BIOL365 Evolutionary Prac 3 — Aligning genetic data and building phylogenies in R

Dr James B Dorey and Dr Damien Esquerré, University of Wollongong

Version 2024-08-08; practical 2024-08-09

Abstract

Today we will be re-downloading genetic data and starting to play around with making phylogenetic trees!
Be certain to save your R script from this week to help you next week!

Contents

1 Script preparation **1**

1.1 Working directory 1

1.2 Install packages 2

1.3 Load packages 2

2 GenBank data **2**

2.1 Read data in 2

3 Filter sequences **3**

4 Align sequences **5**

5 Make a phylogeny **7**

5.1 Distance tree 7

5.2 Maximum likelihood 8

6 What's coming next week? **10**

7 New packages used today **11**

1 Script preparation

1.1 Working directory

Just like last prac, we will set our working directory. This will, likely, be the same directory as you used last week as it's all one project.

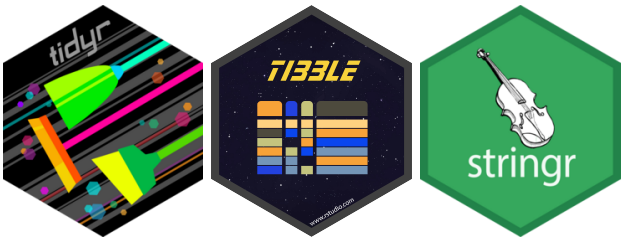
```r
  # Set the RootPath to Prac1 folder
RootPath <- "YourFolderPathHere"
  # You can then set this as the project's working directory.
setwd(RootPath)
```

1.2 Install packages

We should already have the packages from last week installed. If they are not installed I'm sure that you can figure out how to do them using the below example or copying the code from last prac!

```r
packageList <- c(
    # Tidyverse packages:
  "tibble",             #  A lovely table format package
  "tidyr",              # Another data maniupulation package from the tidyverse
  "stringr"            #  to manipulate text strings
)

# List the new (not installed) packages and then if there are any, install them.
install.packages(pkgs = c(packageList),
                 rebuild = FALSE,
                 repos = "http://cran.us.r-project.org")
```



1.3 Load packages

Once again, we will need to load the packages into R, even the ones that we used last week which were probably unloaded after you closed R following our last session. This week, we'll use a *base* function, `lapply()` that will **apply** a function over a **l**ist. In this case, we will apply our package list over the `library()` function. We will also add in the relevant packages from last week that we want to use this week. Do it all in one go!

```r
lapply(c(packageList, "dplyr", "magrittr", "ape", "rentrez"),
       library, character.only = TRUE)
```

2 GenBank data

2.1 Read data in

Let us start out by reading our fasta file backin from last week. If you've not saved the file, go ahead and download my version from the **Moodle** site and save it to your working directory now.

In addition to last week I will also get an **outgroup** sequence from a sister taxon, *Asemospiza obscura*, that will help our phylogenetic analysis **root** our tree (i.e. it will help place the most recent common ancestor for

our finches). I found this outgroup in the publication "Phylogeny of Darwin's finches as revealed by mtDNA sequences" and then found a cytB sequence accession number on GenBank — "**HQ153059**"!

Note: In this publication, the species was called *Tiaris obscurus*, but it seems like the name has changed! This is something to be aware of; taxonomy changes.

```
# Read in the data from last week
DarwinFinches_cytB_lw <- base::readRDS("DarwinFinches_cytB.rds")

# Download the outgroup cytB sequence, HQ153059, for the sister taxon, *Asemospiza obscura*
A_obscura_cytB <- ape::read.GenBank(access.nb = "HQ153059")

# We can combine the dataset from last week with the outgroup DNAbin objects simply
# by appending them together.
DarwinFinches_cytB <- append(DarwinFinches_cytB_lw, A_obscura_cytB)

# Annoyingly, we need to maually re-create the attributes for the DNAbin object and then re-
# add them to the DNAbin object
DarwinAttributes <- list(
  names = c(attributes(DarwinFinches_cytB_lw)$names,
            attributes(A_obscura_cytB)$names),
  class = attributes(DarwinFinches_cytB_lw)$class,
  description = c(attributes(DarwinFinches_cytB_lw)$description,
                  attributes(A_obscura_cytB)$description),
  species = c(attributes(DarwinFinches_cytB_lw)$species,
              attributes(A_obscura_cytB)$species)
)

attributes(DarwinFinches_cytB) <- DarwinAttributes
```

Great, that was pretty easy wasn't it? This demonstrates one of the major advantages of running these kinds of things in a language like R; once you have things set up you can run them again quite quickly. Not only that, but what you did **is easily reproducible and resilient to new mistakes!** That's important in science.

3 Filter sequences

Alright, now that I've had my little preach, let's address another issue. We have 171 sequences, so there are more than one per species. That might be fine depending on your analysis but it'll complicate things down the line in a way that we really just don't need. Lets make things easy; we'll try to take *one individual per species*. But, not all of the sequences are the same length! Longer sequences *might* lead to better chances of overlapping our gene regions, so lets first figure out how long each sequence is. Unfortunately, **ape** doesn't keep the sequence length in the attributes of the file, but we can figure that out using `lapply()` — the same function that we used to read in all of our packages with the `library()` function! Let us do the same with the `length()` function.

Note: Sometimes I'll give you some complex/confusing script, like the below, don't stress too much about understanding these data manipulations yet, just focus on the outputs and what they mean.

```
# Find the sequence lengths and turn them into a numeric list in order of how they were
# found in DarwinFinches_cytB
seqLengths <- lapply(DarwinFinches_cytB, length) %>%
  as.numeric()
```

Q1: What is the range of sequence lengths in seqLengths?

Now, we know some lengths and we want to use those to help us choose which sequences to take. What we'll do now is create a data frame (or a tibble from the **tibble** package which is nearly the same thing but a bit nicer) with all of the information that we might want to choose our sequences!

```
# Lets build a tibble column-by-column
genBankSummaryTibble <- tibble::tibble(
  accessionID = attributes(DarwinFinches_cytB)$names,
  species = attributes(DarwinFinches_cytB)$species,
  seqLength = seqLengths,
  description = attributes(DarwinFinches_cytB)$description,
)

# Let's have a look!
genBankSummaryTibble
```

Q2: What do you think would happen in our table if seqLengths were not in the same order of accession numbers as the rest of the table?

Great, now we can very quickly look at the data that we downloadeded and make some choices on which sequences that we're going to keep. For simplicity's sake, let's just take the sequence that is longest per species. To do that we will need to sort by sequence length and then choose the top sequence. Who can help us with that? Our good friend **dplyr**.

Note: There may be some full or partial mitochondrial genomes in your data (lengths > 10,000 basepairs). We will `filter()` these out for simplicity **Additionally**, there are actually two sequences here (1) the cytB sequence and (2) the control region for cytB. I'll also remove all of the "control region" sequences for you below; but be aware of this for your project and check you alignments.

```
# Let us begin to manipulate the tibble to do what we want!
  # We will put this new table into a new R object called theChosenSeqs
theChosenSeqs <- genBankSummaryTibble %>%
    # Keep only sequences less than 1,500 basepairs to ignore the mitogenomes
  dplyr::filter(seqLength < 1500) %>%
    # remove the control region sequences
      # What's that "!" mean?! This simple bit of syntax just means "get the opposite of"
      # so, instead of returning all strigns where "control region" is detected, it
      # returns all where it is NOT detected.
  dplyr::filter(!stringr::str_detect(description, "control region")) %>%
    # Group the data by species and then any sorting or filtering will be done
    # WITHIN that group
  dplyr::group_by(species) %>%
    # Sort by species and then by sequence length (top-doum; descending)
  dplyr::arrange(species, dplyr::desc(seqLength))

# Quickly, have a look at what you've made...
theChosenSeqs

# Now, let us take the top sequence of each group
theChosenSeqs <- theChosenSeqs %>%
  dplyr::filter(dplyr::row_number() == 1)
```

Q3: What's the longest sequence and what's the range of sequence lengths?

Amazing. So, now we have a table with the information that we want, how do we filter our *DNAbin* object to only include the specimens that we want? Well, now that we have found the longest sequences using our data manipulation, let's try and use the accession numbers from the above table to filter our *DNAbin* object.

In R, there are usually several different ways to do the same thing. But, we will use some *base* R syntax, the square brackets, the will select the matching elements to what's inside the square brackets.

```
# List objects often have names for each element in the list; have a look at those
names(DarwinFinches_cytB)

# If we wanted to take the FIRST element, we could simply call the first element
DarwinFinches_cytB[1]
# We could call an element using its name
DarwinFinches_cytB["56682246"]

# Extending this logic, we can use the accessionID column from theChosenSeqs to filter out
# all of the sequences that we're interested in.
finchSpecies <- DarwinFinches_cytB[theChosenSeqs$accessionID]

# But you know what having accession numbers alone is a bit annoying. Lets add
# species names as well!
  # NOTE: This is an example of complex code that you need not worry too much about
attributes(finchSpecies)$names <- stringr::str_c(theChosenSeqs$species,
                                                 theChosenSeqs$accessionID,
                                                 sep = "_")
```

Q4: What is the name of the 5th sequence in DarwinFinches_cytB?

Q5: What is the name of the 5th sequence in finchSpecies?

Q6 If you run "finchSpecies", what is the base composition (proportion) of "a"?

4 Align sequences

Now we have 15 sequences; one per species in the cytB dataset on GenBank. But, the sequences are so far just running from their first base pair until their last... We can't say that each base pair at position 1 are the same by descent (that they are homologous). We want to compare how base pairs might have changed over time at each and every site along the sequences; we need to **align the sequences**. Sometimes this is straightforwards, sometimes it really is not. But let's find out together if we've got an easy alignment ahead of us!

```
# Here, we'll use a package called msaR to observe and align our sequences
install.packages("msaR",
                 repos = "http://cran.us.r-project.org")
# We also need the BiocManager to install packages from bioconductor, where many
# genetics packages, like Biostrings and msa, are found
install.packages("BiocManager",
                 repos = "http://cran.us.r-project.org")
  # you may get asked to install extra packages, select all "a"
BiocManager::install("Biostrings")
BiocManager::install("msa")
library(msaR)
```

```r
library(Biostrings)
library(msa)
```


```r
# msaR agrees that fasta is a fantastic data format and but it recognises that not
# everyone thinks this way. So, it has provided a function to transform a DNAbin object,
# among several others, to fasta
finchSpecies_fasta <- msaR::as.fasta(finchSpecies)

# You can have a look at this output file and it looks like a horrible text string, but
# if you look closer, you'll see that it's a fasta text string with line breaks denoted
# by "\n" for "new line".
finchSpecies_fasta

# But, many R programs demand sequences of equal length. Don't worry too much about the
# below but let me help you out here...
  # Please DON'T FREAK OUT when seeing this code; I'm just giving it to you
  # to do some data wizardry (with some notes IF you're interested)
# First, lets add our fasta text into the "sequence" column of a new tibble (data frame)
lengthCorrectionTable <- tibble::tibble(sequence = finchSpecies_fasta) %>%
    # Separate sequences from one another, using the ">" as a delimiter
  tidyr::separate_longer_delim(cols = "sequence",
                               delim = ">") %>%
    # Remove the first row from the tibble
  dplyr::filter(!dplyr::row_number() == 1 ) %>%
  dplyr::as_tibble() %>%
    # Remove the hanging line break at the end of each sequence
  dplyr::mutate(sequence = sequence %>% stringr::str_remove("\\n$")) %>%
    # Use the name > sequence line break to separate them
  tidyr::separate_wider_delim(sequence, delim = "\n",
                              names = c("name","sequence")) %>%
    # THIS is the magic part, let's add blank "-" loci to pad out our alignments to the same length
  dplyr::mutate(lengthenedSequence = stringr::str_pad(sequence, pad = "-", side = "right",
                                                      width = max(stringr::str_count(sequence))),
                # If you want, you can check this object to see that lengths are the same
                length2 = stringr::str_count(lengthenedSequence))

# Now, we can turn this back into a fasta format... Asian don't worry too much about
# this code.
equalisedFasta <- stringr::str_c(">",lengthCorrectionTable$name,"\n",
                                 lengthCorrectionTable$lengthenedSequence,
                                 collapse = "\n")

# Now, let's save this fasta file for later use
write(equalisedFasta, "equalisedFasta.fasta")

# We can read this fasta in as a Biostrings object
equalised_BioString <- Biostrings::readDNAStringSet("equalisedFasta.fasta")
```

If you zoned out for that last section, start paying attention again here!

```
# START PAYING ATTENTION AGAIN!
# finally, we can interactively look at our alignment!
msaR::msaR(equalised_BioString, menu=T, overviewbox = T, labelNameLength = 200)
```

Take a deep breath, that was quite an adventure! But now we have had a first look at our sequences. You'll notice that things look a little messy, especially that the letters don't really line up nicely along columns. You should expect this for an **unaligned** set of sequences. Maybe you can even start to see where they might match up! So, let's see if we can change that.

```
# We can use the msa, Multiple Sequence Alignment package to align our sequences
alignedFinches <- msa::msa(equalised_BioString)

# Then we can access the @unmasked slot of the alignment and view the results
msaR::msaR(alignedFinches@unmasked, menu=T, overviewbox = T, labelNameLength = 200)
```

How does this alignment look? Can you see sections that align better than others? Do you see sequences that look better than others? In some cases, there may be a sequence that does not align well and may need to be removed. For your project, if this is a problem, you may choose to remove it, and add a different sequence in for that species. You can feel free to do that either in a text editor or in R; whatever you're most comfortable with.

Now we have a reasonable alignment of 15 species. That's certainly enough to make a tree with!

5 Make a phylogeny

As you will learn in your next lecture, there are a few different ways to build trees. But, for now let's keep it pretty simple.

5.1 Distance tree

Lets make one of the simplest trees possible, a *neighbour-joining* tree. This kind of tree is built based on genetic distances between sequences. The closer the sequences, the closer they'll be on the tree! We can also tell **ape** that we have an outgroup species and it will make that species sister to all other taxa.

```
# Let us start by converting the msa format sequences back into ape's DNAbin format
alignedDNAbin <- ape::as.DNAbin(alignedFinches@unmasked)

# We then need to build a pairwise distance matrix - the distances between each
# pair of sequences
distanceFinches <- ape::dist.dna(alignedDNAbin)
```

```
# Go ahead and have a look at the matrix
distanceFinches
```

```
# Now we can use that distance matrix to build a phylogenetic tree
njFinches <- ape::nj(distanceFinches) %>%
    # Specify the outgroup to be sister to all other taxa
  ape::root(outgroup = "Asemospiza_obscura_HQ153059")

# This is a tree object of class "phylo", and we need to visualise it with another
# function... creatively called "plot"
njFinches

# Let's have a look!
plot(njFinches)
  # If you have gaps in your sequence, you could look at ?ape::njs()

# the ape package also provides an "improved" version of the nj algorithm.
  # Why not compare the tree?
ape::bionj(distanceFinches) %>%
  ape::root(outgroup = "Asemospiza_obscura_HQ153059") %>%
  plot()
# Are you starting to understand how we use the pipes (%>%) from the magrittr package?
```

Q7: What instantly jumps out about the nj tree and the groupings?

Q8: Are the nj and bionj trees very different?

5.2 Maximum likelihood

The *neighbour-joining* tree is nice, quick, easy, but not the most robust implementation for tree-building. Other methods exist which you will learn more about next week. But, for now let's have a peak at these methods and the trees and see if anything jumps out at us. We'll want yet another package, for this — "*phangorn*", Phylogenetic Reconstruction and Analysis.

```
# install and load phangorn
install.packages("phangorn",
                 repos = "http://cran.us.r-project.org")
library(phangorn)
```

Next week we will be talking about genealogies and models of evolution so I don't want to overload you now. However, a short prelude to that might be: A, T, C, and G can each evolve differently and can mutate to-and-from different states in a way that we would **not** expect from random chance. So, clever folks have come up with various *models of evolution* to account for this. The model that we choose depends on our data in terms of which model best fits our data and if our data are rich enough to make using a more complex model possible! Let us use **phangorn**'s `modelTest()` function to look at this.

```
# Sadly we need to transform yet another data format from DNAbin to phyDat
# Don't ask me why they do this to us ;)
aligned_phyDat <- phangorn::phyDat(alignedDNAbin)
  # Do you see how we try to name objects so that they are concise but human-readable?

# Run the model test
finchTest <- phangorn::modelTest(aligned_phyDat,
                                 # Which models to compare
                                 model=c("JC", "F81", "K80", "HKY", "SYM"),
                                 # Let's not worry about invariant (I) or gamma (G)
                                 # variables for now. Look these up later IF you need
                                 I = FALSE, G = FALSE)
```

The `modelTest()` function lets you compare which model is the best fit for the data. Let me help you at least interpret the values that it returns! I don't expect you to remember these, but they are helpful concepts for phylogenetics and statistics at large.

- **df** — the degrees of freedom
- **logLike** — the log of the likelihood. Because likelihood values are often very small (like **really** small), we take their log. Log values <1 are negative values and they get more negative as you add decimal places. Hence, the larger number (e.g., -100 > -500) is more likely! **Side quest:** why don't you compare the outputs of *log(0.1)*; *log(0.0001)*; and *log(0.00000001)*
- **AIC** — the Akaike information criterion provides a comparison between models, given a set of data. Again, a smaller AIC value represents a more *parsimonious* (i.e., simpler) option, given the data. As a relative measure of support these numbers can only be compared within alternative hypotheses.
- **BIC** — the Bayesian Information Criterion is very similar to the AIC except it also considers the number of observations in the formula. But again, a lower value is better.

Q9: Which model is best supported by all three statistics?

You can also ask phangorn to choose the best model for you!

```
# Find me the best model, phangoorn!
fit_mt <- phangorn::pml_bb(finchTest,
                  # Change to trace = 1 to see more outputs
                 control = phangorn::pml.control(trace = 0))

# Now have a look at the best model and the relative rates of mutations table!
fit_mt
```

Q10: How many different rates are there REALLY in this model?

Q11: What are the relative rates of change for these rates?

Now, let's make a bootstrapped *maximum likelihood* tree!

```
# You can then feed in the fit_mt from above to build your bootstrap tree
bs <- phangorn::bootstrap.pml(fit_mt,
                              # Using 100 bootstraps
                              bs = 100, optNni = TRUE,
                              method = "ultrametric",
```

```
                              control = phangorn::pml.control(trace = 0))

# Plot your maximum likelihood tree. You will input a "tree" which should be the
# "Maximum clade credibility" tree (i.e., the statistically best tree from your
# bootstrap), then you can give the bootstrap trees (BStrees), which include ALL
# of the bootstrap trees
phangorn::plotBS(
  tree = bs %>%
      # Find the Maximum clade credibility tree
    phangorn::maxCladeCred() %>%
      # We can root the tree at this point
    ape::root(outgroup = "Asemospiza_obscura_HQ153059"),
  BStrees = bs ,
  p = 10, type="p", digits=2, main="Ultrafast bootstrap")
```

Q12: What's different about this tree from the last one?

Q13: Are some nodes better supported than others?

I am also going to save the DNAbin data for use in next week's prac.

```
  # Save the DNAbin as a fasta file
ape::write.FASTA(alignedDNAbin,
                 "alignedDNAbin.fasta")
```

6 What's coming next week?

Next week we will be focusing on doing something interesting and useful with our phylogeny! We will be starting to think about harvesting ecological, phenotypic, or biogeographic data to test a macro-evolutionary hypothesis using your phylogeny. We will be using Darwin's finches once again but you should be advanced enough to have at the very least an idea of what group you will examine for your project.

Be certain to save your R script from this week to help you next week! If you don't, well... sounds like you've got some catching up to do.

7 New packages used today