Week 1 Computer Lab Introduction: Getting LG Data into R

Scene 1: Getting LG Data into R

Hello, and welcome to this week's Landscape Genetics Lab!

The first task is obviously to get landscape genetic data into R, and while doing so, we will learn also how to trouble shoot R. In this introductory video, I will talk about landscape genetic data types and how they match with R object classes. Then I'll walk you through an example, where we import the Colombia spotted frog data set with two landscape genetic R packages: 'gstudio' and 'adegenet'. Along the way, I'll explain a few basic things that can go wrong.

Scene 2: Trouble Shooting R

Historically, population geneticists used a different software for every method. Nowadays, we do practically everything in R. If you've ever tried to analyze your own data in R, you'll know that data management can be more tricky and time consuming than the actual analysis. Over the years, I've learned to look for the usual suspects if something does not work. Here's my entirely subjective statistic: issues with object types are very common, for example, confusion between matrix and data frame objects. Other candidates are: missing values, sorting issues where the sampling units are in different order in different tables, or compatibility issues due to different versions of R and contributed packages.

If such a problem triggers an error message and the code stops working, it is annoying but you're actually lucky. The bigger problem is those issues that don't give you an error message but invalidate the results. Therefore, it is always important to double check every step along the way, look at the objects generated and check whether they are plausible. The good news is that if you can identify the issue, you may likely be able to fix it. Hence it is good to know what to look for.

Scene 3: Data Types – Boring yet Important

This starts with data types. On one hand, we have our landscape genetic data. Let's say we have a few individuals sampled from population A, and others from population B. We have their spatial coordinates x and y, and we have data on some relevant landscape features, like this mountain ridge between the two populations.

To get these data into R, we need to code them into the basic data types in R. Here we have a vector with four numeric values: 1, 2, -5 and 2. These could be spatial coordinates, for example. Character data are interpreted as text, even if they use numbers. This is indicated in R with quotes around each value. Think of an identifier variable, for example. If the character strings represent categorical data, like population, they should be declared a factor. A factor is a

character vector with a defined list of categories or levels. Another common type is logical data: true or false. These can be written out or we can use just capital T and F, without quotes.

Once we have coded the data, we need to store them in an R object. R is an object oriented programming language, and it has a few basic object classes. The simplest is a vector. The "c" that indicates a vector stands for 'concatenate' but you can think of it as 'combine'. A vector combines elements of the same type. Here we have a vector with four numeric values, which makes it a numeric vector. We can have character vectors, factors, and logical vectors. Each vector can have one or more elements or values, and we can even define an empty vector, simply by writing nothing between the brackets. And as a reminder here, the default for missing values is "NA", which stands for "not available". Any empty cell will be filled with NA.

The next object type is a matrix. It is essentially the two-dimensional version of a vector. We have rows and columns, but all values must be of the same type. If there are more than 2 dimensions, we call it an array. Arrays are useful for storing results from simulations.

The best way to store your data table is usually a data frame. It looks like a matrix because it has rows and columns. In a data frame, the rows should be the sampling units and the columns the variables. Each column or variable is itself a vector of a specific type, but the columns don't all have to be of the same type. Here we have a numerical column, a character one and a logical one. However, each column must have the same number of values and the same order of sampling units.

A list is more flexible. It can have many elements, and each element can be a vector, a matrix, a data frame or any other type of object we have not covered yet. Once we have the data stored in R objects, we can query them by individual, population or location.

Many problems go back to one of three things: confusion between character vectors and factors, the handling of missing values, and confusion between matrix and data frame. We'll see some of these issues in the tutorial, when we import the frog data. Make sure to check out the description video for the Colombia spotted frog data set first!

Scene 4: Example: Import Frog Data Set

So how do the frogs get into R? The frog data set contains 8 microsatellite loci. These are typically polymorphic, whereas SNPs have only two alleles per locus. The frogs are diploid and the microsatellites are co-dominant markers, hence we should have two alleles for each individual at each locus. Here we will interpret the microsat data under an infinite allele model, which means that we treat each allele as a category and code it as a factor level.

Physically, the data are organized in a couple of Excel files that were saved in comma-separated CSV format. Here are the first few lines of the file with the genetic data. Each row is one frog, and the first column indicates the population to which it belongs. The next eight columns contain the loci. The two alleles are entered in the same cell and separated by a colon. Missing

values are indicated with NA. Here the alleles of each locus are numbered 1, 2 etc but these numbers are not numerical values, we should not do math on them. We will import these data into R in two different ways, with the two R packages ‘gstudio’ and ‘adegenet’. You can do this yourself using the model code and the tutorial. The model code is meant for keeping, you can adapt it to your own projects. The interactive tutorial will help you understand the code so that you can start tweaking it.

Scene 5: Example: Import with ‘gstudio’

Let’s start with a flow chart. We have a CSV file with the genetic data. We use the function ‘read_population’ from the ‘gstudio’ package directly to import the data and create an object that we call Frogs.gstudio, all in a single step. The new object is an R data frame. The site names are coded as 'character', and the columns with the loci have a special object class, they are coded as vectors of locus objects. Each locus, for each individual, is coded as a vector of alleles. In this case, each locus object is a vector of two alleles, which may be the same or different, depending on whether the individual is homozygote or heterozygote at that locus.

Here we see the data for the first six individuals. The table looks similar to the CSV file, but internally, they are coded differently. The table itself is still a data frame, which means that we can use all the standard R functions that work on data frames. Each column, however, is a vector of class ‘locus’, and the gstudio packages contains functions that are dedicated specifically for ‘locus’ objects. For example, here I used the function ‘is_heterozygote’ on the column with locus A. We see that the first four values are ‘FALSE’ and the next two are ‘TRUE’, which means that the first four frogs were homozygotes at locus A, and frogs 5 and 6 were heterozygotes. This is correct when we check in the data table above.

Scene 6: Example: Import with ‘adegenet’

The R package ‘adegenet’ works quite differently. Here we start by importing the CSV file into an R data frame that we call ‘Frogs’. With the function 'read.csv', all text variables are automatically coded as factors, that means that our site names and the loci are stored as factors. In a second step, we use the function ‘df2genind’ from the ‘adegenet’ package. The name of the function literally means ‘data frame to genind’ object, and a ‘genind’ object contains genetic data for individuals, so this is easy to remember.

A ‘genind’ object, however, looks rather different from the original data table. Instead of a single table, it has many attributes or slots, for example, one for a table of allele frequencies and one for the number of alleles per locus. This information is automatically derived from the genetic data that were imported. Here we see a summary of the genind object. It has 181 frog individuals, genotyped at 8 loci, with a total of 39 alleles. Then we have a list of all available slots. Here’s a preview of the table of allele frequencies. Instead of eight columns, one for each locus, we now have 39 columns, one for each allele of each locus. We have three columns for alleles of locus A, and each frog has either zero, one or two copies of each allele.

The import function here is a bit tricky, and we may have to convert from factor to character vectors etc. Once it all works, the advantage is that a genind object knows how to interpret genetic data, and a lot of other functions can access the slots of a genind object as input data.

Scene 7: Under the Hood: S3 vs. S4 objects

Before we jump into the tutorial, I want to point out one more thing. You may have noticed that I used a $ symbol to access a vector with 'gstudio', and an @ at symbol for the slots with 'adegenet'. This is because of the difference between S3 and S4 objects. A data frame is an S3 object. From a programmer's perspective, S3 objects are not well defined, we can actually use the $ symbol to add any kind of element, or attribute, to an S3 object. In contrast, a genind object is an S4 object. These are clearly defined and each S4 object can have only certain predefined slots. That makes them cleaner and safer for programmers, which can be important when different R packages build on each other. What you need to remember for now is that we use the $ symbol to access an element of an S3 object, and the @ at symbol to access an element of an S4 object.

Why this confusion? A little bit of history goes a long way here. Before R, there was already a programming language called S. It was commercially available as a software called S-Plus. The S language evolved from S1 to S4, with major changes in data structure between versions. R goes back to 1995 and essentially became a non-commercial version of S. It thus branched off from S3, before the introduction of S4. Some developers of R packages have embraced the new S4 standard, others have not, and that's why as a user we sometimes get caught right in the middle. I think of it as the eternal battle between angry birds and piggies, which at least makes me smile rather than cry.

Scene 8: Getting LG Data into R

This brings us to the end. I hope this intro has prepared you to get the most out of the tutorial and model code. If you are doing this lab for credit, remember to submit your answers after working through the tutorial. Have fun!