protein*? Make a histogram plot of the number of interactions for the *S. cerevisiae* proteins in de Lichtenberg *et al*'s data set.

Q3. Make a random graph with the same number of vertices and edges as the graph of de Lichtenberg *et al*'s **data. What is**
Is there a difference in the minimum, maximum and mean degree of the vertices for the random graph, when compared to the graph of de Lichtenberg *et al*'s data? Compare a histogram plot of the degree distribution for the random graph to a histogram plot of the degree distribution for Lichtenberg *et al*'s data set. What do the differences tell you?

Q4. How many connected components exist in the graph of de Lichtenberg *et al*'s **data?** How many connected components just contain 2 proteins? Make a plot of the largest connected component in the graph of de Lichtenberg *et al*'s data.

Q5. What proteins does yeast protein YPR119W interact with, in de Lichtenberg *et al*'s **data?** Draw a picture of the connected component that yeast protein YPR119W belongs to. Can you see YPR119W in the picture? Plot the communities in this connected component. Which communities does YPR119W belong to? What protein complex(es) do you think YPR119W belongs to? Can you find anything about the nature of the interactions between YPR119W and the proteins that it interacts with? Hint: search for YPR119W and the proteins that it interacts with in the Saccharomyces Genome Database (www.yeastgenome.org/). It may also be useful to look at Figure 3 in de Lichtenberg *et al*'s paper (http://www.sciencemag.org/content/307/5710/724.abstract). Can you identify the complex(es) that YPR119W belongs to in Figure 1 of de Lichtenberg *et al*'s paper?

CHAPTER 2

Acknowledgements

Thank you to Noel O'Boyle for helping in using Sphinx, http://sphinx.pocoo.org, to create this document, and github, https://github.com/, to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, to build and distribute this document.

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address alc@sanger.ac.uk

License

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