High Performance Computing Programming Exercises

18th – 22nd November 2024 (last updated 16th November 2024)
James Rosindell (j.rosindell@imperial.ac.uk)

Deadline and handing in

- Keep all your code in GitHub.
- You will hand in your files as part of your mini project in spring 2025.
- Refer to the course guidebook for exact dates.

What to hand in

Please hand in the following files, changing the placeholder "abc123" to your own username:

- abc123_HPC_2024_neutral_cluster.R (based on pro-forma)
- abc123_HPC_2024_demographic_cluster.R (based on pro-forma)
- abc123_HPC_2024_main.R (based on pro-forma)
- all your graphs as .png files
- your shell script (.sh) files for running on the cluster
- .e and .o files generated by the cluster for your simulation runs
- .rda output files saved from the cluster for your simulation runs, .rda file from your output in Q20, and .rda results from any Challenge questions which take a long time to produce
- optional additional .rda files or .R files that get sourced by your other files

Important notes about the hand in

Some aspects of the marking will be automatic, or assisted by automated systems, so it's important you pay attention to the specifications and follow them precisely.

- Do not use packages (other than ggplot2, if you wish) – nothing else is needed.
- All files should be in one folder (no sub folders please).
- All challenge questions should be in the one main.R file you hand in.
- Any slow computations for challenge questions (>1 min) should be pre-calculated and stored in a separate .rda file so that I can run the main function quickly.
- Create new functions whenever you wish, e.g., to create the .rda files, to abstract parts of the code, etc. However, **the function names in the pro forma must remain as defined** (as these will be called by name in the auto marking script).
- **Never ever** clear the workspace in your main file. This is because your main file is treated like a library of functions that can be sourced and used, either by your own experimentation to check it works, or by my marking code, or by a file to be run on HPC.
- Always **include your own username and name files properly using lower case**.
- If I run the source command on your main.R file it should run very quickly without error and load all the functions into memory so that they can be tested – **it should not actually run the functions** or perform any other tasks (such as plotting or printing).
- Don't try to change the working directory in the code or do anything clever with the paths that would not work on another machine. Assume that the working directory will be directory that contains all your files with no subfolders.
    - You **can** source("abc123_HPC_2024_main.R")
    - You **cannot** source("Documents/HPC course/abc123_HPC_2024_main.R)

Please tell me if you spot errors in this document so that I can correct them.

Section one: Stochastic demographic population model

We will describe a population using a population state vector called ***state*** which contains the number of individuals in each life stage. So if ***state*** was ***{ 1 0 3 }*** then there would be three life stages, with one individual in stage 1, none in stage 2, and three in stage 3 (so a total population size of four divided into 3 distinct life stages).

You will be given code in an R file ***demographic.R*** which carries out a deterministic simulation of a population with life states. Don't worry about how the underlying model works (we will look at this in a later lecture). This exercise is about using simulation code locally and on an HPC system and you may need to do this in future with code that you didn't write yourself. Later in this worksheet you will write a different simulation of your own and that you will also get to run on the cluster.

You will have access to a function called ***deterministic_simulation*** which takes the inputs ***initial_state***, ***simulation_length*** and ***projection_matrix***. The ***initial_state*** is a vector describing the starting state of your simulation, the ***simulation_length*** is the amount of time that the simulation should run for and the ***projection_matrix*** describes how the simulation will work. The function will return a vector of length (simulation_length+1) representing a time series of the total population size, so that the [n+1]th entry of the vector is the total population size at time step n (and the first entry is the initial population size, at time step 0). Important note: your ***projection_matrix*** should be a square and its number of rows and number of columns should be equal to the length of your ***initial_state*** vector.

0) (Preliminary question, yes, we're starting at zero and not at one)

   Write a function called state_initialise_adult which creates a state vector of length num_stages representing a population of size initial_size, all of which are in the adult life stage (the final life stage). [Hint: The rep function].

   Write a function called state_initialise_spread which creates a state vector of length num_stages representing a population of size initial_size. This time, the population should be spread out across the life stages as evenly as possible. If the population size is not divisible by the number of life stages, then allocate the remaining individuals starting from the youngest life stage first. For example, state_initialise_spread(num_stages=3, initial_size=8) should return { 3 3 2 }. [Hint: The floor function may be useful].

   [3 marks]

1) Write a function in the file with name ending ***main.R*** called ***question_1*** (with no inputs) which carries out a deterministic simulation with two different initial conditions and saves a graph comparing the population size time series. You will need to source the code from the provided ***demographic.R*** file. Run the deterministic model for a species with four life stages, with a simulation length of 24, and projection matrix as given at the end of this question, using the following two initial conditions: a population of 100 adults (final life stage), and a population of 100 individuals spread across the four life stages. Create and save the plot ***question_1.png*** which should show the population size time series for both simulations using different-coloured lines. The function should output as text the answer to the question "*Explain how the initial distribution of the population in different life stages affects the initial and eventual population growth.*"

   For your simulation, the ***projection_matrix*** to be used is defined as follows:

```
growth_matrix <- matrix(c(0.1, 0.0, 0.0, 0.0,
                          0.5, 0.4, 0.0, 0.0,
                          0.0, 0.4, 0.7, 0.0,
                          0.0, 0.0, 0.25, 0.4),
                        nrow=4, ncol=4, byrow=T)
```

Which looks like:

```
     [,1] [,2] [,3] [,4]
[1,]  0.1  0.0  0.0  0.0
[2,]  0.5  0.4  0.0  0.0
[3,]  0.0  0.4  0.7  0.0
[4,]  0.0  0.0  0.25  0.4
```

```
reproduction_matrix <- matrix(c(0.0, 0.0, 0.0, 2.6,
                                0.0, 0.0, 0.0, 0.0,
                                0.0, 0.0, 0.0, 0.0,
                                0.0, 0.0, 0.0, 0.0),
                              nrow=4, ncol=4, byrow=T)

projection_matrix = reproduction_matrix + growth_matrix
```

Note that using the `matrix` function in R, the default is to fill out the matrix column by column. This may be counter-intuitive since in English we write from left to right. By specifying `byrow=T` we are instructing the `matrix` function that we want to fill out the matrix row by row instead.

[4 marks for correct code, results and graph]

We will now look at a stochastic version of the same model, the principles of the simulation are very similar but because of the stochasticity the result you get back will not be the same each time you run the simulation.

You will have access to a function called ***stochastic_simulation*** which takes the inputs ***initial_state***, ***growth_matrix, reproduction_matrix, clutch_distribution***, and ***simulation_length. initial_state*** and ***simulation_length*** have the same meaning as your ***deterministic_simulation*** function. All the other parameters describe the inner workings of the model. The function returns a vector of length ***simulation_length+1*** which is the time series of the population size, just as in ***deterministic_simulation.***

2) Write a function in the file with name ending ***main.R*** called ***question_2*** (with no inputs), which applies the ***stochastic_simulation*** with the same two sets of initial conditions and other parameters as given to the deterministic model in Question 1, ***growth_matrix, reproduction_matrix*** are now used separately but are as defined above, the ***clutch_distribution*** is as provided at the end of the question. Create and save a plot called ***question_2.png*** which plots the population size time series for both simulations using different-coloured lines. Your function should output a plain text answer to the question "*How does the smoothness of the earlier deterministic simulations compare with these stochastic simulations? Explain why this is the case.*"

```
clutch_distribution <- c(0.06,0.08,0.13,0.15,0.16,0.18,0.15,0.06,0.03)
```

[3 marks]

Now you're going to run your stochastic simulation many times, using the HPC cluster.

Important note: don't panic! This question is a lot harder than the previous two but you can do it, just take a small step at a time to break down the problem of writing the code.

3) Write an R script in the file with name ending ***demographic_cluster.R*** which, when sourced, will carry out simulations on the HPC cluster. There will be 100 cluster runs with four different initial conditions – so a quarter of the simulations should be assigned to each initial condition. Each cluster run will perform the simulation 150 times (so in total there will be 15,000 simulations spread across your cluster nodes).

   In all simulations, there are four life stages. The projection matrix and clutch distribution are those used previously in Question 2, but this time, run the simulation for 120 steps (which is ten years since our time scale is months). The four initial conditions are (1) a large population of 100 adults, (2) a small population of 10 adults, (3) a large population of 100 individuals spread across the life stages, and (4) a small population of 10 individuals spread across the life stages.

   Your code will need to achieve, in this order:

   - Clear the workspace and turn off graphics.
   - Load all the functions you need by sourcing the provided demographic.R file
   - Read in the job number from the cluster. To do this, your code should include a new variable ***iter*** and should start with the line:

     ```
     iter <- as.numeric(Sys.getenv("PBS_ARRAY_INDEX"))
     ```
   - Control the random number seeds so that each parallel simulation takes place with a different seed. Your function should therefore set the random number seed as ***iter*** so that each parallel simulation has a unique random seed.
   - In each parallel run, select the initial condition to be used. Ensure that 25 of the parallel simulations are allocated to each of the initial conditions.
   - Create a variable which is the filename to store your results. The end of the filename should be the number ***iter*** to ensure that simulation files do not overwrite one another on the cluster.
   - Initialise a list which will contain the results of your 150 simulations.
   - Call the ***stochastic_simulation*** function to do the appropriate simulation 150 times, appending the output to the results list, and then save the results list.

   [10 marks for correct code]

4) Write a shell script for running your code on the cluster. You need to decide what wall time to give the cluster. I recommend you check how long it takes to run one of the large initial population size simulations on your own computer and then use that to inform how long you should give a cluster node to do this 150 times.

   Use sftp and ssh to set your jobs running on the cluster as instructed during the lecture (and see lecture notes). Then test on the cluster with a job number (***iter***) of 1, 2, 3 only (use -J 1-3 as in the lecture notes). Finally, run the full set of jobs to the cluster (-J 4-100) if the first three came out OK.

   [10 marks for all output files (.rda, .e, and .o) and shell script code]

The next step is to write two R functions to plot the results from your cluster run. These functions will read in and process your results, assuming that all of the necessary .rda files are in the current working directory. If your cluster simulations are still running I suggest skip ahead and make progress on section 2 while you wait.

5) Write a function called ***question_5*** (with no inputs) that should produce and save a bar graph showing the proportion of stochastic simulations which resulted in extinction for each initial condition. This function needs to work out how many extinctions occurred for each initial condition, and divide that by the total number of simulations to compute the proportion of simulations resulting in extinction. It should then plot a bar graph with initial condition on the x-axis and proportion of populations which went extinct on the y-axis, and should save this bar graph as ***question_5.png***.

The x-axis labels should be interpretable, e.g. “adults, large population” rather than “initial condition 1”. Your function should then return a plain text answer to the question “*Which population was most likely to go extinct? Explain why this is the case.”*. [Hint: If a population went extinct, its final population size would be equal to 0].

[8 marks for correct code, graph and results]

6) Write a function called ***question_6*** (with no inputs) that should produce and save a line graph showing the deviation of the average population trend of the stochastic model from the corresponding demographic model. **Only analyse the simulation files for initial conditions 3 and 4** (i.e., we will only look at the small mixed and large mixed populations). For each of these two initial conditions you will need to:
   - Identify the appropriate ***population_size*** results for this initial condition.
   - Compute the mean population size (which we will refer to as the “population trend”) at each time step across all simulations with this initial condition. [Hint: For a given initial condition, the population trend can be calculated as the sum of all the population size time series vectors divided by the number of simulations.]
   - Compute the population size time series produced by the deterministic model ***deterministic_simulation*** (using the same projection matrix and simulation length as your stochastic simulations).
   - Compute the deviation of the stochastic model from the deterministic model by dividing the stochastic population trend vector by the deterministic population size vector (obtaining, at each point in time, the value of the average population size from the stochastic model divided by the value of the deterministic population size). Note that if this value is 1, that means that the deterministic population size is the same as the average behaviour of the stochastic model.
   - Plot on the same graph as panels the deviation of the stochastic model from the deterministic model for each initial condition (ensure that each line is identifiable).
   - Save this figure as ***question_6.png***. Give the graph an interpretable title.

   The function should return the plain text answer to the question “*For which initial condition is it more appropriate to approximate the ‘average’ behaviour of this stochastic system with a deterministic model? Explain why.*”

   [10 marks for correct code, graphs and results]

Challenge Question A: Important: do not attempt unless you’ve finished all the non challenge questions, including those in section two. Write an R function called ***Challenge_A*** with no inputs. When run, this function will go through all the data produced by the cluster simulations and output a data frame called ***population_size_df***. This is a data frame containing ***(simulation_size+1)*150*100*** rows; a long-form data frame where each row represents a specific time point for a specific simulation run. The columns of this data frame should be ***simulation_number*** (such that each individual simulation has a unique identifier - it doesn't necessarily need to be the job number), ***initial_condition*** (either "small adult", "large adult", "small mixed", or "large mixed" as appropriate), ***time_step*** (from 0 to ***simulation_size***) and, finally, ***population_size***, which is the corresponding population size from this stochastic simulation at this time step. Use your data frame to produce and save a graph called ***Challenge_A.png*** which should plot all the population size time series against time in a single graph, using the package *ggplot2*. Use ***geom_line***, set ***aes(x=time_step, y=population_size, group=simulation_number, colour=initial_state)***, and set ***alpha=0.1*** (modify alpha if needed) to produce faint, but visible, overlapping lines.

Section Two: Individual-based ecological neutral theory simulation

These questions build on one another, step by step, so that by the end you will have produced your own individual-based ecological neutral theory simulation code in R.

You will store the state of your simulated system as a vector of individuals called ***community***. Each entry in the vector is a number that tells you the species of the individual in that position. For example, if ***community = c(1,1,1,3,1)*** then four individuals in the community are species 1, and one individual is species 3.

7) You will need to know the species richness of your system, so write a function ***species_richness*** to measure the species richness of the input ***community*** which is a vector. For example, ***species_richness(c(1,4,4,5,1,6,1))*** should return 4. (Hint: the 'unique' command).

[2 marks]

8) Write a function ***init_community_max*** to generate an initial state for your simulation community with the maximum possible number of species for the community of size given by the input number variable ***size***. This means that every individual in the community will be a different species to all the others (Hint: the 'seq' command).

[1 mark]

9) In this type of simulation, it's important to consider the effect of the initial condition. Write another function ***init_community_min*** to generate an alternative initial state for your simulation. Again, the output will be a community of size given by the input variable ***size.*** This is monodominance of one species with a total number of individuals given by ***size***.

[1 mark]

~ Test your code ~
For any reasonable value of x,

- `species_richness(init_community_min(x))` should always return 1, and
- `species_richness(init_community_max(x))` should always return x.

10) Write a function ***choose_two***. This function should first choose a random integer (whole number) according to a uniform distribution between 1 and the input ***max_value***, inclusive of the endpoints. It should also choose a second random integer between 1 and ***max_value*** but not equal to the first number. The numbers should be returned as a vector of length 2. So ***choose_two(4)*** might return the vector {3 4}, or one of a number of other vectors with equal probability. (Hint: the 'sample' command).

[2 marks]

11) Write a function ***neutral_step*** to perform a single step of a simple neutral model simulation, without speciation, on a community vector ***community***. You will need to (randomly, based on a uniform distribution) pick an individual to die and another to reproduce and fill the gap left by the death; they should not be the same individual (though could be of the same species). For example, ***neutral_step(c(1,2))*** would return { 1 1 } or { 2 2 } with equal probability (either the first individual dies and is replaced by the second's offspring, or the second individual dies and is replaced by the first's offspring). (Hint: call your function ***choose_two***, thinking of the numbers returned as indexes of your ***community*** vector where the individuals' species are stored).

[2 marks]

12) Write a function ***neutral_generation*** to simulate several ***neutral_step***s on a community so that a generation has passed. A generation is the amount of time expected between birth and reproduction (not the time between birth and death, which is longer if generations overlap). If the

community consists of x individuals, then x/2 individual neutral steps will correspond to a complete generation for the taxa being simulated. If x is not an even number, choose at random whether to round up or down to the nearest whole number to determine how many neutral steps will correspond to a generation. For example, if there are 10 individuals in the community then 5 neutral steps mean 5 births and 5 deaths (so one generation). Your function should return a vector giving the state of the ***community*** after a generation has passed. (Hint: ***neutral_step*** and a loop).

[2 marks]

13) Write a function ***neutral_time_series*** that will do a neutral theory simulation and return a time series of species richness in the system. The function should have two inputs: ***community*** (the initial condition community vector) and ***duration*** (the number of generations you want to run the simulation for). The function should return a vector giving the species richness at each generation of the simulation run, starting with the initial condition species richness. For example, ***neutral_time_series(community = init_community_max(7), duration = 20)*** should return a vector containing a time series vector of length 21, with the first value being 7 (the species richness of the initial condition community). (Hint: ***neutral_generation*** and a loop).

[2 marks]

14) Write a function ***question_14*** to plot and save a time series graph of your neutral model simulation from an initial condition of maximal diversity in a system size of 100 individuals. Run the simulation for 200 generations. The function should require no inputs to run and should return a plain text answer to the following question: *"What state will the system always converge to if you wait long enough? Why is this?"*. Your plot should be saved as ***question_14.png*** (Hint: use ***neutral_time_series*** and the plot command. You can use the plot saving code given in the pro-forma. Check the plot .png looks OK before you move on!)

[3 marks]

15) Write a new function ***neutral_step_speciation*** which will perform a step of a neutral model with speciation. In each time step, with probability ***speciation_rate*** speciation will replace a dead individual with a new species, and otherwise (with probability 1 − ***speciation_rate****)* the dead individual is replaced with the offspring of another individual, as before in ***neutral_step***. You should make ***speciation_rate*** be an input parameter (between 0 and 1) in your function. For example, ***neutral_step_speciation(c(1,1,2), speciation_rate = 0.2)*** should behave like ***neutral_step(c(1,1,2))*** with probability 0.8, and with probability 0.2 it should instead be equally likely to return { 1 1 n }, { 1 n 2 }, { n 1 2 }, with n in each case being a species number different to any other species number in the current community, here anything other than 1 or 2. (Hint: be careful to make sure that any new species really has a unique number assigned that is currently not used by any other species in the community.)

[3 marks]

16) Make a new function ***neutral_generation_speciation*** which uses a neutral simulation with speciation and advances an initial community one generation according to the rules of the model. (This is similar to what ***neutral_generation*** achieves but with the new speciation rule). The new function should have two inputs: the initial ***community*** vector and the ***speciation_rate***. It should return the state of the community at the end of all the steps of simulation which make up the generation.

[1 mark]

17) Make a new function ***neutral_time_series_speciation*** which uses our neutral simulation with speciation and advances a number of generations given by ***duration***. (This is similar to what ***neutral_time_series*** achieves but using our speciation model). This function should have three

input parameters; the same two as in ***neutral_time_series***, and the additional input ***speciation_rate***. The return should be in the same format as ***neutral_time_series***; a time series vector containing the species richness at each generation, with the first entry of the vector being the species richness of the initial condition community).

[1 mark]

18) Write a function ***question_18*** to perform a neutral theory simulation with speciation, plot species richness against time, and save this plot (similar to in question 14). Use a speciation rate of 0.1, a community size of 100, and run your simulations for 200 generations. Plot two time series on the same axes in different colours showing how the simulation progresses from two different initial states given by ***init_community_max*** and ***init_community_min***. Your function ***question_18*** should require no inputs to run, should display and save the plot as ***question_18.png***, and also return a plain text answer to the following question *"Explain what you found from this plot about the effect of initial conditions. Why does the neutral model simulation give you those particular results?"*

[4 marks]

19) You are going to study the species abundance distribution of these neutral simulations. First, you need to write a function ***species_abundance*** to tell you what the abundances of all the species are in the system ***community***. It should return a vector containing the number of individuals of each species, in descending order of abundance. For example,
species_abundance(c(1,5,3,6,5,6,1,1)) should return { 3 2 2 1}.
This is because there are 3 of species "1", 1 of species "3", 2 of species "5", and 2 of species "6" (and these four abundances are presented in descending order). (Hint: the 'table' and 'sort' commands.)

[3 marks]

20) Write a function called ***octaves*** to bin the abundances of species (e.g., the output of the ***species_abundance*** function) into what would be called 'octave classes'. The first value of the returned vector should tell you how many species have an abundance of only 1, the second value of the returned vector should tell you how many species have an abundance of either 2 or 3, and in general the n^th^ value of the returned vector sholould tell you how many species have an abundance greater than or equal to 2^{n-1} but strictly less than 2^n. For example, in the 6th element of the octave vector will be the number of species in octave class 6. This is the number of species with an abundance of greater than or equal to 2^5 (which is 32) but less than 2^6 (which is 64). (Hint: the 'log', 'floor', and 'tabulate' functions may be useful. If you're not sure what to do, try writing out a table of abundances and the octave that they fall in, then look for a way to generate this in R using the functions mentioned in this hint.)

[3 marks]

These simulations are stochastic, so you will need to average the result from a number of independent readings to get an idea of the overall behaviour of the system. You will find that octave vectors are not always the same length; for example, if there were 100 individuals all of different species, then the octave vector would be { 100 } (vector of length 1), while if these individuals were all the same species, the octave vector would be { 0 0 0 0 0 0 1 } (vector of length 7). In such cases, R will not allow you to simply add them – or, worse, will add them in a way that you do not intend, and give you the wrong answer. The next question is to help you solve this problem.

21) Write a function ***sum_vect(x,y)*** which accepts two vectors as inputs, ***x*** and ***y***, and returns their sum, after filling whichever of the vectors that is shorter with zeros to bring it up to the correct length. For example, ***sum_vect(c(1,3),c(1,0,5,2))*** should return { 2 3 5 2 } by adding { 1 3 0 0 } and { 1 0 5 2 }. (Hint: the 'length' function and 'if' command).

[2 marks]

22) Write a function ***question_22*** to run two neutral model simulations and save bar plots of the species abundance distributions. Use the same parameters as in question 18 for a "burn-in" period of 200 generations, again running the simulation from both initial conditions. After the burn-in period, record the species abundance octave vector. Continue the simulation from where you left off for a further 2000 generations, recording the species abundance octave vector every 20 generations. Produce a bar chart plot of the mean species abundance distribution (as octaves) by calculating a mean for each bar of the octaves plot from all the recorded octave vectors. Perform this for both initial conditions (minimum and maximum initial species richness). Save these plots as two panels in ***question_22.png*** using the sample code included in the function. Your function ***question_22*** should require no inputs to run and should return a plain text answer to the following question: *"Does the initial condition of the system matter? Why is this?"*

[4 marks]

Challenge Question B: Write a function ***Challenge_B*** to plot the mean species richness as a function of time (measured in generations) across many repeat simulations using the same parameters as in question 22. Add a 97.2% confidence interval on the species richness at each point in time. Repeat this for both initial conditions (minimum and maximum initial species richness) shown in different series on the same plot. Your function should have no inputs and should save your plot as ***Challenge_B.png***. Estimate the number of generations needed for the system to reach dynamic equilibrium and output this in a full sentence as a plain text return.

Challenge Question C: Write a function ***Challenge_C*** to plot and save a graph called ***Challenge_C.png*** showing many averaged time series for a whole range of different initial species richnesses. In each initial community state, each individual should be equally likely to take any species identity. (Hint: It's OK here and elsewhere to create additional functions of your own to help streamline your code or make your working cleaner).

You are going to be running a much larger simulation of the same type that you conducted for your answer to question 22 and with more repeat readings. To do this requires use of high performance computing (HPC) and some adaptations of your R code.

23) Create a function ***neutral_cluster_run*** which accepts seven input parameters: ***speciation rate, size, wall_time, interval_rich, interval_oct, burn_in_generations*** and ***output_file_name***. This function should:
    - Start with a community with size given by the input ***size*** and with minimal diversity (species richness).
    - Apply neutral generations with a speciation rate given by ***speciation_rate*** for a predefined amount of computing time ***wall_time*** measured in minutes. (Hint: if you're not sure where to start, get a timer working on its own first; use the 'proc.time' command for this.)
    - Store the species richness at intervals of ***interval_rich***, but only during the burn-in period, which is measured in generations. After the number of generations exceeds the burn-in time ***burn_in_generations***, stop recording the species richness. The species richness should be recorded in a vector called ***time_series*** (Hint: use a vector to store the species richnesses and use '%%' to help detect when to do this. It's probably easier to have one main simulation loop and use ''if' statements inside it to determine whether the simulation is burning in or not. It's worth noting that the point of this is to create a time series; a series of species richness values across time. However, you should *not* use *neutral_time_series_speciation()* because that function doesn't return the community state at the end so cannot be used to continue the simulation.)

- For the entire simulation, until the simulation runs out of time (as determined by ***wall_time***), you should record the species abundances as octaves every ***interval_oc*** generations. (Hint: use 'list'.)
- You should save your simulation results in a file with name given by the input ***output_file_name*** including the following data: the ***time_series*** of species richness recorded during the ***burn_in_generations***, the list of species abundance octaves ***abundance_list***, the state of the ***community*** at the end of the simulation, the total amount of time ***total_time*** actually consumed on the simulation, and all seven of the input parameters for the function (except for the ***output_file_name*** – we don't need to store a reminder of what the file name is inside the file itself!). (Hint: use the 'save' command, and don't call your outputs by the same names as your functions – otherwise the functions and not the outputs will be saved.)
- Test your code locally before proceeding further using the same parameters from question 22 and a short time limit of 5 to 10 minutes. For example:

  ```
  neutral_cluster_run(speciation_rate=0.1, size=100,
  wall_time=10, interval_rich=1, interval_oct=10,
  burn_in_generations=200,
  output_file_name="my_test_file_1.rda")
  ```

  This should run for 10 minutes and return nothing but save a file called *"my_test_file_1.rda"* which you can then open in R and look at the data to check you've got everything you expected.

[6 marks]

24) Use a new R file for running simulations on the cluster, based on the provided pro forma. As you code, press "source" every time you run it because that's what will happen on the cluster. Now you're ready to write lines of code in your new R file which will source and use the functions you have written. It should be that when you run the file using the source command you will get the simulation you want. You will not be writing another function, but rather writing R code in the file to be run. The code needs to achieve, in this order:

- Clear the workspace and turn off graphics.
- Load all the functions you need by sourcing your main.R file (and any other necessary .R files).
- Read in the job number from the cluster. To do this, your code should include a new variable ***iter*** and should start with the line:

  ```
  iter <- as.numeric(Sys.getenv("PBS_ARRAY_INDEX"))
  ```

  However, for testing on your own machine this will not work, so write the line and then comment it out and instead set ***iter*** yourself to different numbers for local testing. The last thing to do before you run your code on the cluster is then to comment out your "local testing" line and uncomment the real job number line.
- Control the random number seeds so that each parallel simulation takes place with a different seed. If you run two simulations with the same seed, you will get the same answer regardless of the fact that it's a stochastic simulation (since computers are only *pseudo*-random). Your function should therefore set the random number seed as ***iter*** so that each parallel simulation has a unique random seed*.*
- In each parallel simulation, select the community size being used. The community sizes to be simulated are 500, 1000, 2500, and 5000. Ensure that 25 of the parallel simulations are allocated to each of these community sizes. For example, you could set ***size=500*** when ***iter*** is between 1 and 25 (inclusive), and so on.
- Your speciation rate will be the same for all your simulations. However, each person will be given a different speciation rate, handed out separately to this workbook.

- Create a variable which is the filename to store your results. The end of the filename should be the number ***iter*** to ensure that simulation files do not overwrite one another on the cluster. (Hint: use the 'paste' command.)
- Call the ***neutral_cluster_run*** function, which will actually do the simulation and save the results. Use ***interval_rich=1***, ***interval_oct=size/10***, and ***burn_in_generations=8*size***.
- Use a time limit of 12 hours for all your jobs (give your code a time of 11.5 hours and tell the cluster 12 hours just in case).

[6 marks for the correct code]

25) Write a shell script for running your code on the cluster. Use sftp and ssh to set your jobs running on the cluster as instructed during the lecture (and see lecture notes). Then test on the cluster with a job number (***iter***) of 1, 2, 3 only (use `-J 1-3` as in the lecture notes). Finally, run the full set of jobs to the cluster (`-J 4-100`) if the first three came out OK.

[10 marks for all output files (.rda, .e, and .o) and shell script code]

26) Plot the results from your cluster run. While your job is running on the cluster, you can write an R function ***process_neutral_cluster_results*** to read in and process the output files. Your code should read in all your output files (assume them to be sitting there in your current working directory). It should only use data of the abundance octaves after the burn-in time is up. The function should calculate a mean value across all the saved (post-burn-in) data for each abundance octave and for each community ***size***. (I.e., averaging not just across time but also across all the 25 simulations of each community size). It should save all the summarised data in a new .rda file as a list of four vectors corresponding to the octave outputs that plot the four bar graphs. The vectors should appear in the list in increasing community ***size*** order (***size=500*** first, then 100, etc.) (Hint: use the 'load' function on your .rda files and use ***sum_vect***). Next, write an R function ***plot_neutral_cluster_results*** that should provide and save four bar graphs in a multi-panel graph (one bar graph for each community ***size***) each showing a mean species abundance octave result from all simulation runs of that size and the list of data it plotted. This function should save the multi-panel graph as ***plot_neutral_cluster_results.png***.

[10 marks for your graphs and correct results]

Challenge Question D: Write a function ***Challenge_D*** to plot a graph of mean species richness against simulation generation and use it to inform you more precisely how long should have been allowed as a burn-in period for different values of ***size***. This function would also need to read in your simulation data and process it, as in ***process_neutral_cluster_results***. Your function should save the graphs as ***Challenge_D.png***.

Challenge Question E: Write a function ***Challenge_E*** to conduct further simulations of the same system using coalescence (see the pseudo code below). Check that your results from the cluster agree with those from coalescence (plot and save a graph in the file ***Challenge_E.png*** to show this) and compare the speed of the two approaches. Return plain text to answer the question *"How many CPU hours were used on the coalescence simulation and how many on the cluster to do an equivalent set of simulations? Why were the coalescence simulations so much faster?"*

Pseudo code for Challenge E, to get a coalescence simulation of the neutral model from question 22 as a function of community ***size*** (J) and ***speciation_rate*** (v).

```
a) Initialise a vector lineages of length $J$ with $1$ in every entry.
b) Initialise an empty vector abundances (of length 0).
c) Initialise a number $N = J$.
d) Calculate $\theta$, where $\theta = v\frac{J-1}{1-v}$.
e) Choose an index j for the vector lineages at random according to a uniform distribution.
```

f) Pick a random (decimal, not integer) number ***randnum*** between 0 and 1 (with a uniform distribution).

g) If ***randnum*** $< \frac{\theta}{\theta+N-1}$, append ***lineages[j]*** to the vector ***abundances***.

h) If ***randnum*** $\geq \frac{\theta}{\theta+N-1}$, choose another index ***i*** for the vector ***lineages*** at random, but do not allow ***i=j***. Then set ***lineages[i] <- lineages[i] + lineages[j]***.

i) Remove ***lineages[j]*** from ***lineages*** so that the ***lineages*** vector is now one shorter.

j) Decrease N by one so that N still gives the length of the ***lineages*** vector.

k) If $N > 1$, repeat the code again from (e) to here.

l) Once $N = 1$, add the only element left in ***lineages*** to the end of ***abundances***.

m) END: A vector of simulated species abundances is stored in ***abundances***.