High Performance Computing Programming Exercises

17th - 21st November 2014

James Rosindell (j.rosindell@imperial.ac.uk) Munro N2.6

On each question, it will be indicated [in brackets] how many marks are available. The marks add up to 100, but together they only count for 60% of your final score for the assignment – you can therefore obtain a merit by only giving technically correct answers to the main questions whilst ignoring the 'challenge questions'. The remaining 40% of your final score will be discretionary and based the overall quality of your text answers and the extent to which they demonstrate your understanding of the topics, the style and quality of the computer code that you handed in and your answers to any of the challenge questions. 'Challenge questions' are therefore not essential but attempting some of them will always improve your result and I will award bonus points for particularly outstanding answers to challenge questions. It is advisable that you only work on 'challenge questions' if you have spare time in any particular day. The reason for this mark scheme is to bring the mark distribution more in line with that which you would expect from an essay question.

Your should hand in three files:

- *A typed document giving your written answers to the questions marked ⋆.*
- *A single file containing all the commented R code that you ran on your own computer to complete the worksheet.*
- *A zip file containing all your results files from the cluster along with the shell script and R code that was run on the cluster to produce them.*

Many of the functions in your R code file will be marked automatically – so be very careful to do the following…

- *Name the file by your username e.g. jrosinde.R*
- *Name all your functions exactly as detailed in the instructions.*
- *Include all your functions in a single file*
- *Put "rm(list=ls())" and "graphics.off()" at the top of your file.*

The deadline for handing in is to be arranged.

Neutral Theory Simulations

1.) 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. You will need to know the species richness of your system so write a function ***'species_richness'*** to measure the species richness in the vector of individuals in your system. For example, ***species_richness(c(1,4,4,5,1,6,1,2))*** should return 5. (Hint: use the 'unique' command) [2 marks]

2.) Write a function ***'initialise_max'*** to generate an initial state for your simulation community with the maximum possible number of species for the community of size J individuals. For example ***initialise_max(7)*** should return a vector { 1 2 3 4 5 6 7 } (Hint: use the 'seq' command) [1 mark]

3.) In this type of simulation, it's important to consider the effect of the initial condition so write another function ***'initialise_min'*** to generate an alternative initial state for your simulation of a certain size with the minimum possible number of species (that's mono-dominance of one species). For example ***initialise_min(4)*** should return a vector { 1 1 1 1 }. [1 mark]

4.) Write a function ***'neutral_step'*** to perform a single step of a simple neutral model simulation without speciation. That is, pick an individual at random according to a uniform distribution from your community vector. This individual will die and be replaced by the offspring of another individual. The reproducing individual is also chosen at random from your community vector according to a uniform distribution; however, the recently deceased individual should not be chosen to reproduce. For example ***neutral_step(c(1,2,3))*** should return one of the following six community states with equal probability:
{ 2 2 3 } when the first individual dies and is replaced by the second's offspring
{ 3 2 3 } when the first individual dies and is replaced by the third's offspring
{ 1 1 3 } when the second individual dies and is replaced by the first's offspring
{ 1 3 3 } when the second individual dies and is replaced by the third's offspring
{ 1 2 1 } when the third individual dies and is replaced by the first's offspring
{ 1 2 2 } when the third individual dies and is replaced by the second's offspring
(Hint: use the 'sample' command) [3 marks]

5.) 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 three inputs: ***initial*** (the initial condition, which also determines the simulation size), ***duration*** (the number of time steps), and ***interval*** (the interval between time steps where you record the species richness of the system). The first value in the time series should be the species richness of the initial condition at time 0. The function should return a list – which enables you to return two objects. The first element in the list will be the time series as a vector and the second element will be the state of the community at the end of the simulation (to allow for further analysis to be carried out on the resulting community, or to allow further simulation). For example ***neutral_time_series (initial = initialise_max(7) , duration = 20 , interval = 2)*** should return a list containing firstly a time series vector of length 11 with the first value being 7 and secondly a community vector of length 7 with values giving species identities. Running the simulation again with a duration of 21 will make no difference to the size of the output time series though the time series itself is stochastic so will probably be different (Hint: use %% and list) [4 marks].

6.) ⋆ Plot 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 10,000 time steps and record the species richness every 10 time steps. Make sure that the x-axis on your plot shows the number of time steps not the number of readings (hint use 'seq'). Include the code you wrote for this question in a function called ***'question_6'*** which should require no inputs to run. What state will the system always converge to if you wait long enough? Why is this? [3 marks]

7.) Write a new function ***'neutral_step_speciation'*** which will perform a step of a neutral model with speciation. In each time step, speciation will replace a dead individual with a new species (with probability ν) otherwise the dead individual is replaced with the offspring of another individual as before in ***'neutral_step'***. You should leave speciation rate, ν , as a parameter in your function. For example, ***'neutral_step_speciation(c(1,2,3),v = 0.2)'*** should behave like ***'neutral_step(c(1,2,3))'***

with probability 0.8, and with probability 0.2 it should instead be equally likely to return any of the following three vectors { 4 2 3 } , { 1 4 3 } , { 1 2 4 } with the number 4 representing the new species. (Hint: use the 'runif' command, also be careful to make sure that any new species really have a unique number assigned to them that has not been used before) [3 marks]

8.) Make a new function ***'neutral_time_series_speciation'*** which uses a neutral simulation with speciation, but otherwise performs in the same way as ***'neutral_time_series'***. The new function should have four inputs, the same three as ***'neutral_time_series'***, and an additional input ***v*** for the speciation rate. The return should be in the same format as before, a list containing a time series vector and a community vector [2 marks].

9.) ⋆ Perform a neutral theory simulation with speciation and plot species richness against time as you above. Use a speciation rate of $\nu = 0.1$, a community size of J = 100 and run your simulation for 10,000 time steps taking readings every 10 time steps. Plot two time series on the same axes in different colours showing how the simulation progresses from two different initial states given by ***initialise_max*** and ***initialise_min***. Include the code you wrote for this question in a function called ***'question_9'*** which should require no inputs to run. 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]

10.) 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 from an input of your community vector. For example ***species_abundance(c(1,5,3,6,5,6,1,1))*** should return 3 2 2 1 (in that order - decreasing). This is because there are 3 of species '1', 2 of species '6', 2 of species '5' and 1 of species '3'. (Hint: use table and sort) [3 marks]

11.) Write a function ***'octaves'*** to bin the abundances of species 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 should tell you have many species have an abundance greater than or equal to 2^{n-1} whilst strictly less than 2^n. E.g. ***octaves(c(100,64,63,5,4,3,2,2,1,1,1,1))*** should return
4 3 2 0 0 1 2 in that order. (Hint: use the log and floor functions) [3 marks]

12.) The simulations are stochastic you will therefore 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 the octave vectors that are not always the same length, so R will not allow you to simply add them, or worse will sum them in a way that you do not intend so will give the wrong answer. Write a function ***'sum_vect'*** 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). (hint: use length) [2 marks]

13.) ⋆ Run a neutral model simulation using the same parameters as in question 9 for a 'burn in' period of 10,000 time steps. Next record the species abundance octave vector. Then repeatedly continue the simulation for a further 1000 time steps, and record the species abundance octave vector again. You should repeat the further simulation and recording process 100 times. Produce a bar chart plot of the average species abundance distribution (as octaves). Include the code you wrote for this question in a function called ***'question_13'*** which should require no inputs to run (hint: it's OK to use a for loop here). [4 marks]

Challenge Question A: ⋆ Plot the mean species richness as a function of time (measured in simulation steps) across a large number of repeat simulations using the same parameters as in question 13. Add a 99.9% confidence interval on the species richness at each point in time. Repeat this for both initial conditions (high initial diversity and low initial diversity). Estimate the number of time steps needed for the system to reach dynamic equilibrium. Include the code you wrote for this question in a function called ***'challenge_A'***.

Challenge Question B: ⋆ Plot a graph showing many averaged time series for a whole range of different initial species richnesses. In each initial community, the species identity should be equally likely to take any species value. Include the code you wrote for this question in a function called ***'challenge_B'***. (Hint: it's OK both here and elsewhere to make additional functions of your own to help make your code neater)

Simulations using HPC

14.) You are going to be running a much larger simulation of the same type that you conducted for your answer to question 13 and with more repeat readings. To do this requires use of high performance computing (HPC) and some adaptation of your R code. You should develop your code for running on the cluster in a separate file to your main answers. (Hint: I did not call the neutral_time_series_speciation function or the question_13 function for this and instead copied the code down and created a new function for running on the cluster.)

First, create a function ***'cluster_run'*** which accepts seven input parameters: speciation_rate , size , wall_time, rand_seed, interval_rich , interval_oct and burn_in_time.

- You need to 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. So your function should set the random number seed as rand_seed

- Your code will need to run for a predefined amount of time – given in minutes by the variable wall_time. Don't keep checking the time as this will waste a lot of calculations. The simplest solution is to check the time every generation – where one generation involves a birth and death for every individual in the system (if size = 1000, one generation is 1000 time steps). (Hint: use the proc.time command)

- Your code should run for the number of generations specified by burn_in_time and store the species richness at intervals of interval_rich during this period. After the number of generations exceeds the burn in time, stop recording the species richness.

- For the entire simulation, until the simulation runs out of time, you should record the species abundances as octaves every interval_oct generations. (Hint: use list)

- You should save your simulation results in a file including the following data: the time_series recorded during the burn_in_time, the list of species abundance

octaves, the state of the community at the end of the simulation, the total amount of time actually consumed on the simulations and all seven of the input parameters for the function. Record all this in a file where the end of the file name is the number of the random seed. This way simulation files will not overwrite one another on the cluster where there will be no nice warning message asking if you want to replace the file or save under a different name! Also this gives you have a sanity check that no pair of simulations were conducted with the same random seed.

- Test your code locally before proceeding further using the same parameters from question 13 and a short time limit of 5-10 minutes.

Now you're ready to write lines of code in your R file around the functions you have written so that when you run the file using the source command you will get the simulation you want.

- Your code should include a new variable ***'iter'*** and should start with the line: ***iter <- as.numeric(Sys.getenv("PBS_ARRAY_INDEX"))***. Your code will be run 100 times in parallel on the cluster, it will be run with iter = 1,2,3, …, 100 so you should use the variable iter in your code to make sure you don't just repeat the identical simulation 100 times.

- Everyone will use the same values for community size J in their simulations (500,1000,2500,5000) but each person will have different speciation rates (to be decided on in class). You will have to select the correct value for J in each parallel simulation based on the value of iter (e.g. J = 500 when iter = 1,5,9,13,….).

- I suggest a time limit of 12 hours for all your jobs (you will put 11.5 hours into your code and tell the cluster 12 hours just in case).

- Use iter to se the seed for your simulation

- I suggest interval_rich = 1, interval_oct = roughly size/10 and burn_in_time = 4*size

- Test your code locally before running it on the cluster

[11 marks for correct test code and correct code for running on the cluster]

15.) Write 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 (also see lecture notes). Run a small job first just to test, then run the full set of jobs to the cluster [11 marks for all your output files shell script code and R code in a zip]

16.) ⋆ While your job is running on the cluster you can write R code to read in and process the output files. Your code should provide a mean species abundance result for each set of parameter values and plot the species abundance distribution as octaves in a multi-panel graph. Only use data of the abundance octaves after the burn in time is up (hint: use the load function on your .rda files). Please hand in all your plots and the actual mean abundance octave numbers along with the exact speciation rate that you used. [11 marks for your graphs and results]

Challenge Question C: ⋆ 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 J. Include the code you wrote for this question in a function called ***'challenge_C'***.

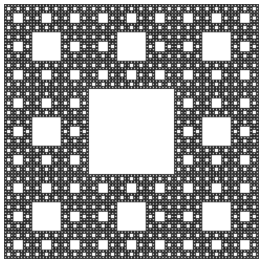
Challenge Question D: ⋆ Conduct further simulations of the same system using coalescence (see the pseudo code below). Check that your results from the cluster agree with those from coalescence and compare the speed of the two approaches. How many CPU hours were used on the coalescence simulation and how many on the cluster to do an equivalent set of simulations? Why were the coalescence simulations so much faster? Include the code you wrote for this question in a function called ***'challenge_D'***. To get a coalescence simulation in R of the neutral model from question 12 as a function of community size J and speciation rate ν.

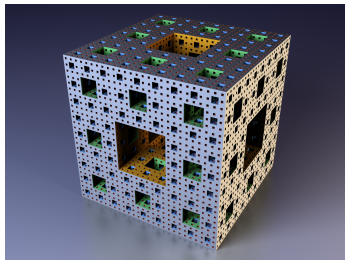
a. Initialise a vector ***lineages*** of length J with 1 as every entry.
b. Initialise an empty vector ***abundances***.
c. Initialise a number $N = J$.
d. Calculate θ, where $\theta = \nu\frac{J-1}{1-\nu}$.
e. Choose an index ***j*** of the vector ***lineages*** at random according to a uniform distribution.
f. Pick a random decimal number ***randnum*** between 0 and 1.
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*** of the vector ***lineages*** at random, but not allowing ***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 through to here.
l. Add ***lineages[0]*** to the end of abundances.
m. END: a vector of simulated species abundances is stored in ***abundances***.

Fractals in nature

17.) ⋆ What are the fractal dimensions of these objects? Show and briefly explain your workings. [4 marks]

Hint: the object on the right looks the same from all six faces and is hollow in the very center; it should have a dimension somewhere between 2 and 3.





18.) ⋆ The chaos game

a. Store the following three points that correspond to coordinate on a graph: a=(0,0), b=(3,4) and c=(4,1).
b. Initialize the point vector X to indicate the point (0,0).
c. Plot a small point on the graph at X. (hint: use cex)
d. Choose one of the three points (a, b or c) at random and move X half way towards whichever of the three points you chose.
e. Write a loop to repeat the code of c. and d. 100 times – what do you see? Now try increasing the number of repeats to 1000 or more. The function that does this should be called ***'chaos_game'*** [8 marks]

Challenge question E: ⋆ Try starting the chaos game from a completely different initial position X what happens now and why? Try plotting the first n steps in a different colour for various values of n to help you answer this. Try starting with the points of an equilateral triangle as a , b and c to produce a classic Sierpinski Gasket.

19.) Create a function ***'turtle'*** in R to draw a line of a given length from a given point (defined as a vector) and in a given direction. So, ***'turtle'*** will have three inputs: start position, direction and length. As well as drawing the line, turtle should return the endpoint of the line it just drew as a vector (Hint: you need to use sin and cos). [2 marks]

20.) Now create another function ***'elbow'*** that calls ***'turtle'*** twice to draw a pair of lines that join together. ***'elbow'*** should accept as an input: the starting point, direction and length of the first line. The second line should start at the end point of the first line, have a direction that is 45 degrees ($\pi/4$ radians) to the right of that of the first line and a length that is 0.95 times the length of the first line. [2 marks]

21.) ⋆ Now copy your 'elbow' function and rename it ***'spiral'***. Spiral will be an iterative function that draws a spiral. Instead of calling ***'turtle'*** twice to draw the first and second lines, spiral should call ***'turtle'*** to draw the first line and then call itself ***'spiral'*** instead of ***'turtle'*** to draw the second line. What happens and now and why? (hint: if you get an error message, try and think about why you're getting it by thinking like a computer – run through the code you just wrote in your own head and see where it gets you) [2 marks]

22.) ⋆ Edit the 'spiral' function calling it ***'spiral_2'***. The edit should make it so that 'spiral 2' will only act if it's called with a line length that's above a certain size (e, a variable that you can experiment with). Now your code will draw a spiral shape on the graph without crashing or giving any error messages. [3 marks for a working spiral plotting function]

23.) ⋆ Now, copy the ***'spiral_2'*** function and rename the copy ***'tree'***. Instead of having ***'tree'*** call itself only once (as ***'spiral_2'*** did), you should have it call itself twice: with directions that are 45 degrees to the right and 45 degrees to the left. Also, make the length of each subsequent call 0.65 times the length of the previous call (instead of 0.95 as it was for drawing the spiral). Don't forget that ***'tree'*** should still call ***'turtle'*** once as well as ***'tree'*** twice. You should get an attractive tree shape as your output plot. [4 marks for a working tree plotting function]

24.) Now copy ***'tree'*** and rename it to ***'fern'***. Change your variables so that whilst one of the two branches goes 45 degrees to the left (as it did in f.) the other goes straight up (instead of to the right). Length multiples should now be 0.38 for the branch going to the left and 0.87 for the branch going straight up (instead of 0.65 for both as it was before). [3 marks]

25.) ⋆ Now copy the function 'fern' and rename it to ***'fern_2'***. This should have an input parameter 'dir' which will decide whether the side branch of the fern goes to the left or right (it's easiest to do this with a variable that takes the value of either -1 or +1). When calling ***'fern_2'*** iteratively from within itself allow the direction of the side branch to alternate by passing on the 'dir' variable that has been multiplied by -1 to revers the direction. You should now get an attractive fern picture. [4 marks for a working fern plotting function]

Challenge question F: ⋆ What do you notice about the image produced and the time the program takes to run as you vary the value of e (the line size threshold)? experiment with the variables and colours to produce other types of fern and tree. Try using multiple colours – bonus points for being imaginative. Include the code you wrote for this question in a function called ***'challenge_F'***, you can create a '***challenge_F2'*** and a ***'challenge_F3'*** if you need to.

Challenge question G: See how small you can make your code answer to question 25 without breaking it. To beat the record you would need to do it in less than 154 characters on one line of code (it would fit in a single text message). If you attempt this challenge, please make your shortened code a separate function named ***'challenge_G'*** from your main answer to question 25 because you'll have to remove all your comments to shorten the code. Last year there was a fair amount of spirited debate about who had done this the best! To be clear, the rules are: your code should work fine even after the workspace has been cleared and should require no libraries, also, no marks should appear on your output axis apart from the lines that make up the fern. Finally, the fern should be of reasonable quality and at least very close in appearance to the fern produced in question 25 – if your code is shorter but doesn't produce the correct result then it doesn't count! This is what it should look like when displayed within in a normal window in R on a normal display, though there could be axes around it.