Week 8 Computer Lab Introduction: Simulation Experiments in R

VIDEO 1

Scene 1: Simulation Experiments in R

This is a draft video to introduce the worked example on simulation modelling. In this video, we'll look at the workflow of a simulation experiment and how we can use simulations to test and compare the performance of statistical methods in landscape genetics, here the partial Mantel and an alternative with the package 'Sunder'.

A second video that is not available yet will cover a number of computational issues, that is, how we can make efficient use of R, including questions like: "where's my stuff"?

Scene 2: Simulation Workflow

Initialize: A landscape genetic simulation starts with some kind of map that determines where individuals can live. The map consists of cells. In the simplest case, this is a binary map where each cell is either habitat or non-habitat, but it could also be a mosaic of land cover types or a gradient map of habitat suitability. In addition, we will need to make assumptions about each cell's resistance to movement, that is, each cell gets a resistance value according to its cover type. We'll see more on this in Week 10.

In this lab, we'll simulate binary habitat maps with the package 'secr', which stands for spatially explicit catpure-recapture models. We can vary two landscape parameters: the proportion of habitat, and an aggregration factor that determines the patchiness. We'll assume that habitat cells have a resistance value of 1, and non-habitat cells have a resistance value of 10. In this map, green is non-habitat and light gray is habitat. That's a bit counter-intuitive, I know.

Next, we populate the landscape. Here, we will simulate a set of 8 local populations. We randomly select a habitat cell for each population, but we want them to be at least 5 cells apart. So we start with one random habitat cell, then we randomly sample habitat cells until we get one that is at least 5 cells apart, then we repeat this until we have 8 habitat cells as homes for our simulated populations.

We initialize each population with 100 diploid individuals, 50 males and 50 females. We make some assumptions about the genetic markers, here we simulate 20 microsatellite loci, each with 10 alleles and a mutation rate of 0.001. Initially, each individual gets two random alleles from each locus.

One thing to consider about the initialization is what to keep constant and what to randomly vary between simulations. Here, we keep the map constant and also the locations of the

populations will be the same for all simulation runs. However, the genotypes are randomly sampled at the beginning of each simulation run.
Time step: Next, we need to define what is going to happen at each time step. In short, we need to mimic demographic processes like survival and growth. Here we assume that we have non-overlapping generation, so that each time step is one generation. We need to define rules for mating and reproduction. Here we assume that each local population is panmictic and each female has two offspring, with a sex ratio of 0.5. At the end of each time step, we let a certain percentage of offspring disperse, and we define this dispersal with several parameters like migration rate and a maximum dispersal threshold.

In summary, we need to define a lot of demographic and genetic parameters for each time step. We store them as a list of simulation parameters. Some of these we'll keep constant, others we may want to vary between simulation scenarios.

Run a single simulation: A single simulation run starts with initializing the genotypes, then we run a predefined number of time steps, and then collect genotype information for the final generation and summarize it. Here we'll calculate a mean of pairwise Fst values to get an idea of how much differentiation there is among the populations, and we'll perform a partial Mantel test to see whether we can discriminate between isolation by distance and isolation by resistance. We'll come back to this in a moment. We can either store the full genotype information, or just the summary results like the mean Fst value and the results from the partial Mantel test.

Batch run simulations: A single simulation run does not tell us much yet. On one hand, we want to run many replicate simulations. This allows us to get an idea of the variability we should expect, and we can estimate things like type I error rates and the statistical power of a test.

On the other hand, we want to compare results between different simulation scenarios. This means that we want to vary a few parameters of interest, and ideally look at all their combinations. We define a parameter space that describes all the combinations of parameter values that we are going to explore, and then we run several replicate simulations for each parameter combination. Obviously we need to extract and store not only the results but also the information on which parameters were used for which simulation run.

Synthesize results: Once we have run our batches of simulations across parameter space, we want to extract the summary data, visualize them in parameter space and perform what is called a sensitivity analysis. That's where we quantify how much the results change when we change one parameter at a time. If there is little change, then the results are robust, they don't depend on the specific parameter setting. If there is a lot of change when we vary a parameter, then obviously this parameter has a big effect on the results. Let's think this through with the example of discriminating between isolation by distance and isolation by resistance.

Scene 3: Partial Mantel tests

The simplest way to do this is a partial Mantel test, or rather, Here we use a function 'wassermann' from the package 'PopGenReport' that performs a series of partial Mantel tests using one distance matrix for IBD and one or more distance matrices for IBR. The function is named after a paper by Wassermann et al. 2010. In the output, it ranks all models, with the best model at the top. Here the top model is a partial Mantel test of genetic distance, Gen, and the distance matrix representing IBR, or cost, partialling out the effect of IBD, or the Euclidean distance matrix, this is indicated by the vertical bar. For each model, we get the value of the Mantel r statistic and the p-value from a permutation test. This result suggests that there is a statistically significant effect of landscape resistance after accounting for geographic distance.

Scene 4: Alternative with 'Sunder'
Mantel tests have been criticized for a number of reasons, including low statistical power and inflated type I error rates in the presence of spatial autocorrelation. These findings themselves are based on simulations, and we can use simulations to test whether an alternative method performs better. We will use a method that was first proposed as 'Bedassle', but here we'll use the implementation in the R package 'Sunder'.

First we calculate D.G as the matrix of Euclidean distances between the locations of the eight populations, which we take from the list of simulation parameters. Then we defines D.E by a matrix of cost distances, ‘cost.mat’, that was derived as the cumulative cost of the least cost paths across the resistance surface. The parameter ‘n.it’ determines how many iterations will be run (…). For your final analysis, this should be set to a larger value, but in the simulation study here we’ll use 10 to the power of two or three.

The statistical analyses are all done within a function MCMCCV, which stands for Markov Chain Monte Carlo by cross-validation. The first argument is an array with the genetic data. We’ll see in the worked example how to rearrange the data in this format. Then we specify the distance matrices that hold the geographic and ecological distances, D.G and D.E. And then we need to specify a long list of parameters, and you can read up on the with this URL.

The main result we are interested in is the ranking of three models: G only, which is our IBD model, E only, which is our isolation by resistance model, or a model with both G and E. What is returned here is the likelihood, where a higher value indicates a better fit. The model ‘E’ has the highest value, which means here the least negative, and thus is considered the best-supported model. We use the function ‘which’ to extract the name of the best model (show!).

Scene 5: Testing Method Performance

How would we now compare type I error rates and statistical power between the two methods, partial Mantel test and Sunder?

To estimate type I error rates, we simulate data under the null hypothesis. If the null is panmixia, we would run the simulations with a cost distance matrix where all pairwise cost distances are zero. Here the null model is isolation by distance. Therefore, we run the simulations with the matrix of Euclidean distances as the cost distances. This means that the IBD model is correct. However, if we use a significance level of alpha = 5%, we would expect a false positive result in 5% of simulation runs. That means, even though we know the IBD model is the true model, we would expect the IBR model or the combined model, to be selected in 5% of simulation runs. More generally, we can plot the type I error rate against the significance level alpha (or simply plot a sorted vector of p-values) and it should all fall on a line.

How do we determine the power of the test? Let's consider what statistical power means. It is the opposite of the type II error rate. The type II error rate is when there is an effect but we fail to detect it, that is, the p-value was too large to reject the null hypothesis. So in this case, we simulate under the alternative hypothesis of isolation by resistance. This means that we run the simulations with the cost matrix derived from the resistance values that we set to 1 for habitat and 10 for non-habitat. Then we run many replicate simulations and determine the percentage that resulted in the IBR model being selected. That's the true positive rate, which is the same as the power of the test to detect the effect. The type II error rate is the false negative rate, which is 1 minus power.

The power will depend on the size of the effect, which is in essence the degree to which the IBR model differs from the IBD model. The lower the correlation between the two matrices, the larger the effect of IBR, and the higher the power. Power also depends on the number of populations and individuals sampled and the sampling design, genetic resolution. The worked example provides you with a tool to play around with these factors yourself!

Scene 1: Simulation Experiments in R

This brings us to the end of this first video. Check back next week for the second video on efficient R programming.

VIDEO 2

Scene 1: Simulation Experiments in R

Simulation experiments can take a long time to run, which brings us to the topic of efficient R programming. This second video serves as a brief introduction to the topic and provides links to free online sources for further reading. The two main topics covered here are 'where is my stuff', and 'why is my code slow'.

Scene 6: Where is my stuff?

Simulations can produce lots of output, and we want to control where this is stored so that we can import it again.

A general rule is to use relative paths instead of absolute paths. This makes your code more portable, between computers, operating systems and users. A relative path is relative to a root folder, or home. However, R has three homes!

The first one is the working directory. The function 'getwd' stands for 'get the working directory', and it returns the absolute path to the working directory.

What is the default working directory? It depends! In the old days before R projects, the first thing we would do is set the working directory with 'setwd'. This is because the default in a fresh R session is R's second home: the user's home directory on the computer. We can find that location with the function 'Sys.getenv("HOME")'. R's third home is the place in your system files where R is installed, and we can find it with the function 'R.home'.

If you are working in an R project, the default working directory is the project folder. We can see this here. The top line shows the path to the project folder, relative to the user's home directory: from home, go to desktop, then MyProject.

Every R project folder has an .Rproj file, and one way to open the project is by double clicking this file. My project folder here also has two subfolders, 'data' and 'downloads'. A folder structure helps organizing my files. The downside is that we need to tell R where to go looking for the file.

With R Notebooks, the working directory defaults to the place where the notebook is stored, for example the 'downloads' folder inside my project folder. This means that if I copy-paste a line of code from the Notebook to the console and run it there, the working directory may be different than if I execute the same line of code in the Notebook, and R may not find my files anymore.

Here's my advice: First, always work in an R project. Then there is no need to set a working directory. Second, use path names relative to the project folder. Here's an absolute path to myFile.csv, which is stored in the subfolder 'data' inside 'myProject'. Relative paths start with a dot and a slash, and thus, the relative path is './data/myFile.csv'.

In R Notebooks, I use a trick with the function 'here' from the package 'here'. This returns the absolute path to the project folder. With the function 'file.path', I can simply piece together the different parts of the path, here the path to the project folder, then the folder 'data', then the file 'myFile.csv'. This makes R look in the same location even when I knit the notebook.

Scene 7: Input / Output

Now we know how to tell R where to go and look for a file. The next step is telling R how to import the file.

One thing that is confusing here is that there are two parallel worlds in my R project. One is the working directory in the external file system, the other is the R internal workspace. Input means importing the content of a file from the file system into an R object in the workspace, and output means exporting the content of an R object to an external file.

We can carry out basic tasks in the file system directly from within R: list all files in the folder with 'dir', create a subfolder with 'dir.create', download a file from the web with 'download.file', unzip an archive with 'unzip', check file size with 'file.size', or delete a file with 'file.remove'.

The R workspace is the invisible place where my R objects live. It is defined by the .Rproj file, but we can't read it directly or open it with a different software. There are some analogue functions for R object: With the function 'ls', I can list all objects in the workspace, with 'object.size' I can check the size of an object, and with 'rm' I can remove objects.

The most commonly use file format for importing data into R is a '.csv' or comma separated file. The basic function for importing data is 'read.table', and 'read.csv' is simply a wrapper function for 'read.table'. Note that each import function has a corresponding export function.

The way we import or export a .csv file has evolved, and when you are dealing with large datasets, this can make a difference. The default function in RStudio is 'read_csv' from the 'readr' package, but the function 'fread' from the 'data.table' package is much faster, the 'f' here stands for 'fast and friendly'.

Binary files can be much more efficient than csv files. R has its own file formats, 'RData' and 'rds'. They are very similar, but I would recommend 'rds' because the way it imports data into an object is more intuitive. There is also a new binary file format, 'feather'.

'rds' is very efficient as it keeps files small, is fast for importing and exporting, and it can store any type of objects, including lists, but it can't be read by other software. 'feather' files are larger and only for tabular data with rows and columns, but they are ideal for switching between R and Python, which may be useful for landscape genomic projects.

A great package to use is 'rio' which stands for 'R input output'. You can use a simple function 'import' to import almost any file type, including 'csv', 'rds', 'feather', but also excel files or 'json' files. It will automatically detect the file type from the file extension and chose the best import function for the given file type.

A word of caution: these functions may differ in how they define categorical variables as 'character' or 'factor', and how they handle missing values. Always double check the data after importing them!

Scene 8: Why is my Code Slow?

File import is one example of a situation where the choice of function can make a big difference in speed. This brings us to the question 'why is my code slow'? There is a simple answer: R was not designed to be fast! The beauty of R is that it is flexible and free, and for most users who are not computer scientists, it is easier to understand code and get things done in R than in a programming language like C or Java.

If your code seems slow, I recommend three steps: identify the bottlenecks, use faster functions if available, and speed up your own code. Here are some simple things you can do for each, and ideas how to take it further.

If you are already using an R Notebook, a really simple way to find bottlenecks is simply to knit your notebook and monitor progress in the R markdown pane. This works best if you give each chunk a name, as you can then identify which chunks take a long time to knit. Those are the ones you will want to focus on to speed up your code.

A more sophisticated way to achieve this is profiling. You can measure the computation time and resource use of each line in your code. To do so, you first need to extract the R code from your notebook and save it as an R script. This is easy to do with the function 'purl. Then you source the R script with the function 'source'. Now you can use either 'lineprof' or 'profvis' to profile the code line by line, and visualize the results.

Here's an example with 'profvis': first we navigate to 'withVisible', 'eval', 'eval' until we get to the list of lines of code. Here they are sorted by computing time. The line with 'popgensim' used the most time, followed by 'getSunder' and 'wassermann'.

Different implementations of the same task can vary quite a bit in computation time. As a simple rule, vectorized functions like 'lapply' are faster than 'for' loops, and functions from the 'tidyverse' set of packages are often faster than those from base R and help speed up your workflow. A good resource for finding alternative implementations are the 'CRAN task views', which list and discuss all relevant packages on CRAN for a given set of tasks, such as spatial analysis, multivariate statistics, etc.

It is surprisingly easy to compare the performance of alternative methods with the package 'microbenchmark'. Start by wrapping each method into a function, then use 'microbenchmark' to compare their speed. Keep in mind though that speed is not the only thing that matters, there may also be differences in precision or behavior, for instance, how missing values are handled.

Here's an example comparing the speed of importing four different file types with the function 'import'. The 'csv' file took about twice as long on average than the binary file types 'RData', 'rds' and 'feather', even though the function 'import' uses the fast function 'fread' for 'csv' files.

When it comes to your own code, there are many small things you can do. For example, when you evaluate the simulation results, it is much more efficient to create the result table beforehand, with one row per simulation run, than to append rows as you loop through the runs. Overwriting the same object for each run avoids having multiple copies in memory, and using R binary files to store the simulation output reduces computation time and storage space.

Advanced options dig a bit deeper. Packages like 'data.table' or 'bigmemory' use more efficient data structures than base R. You can make your own functions quite a bit faster by compiling them with 'cmpfun'. And finally, you can increase your computational power. Keep in mind that standard R uses only one core, hence the more RAM, or random-access memory, that you have in your machine, the faster R will run. Today, most computers have multiple cores, often four of them. In order to use multiple cores, you need to parallelize your code. The actual change to the code is relatively small, just replace 'lapply' by 'mclapply', the multicore version of it, and replace 'for' by 'foreach'. Yet another step is from a single computer, or node, to a cluster, where you distribute the calculations among multiple machines. Generally, however, having all cores in the same node is often much faster than having the same number of cores distributed across multiple machines.

Scene 9: Bash R scripts 101

If you want to go beyond your own computer, you will need to leave the interactive RStudio environment and work in the terminal, or shell, using a different programming language, usually Bash. That can be scary, so here's a very brief introduction.

The first question is again, where am I? Again, this depends. If you open the shell from outside R, the default is your system home. If you open the shell from within RStudio, the default location is your project folder. You can list the content of the current folder with 'ls', and you can move to a specific folder with the function 'cd' and the path. If you use a relative path, it should start with a period and slash. You can move up one level in the file system with 'cd..'.

Once you are in the right folder, you should write a Bash R script. The core is simply your R code, which you can export from a Notebook with 'purl'. The code inside your Bash script has to be able to run on its own, outside of your session environment in RStudio. As a simple rule, if you can knit your R Notebook without an error message, then you can convert the code into an R script and insert it into your Bash file, and it should be fine.

Here we have a few lines of code that define a function 'myFunction' and applies it. To make this a Bash R script, we add the first line, called the shebang line. At the end, we insert 'EOF' for end of file, and then we tell Bash to evaluate all code in R until it hits the end of file.

We save the Bash R script with extension '.sh' for shell file. The first time we want to execute it, we first need to change file permission. Then we execute it simply by entering the file name with the relative path.

Sometimes we need to change some parameter everytime we call and run a script. In that case, we can pass one or more arguments if we add '--args' to the second line. Inside the R code, we use the function 'commandArgs' to read in the arguments from the command line. Now when we execute the file, we simply add the arguments after the file name. Obviously, we need to remember how many arguments to provide and in which order, and account for the fact that even if we provide numerical values, like here, they will be read in as character.

Hopefully, this crash course has convinced you that while the bash shell may be scary, you can figure it out if you need it.

Scene 10: Recommended Reading

Here are further resources.

The following three books are available for free online, or you can buy them printed.

- 'Efficient R programming' by Gillespie and Lovelace focuses on making your workflow and your code more efficient.
- 'Advanced R' by Hadley Wickham helps you develop your programming skills.
- 'R for Data Science', also by Hadley Wickham, focuses on efficient workflow with the tidyverse set of R packages.

'Tidyverse' refers to a coherent system of packages for data manipulation, exploration and visualisation. This should help researchers be more productive and make it easy to develop a good workflow, communicate research, and make it reproducible.

If you work with genomic sequence data, here's a highly recommended book by Vince Buffalo, though this one is not free.

Finally, there are many useful blogs with worked examples, here are a few of them.

Scene 1: Simulation Experiments in R

This brings us to the end of this video. This week's interactive tutorial focuses on generating data and manipulating character strings, and in the worked example, you will carry out a complete landscape genetic simulation experiment. The bonus material shows examples of file manipulation, including file import and export, how to benchmark and profile your code, and how to write a Bash R script.

This concludes the eight-week series of videos and interactive R tutorials. With this preparation, we hope that you will find it easy to understand the additional worked examples and adapt them to your own data.