High Performance Computing Programming Exercises

7th - 11th December 2020 (last updated 7th December 14:30)

James Rosindell (j.rosindell@imperial.ac.uk)

Please answer all the questions by editing and adding to the provided R files. You will hand in your R files, one shell script file, and the set of files produced by a successful run of your code on the HPC system. Specifically please hand in the following files by giving me access to your repo on GitHub.

- jrosinde_HPC_2020_cluster.R (based on proforma)
- jrosinde_HPC_2020_main.R (based on proforma)
- your shell script (.sh) file for running on the cluster
- .e and .o files generated by the cluster for your 100 simulation runs
- .rda output files saved from the cluster for your 100 simulation runs and 1 summary file
- jrosinde_HPC_2020_challengeG.R (if you do this optional question)
- optional additional .rda files or .R files that get sourced by your other files

I've put my username here as a placeholder but you should change this to use your own username. Some aspects of the marking will be automatic, or assisted by automated systems therefore it's important you pay attention to the specifications and follow them precisely.

- Please do not use packages - they should not be needed.
- All files should be in one folder (no sub folders please)
- All challenge questions (except challenge G) in the one HPC_2020_main.R file you hand in.
- Any slow computations for challenge questions (>1 min) should be pre-done and stored in a separate .rda file so that I can run the main function quickly - create new functions wherever you wish e.g. to create the rda files, or to abstract parts of the code.
- Never ever clear the workspace in your main file…
- Always add your username and name the file 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 functions.
- 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.
    - You can do things like source("jrosinde_HPC_2020_main.R")
    - but not e.g. source("Documents/HPC/jrosinde_HPC_2020_main.R")

Please refer to the separate mark scheme for details on the grading process. Formative feedback will be given during the practical sessions and as part of this you will get the chance to 'automark' some aspects of your own work to check that they are OK and to learn about testing. Please be aware that due to the large workload involved in the final marking process your final grades may not be available for a long while after handing in.

Neutral Theory Simulations

These questions build on one another step by step so by the end you will have produced your own individual based 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.

1.) 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: use the 'unique' command) [2 marks]

2.) 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***. For example ***init_community_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 ***init_community_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 with a total number of individuals given by the input number ***size***). For example ***init_community_min(4)*** should return a vector { 1 1 1 1 }. [1 mark]

Now test what you've done….
species_richness(init_community_min(x)) should return 1 (no matter what your value of x was)
species_richness(init_community_max(x)) should return x (no matter what your value of x was)

4.) Write a function ***choose_two***. This function should first choose a random number according to a uniform distribution between 1 and ***max_value*** inclusive of the endpoints. It should also choose a second random number also 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)'*** should return one of the following vectors with equal probability:
{1 2} , {1 3} , {1 4} , {2 1} , {2 3} , {2 4} , {3 1} , {3 2} , {3 4} , {4 1} , {4 2} , {4 3}
(Hint: use the 'sample' command) [2 marks]

5.) Write a function ***neutral_step*** to perform a single step of a simple neutral model simulation, without speciation, on a community vector. You will need to 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 they could be of the same species). For example ***neutral_step(c(10,5,13))*** should return one of the following six community states with equal probability:
{ 5 5 13 } when the first individual dies and is replaced by the second's offspring
{ 13 5 13 } when the first individual dies and is replaced by the third's offspring
{ 10 10 13 } when the second individual dies and is replaced by the first's offspring
{ 10 13 13 } when the second individual dies and is replaced by the third's offspring
{ 10 5 10 } when the third individual dies and is replaced by the first's offspring
{ 10 5 5 } when the third individual dies and is replaced by the second's offspring
(Hint: call your function choose_two, *but don't think of the numbers it returns and being the species identity of the individuals to die and reproduce*, instead think if them as the *indexes* of your ***community*** vector where the individuals that will die and reproduce are stored.) [2 marks]

6.) Write a function ***neutral_generation*** to simulate several neutral_steps on a community so that a generation has passed. 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 generation length. 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). For example, if there are 10 individuals in the system then 5 neutral steps correspond to 5 births and 5 deaths, one generation. Your function should return a vector giving the state of the ***community*** after a generation has passed. (Hint: use ***neutral_step*** when writing this function, and use a loop) [2 marks]

7.) 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, which also determines the simulation size) and ***duration*** (the number generations that 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 firstly a time series vector of length 21 with the first value being 7. (Hint: use your own function ***netural_generation*** from above) [2 marks].

8.) Write a function ***question_8*** to 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 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?"* (Hint: use your own function ***neutral_time_series*** from above, as well as the plot command) [3 marks]

9.) 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 ***speciation_rate***) otherwise the dead individual is replaced with the offspring of another individual as before in ***neutral_step***. You should leave ***speciation_rate*** as an input parameter in your function. For example, ***neutral_step_speciation(c(10,5,13),v = 0.2)*** should behave like ***neutral_step(c(10,5,13))*** with probability 0.8, and with probability 0.2 it should instead be equally likely to return any of the following three vectors…
{ x 5 13 } where x is not 5 or 13
{ 10 x 13 } where x is not 10 or 13
{ 10 5 x } where x is not 5 or 10
(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 - try to think of a simple way to get a number to represent the new species that is different from any of the species numbers you have already. You'll find it easiest to copy and paste the contents of your ***neutral_step*** function and then edit it) [3 marks]

10.) Make a new function ***neutral_generation_speciation*** which uses a neutral simulation with speciation, but otherwise performs in the same way as ***neutral_generation*** so it advances one generation according to the rules of the model. 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 the generation long set of simulation steps. (Hint: You'll be using ***neutral_step_speciation***. It'll be easiest to copy and paste the contents of the ***neutral_generation*** function and then edit it to make this function) [1 mark].

11.) 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 function should have three input parameters: the same two as ***neutral_time_series***, and an additional input ***speciation_rate***. The return should be in the same format as before, a time series vector (Hint: You'll be using ***neutral_generation_speciation***. It'll be easiest to copy and paste the contents of your ***neutral_time_series*** function and then edit it to make this function) [1 mark].

12.) Write a function ***question_12*** to perform a neutral theory simulation with speciation and plot species richness against time as you above. Use a speciation rate of 0.1, a community size of 100 and run your simulation 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 ***question_12*** function should require no inputs to run and should produce the plot. It should 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]

13.) 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]

14.) Write a function ***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 should tell you have many species have an abundance greater than or equal to 2^{n-1} whilst strictly less than 2^{n}. For example, ***octaves(c(100,64,63,5,4,3,2,2,1,1,1,1))*** is asking us to sort 12 species into bins, the first species has an abundance of 100, the second 64, and the 4 rarest species are all represented by one individual only.
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, floor and tabulate functions, if you're not sure what to do try making a table of abundances and the octave that they fall in - then look for a way to generate it using the functions mentioned in this hint) [3 marks]

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. The next question is to help you solve this problem.

15.) 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). (Hint: use length and if) [2 marks]

16.) Write a function ***question_16*** to run a neutral model simulation using the same parameters as in question 12 for a 'burn in' period of 200 generations. Next record the species abundance octave vector. Then repeatedly continue the simulation from where you left off for a further 2000 generations, and record the species abundance octave vector every 20 generations. Your function should produce a bar chart plot of the average

species abundance distribution (as octaves). ***question_16*** 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?"* (Hint: it's OK to use a for loop here, it will also be helpful to use the ***sum_vect*** , ***octaves*** and ***species_abundance*** and ***neutral_generation_speciation*** functions that you already wrote. You will also find the %% function useful). [4 marks]

Challenge Question A: Write a function ***Challenge_A*** to plot the mean species richness as a function of time (measured in generations) across a large number of repeat simulations using the same parameters as in question 16. Add a 97.2% 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 generations needed for the system to reach dynamic equilibrium and mark this on the graph.

Challenge Question B: Write a function ***Challenge_B*** to plot a graph 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 both here and elsewhere to make additional functions of your own to help make your code neater)

Simulations using HPC

You are going to be running a much larger simulation of the same type that you conducted for your answer to question 16 and with more repeat readings. To do this requires use of high performance computing (HPC) and some adaptation of your R code.

17.) Create a function ***cluster_run*** which accepts seven input parameters: ***speciation_rate***, ***size***, ***wall_time***, ***interval_rich***, ***interval_oct***, ***burn_in_generations*** and ***output_file_name***. [6 marks] This function should:

- Start with a community with size given by the input ***size*** and minimal diversity.
- Apply neutral generations with a speciation rate given by ***speciation_rate*** for a predefined amount of time ***wall_time*** measured in *minutes*. (Hint: if you're not sure where to start get a timer working on its own first and 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, stop recording the species richness. So, suppose ***intererval_rich*** is 2, this means save the species richness every other generation (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).
- 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 with name given by the input ***output_file_name*** 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 simulation and all six of the input parameters for the function (obviously you don't need to store a reminder of what the file name is inside the file itself!)

- Test your code locally before proceeding further using the same parameters from question 16 and a short time limit of 5-10 minutes. For example… ***cluster_run(speciation_rate = 0.1, size=100, wall_time=10, interval_rich=1, interval_oct=10, burn_in_generations=200 and output_file_name="my_test_file_1.rda")*** 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 from to check you've got everything you need as described above. (Hint: use the save command, and don't call your outputs by the same names as your functions - otherwise this will save the functions and not the outputs.)

18.) Use a new R file for running on the cluster based on the provided proforma. 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. Here's what the code needs to do (in this order)…

a) Clear workspace and turn off graphics
b) Load all the functions you need by sourcing your main R file
c) 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 1,2,3,…. 100 for testing locally. The last thing you need to do is uncomment the line and remove your setting of iter so that now the cluster chooses what iter is equal to. 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.
d) 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 ***iter***
e) Everyone will use the same values for community size in their simulations (500,1000,2500,5000) but each person will have different speciation rates - handed out separately. You will have to select the correct value for community in each parallel simulation based on the value of ***iter*** (you could do this any way you like e.g. ***size*** = 500 when ***iter*** = 1,5,9,13,…. And ***size*** = 1000 when ***iter*** = 2,6,10,14,….. or ***size*** = 500 when 1 <= ***iter*** <= 25). Think if this as a 'to do list' with 100 items on that will in the end be done in parallel.
f) Create a filename to store your results - the end of the file name should be a number given by ***iter*** (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. (Hint: use the paste command)
g) Call the ***cluster_run*** function, which will actually do the simulation and save the results. Use 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 ***interval_rich*** = 1, ***interval_oct*** = ***size***/10 (roughly) and ***burn_in_generations*** = 8*****size***

Test your code locally for a shorter length of time before running it on the cluster. Then test on the cluster with iter of 1,2,3 only before doing the rest (use -J 1-3 as in the lecture notes) [6 marks for correct test code and correct code for running on the cluster]

19.) 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 [10 marks for all your output files shell script code and R code in a zip]

20.) Plot the results from your cluster run. While your job is running on the cluster you can write an R function ***process_cluster_results*** to read in and process the output files. Your code should read in all your output files (assuming 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 (post burn in) data for each abundance octave and for each simulation size (500,1000,2500,5000). 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 1000, then 2500, then 5000) (Hint: use the load function on your .rda files and use ***sum_vect***). Next write an R function **plot_cluster_results** that should provide four bar graphs in a multi-panel graph (one for each simulation ***size***) each showing a mean species abundance octave result from all your simulation runs of that size and it should return the list of the data it just plotted [10 marks for your graphs and correct results]

Challenge Question C: Write a function ***Challenge_C*** 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, like ***process_cluster_results***.

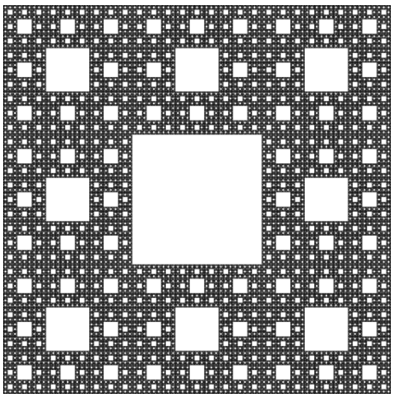
Challenge Question D: Write a function called ***Challenge_D*** to conduct further simulations of the same system using coalescence (see the pseudo code below). Check that your results from the cluster agree with those from coalescence (plot a graph to show this) and compare the speed of the two approaches. 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?"*

To get a coalescence simulation in R of the neutral model from question 16 as a function of community ***size*** (J) and ***speciation_rate*** (ν).

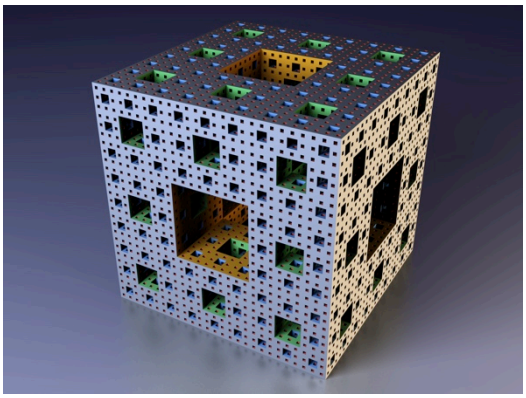
a. Initialise a vector ***lineages*** of length J with 1 as every entry.
b. Initialise an empty vector ***abundances***.
c. Initialise a number $N = J$.
d. Calculate θ, where $\theta = \nu \frac{J-1}{1-\nu}$.
e. Choose an index ***j*** of the vector ***lineages*** at random according to a uniform distribution.
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 the only element left in ***lineages*** to the end of ***abundances***.
m. END: a vector of simulated species abundances is stored in ***abundances***.

Fractals in nature

21.) Write a function ***question_21*** to return a list giving the fractal dimension of the following object as the first entry and a string explaining your workings as the second entry. [2 marks]



22.) Write a function ***question_22*** to return a list giving the fractal dimension of the following object as the first entry and a string explaining your workings as the second entry. [2 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.

23.) Write a function ***chaos_game*** which should perform the following algorithm

a. Store the following three points that correspond to coordinates 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 **very 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? return your answer as plain text* – Now try increasing the number of repeats to 1000 or more but don't leave it too high for marking, it should complete in at most 30 seconds. [8 marks]

Challenge question E: Write a function ***Challenge_E*** which tries starting the chaos game from a completely different initial position X *what happens now and why? return your*

answer as plain text. 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. If you're feeling super enthusiastic you could have more than 3 points and a distance of movement different from a half towards the next point in which case you'd be producing a multi panel graph to contain all your findings.

24.) 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*** (measured in radians, not degrees, direction zero means left to right, positive direction moves around anticlockwise) and ***length***. As well as drawing the line, turtle should return the endpoint of the line it just drew as a vector. *Turtle should not open the plot it should just draw the line on an already open plot*, this is because in a moment you are going to use successive calls of the function to draw things and you want all the lines to be on the same axes. *So, you will need to open the plot with a command outside of turtle when you want to test it.* (Hint: you need to use sin and cos). [2 marks]

25.) Now create another function ***elbow*** that calls ***turtle*** twice to draw a pair of lines that join together with a given angle between them. As with turtle, it should not open a new plot and should assume a suitable plot is already open. The function ***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]

26.) Now copy and paste 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. As with turtle, it should not open a new plot and should assume a suitable plot is already open. *What happens and now and why? return your answer from the function as plain text*. (Hint: if you get an error message that might be what's expected! Try and think about why you're getting it - think like a computer – run through the code you just wrote in your own head and see where it gets you) [2 marks]

27.) Edit the ***spiral*** function to get it to work. Now make a new function ***draw_spiral***. That will open a new plot, draw a spiral and then return the same text answer from the ***spiral*** function (try doing this so that you don't have to include two copies of the text answer in your code). (Hint: the edit should make it so that spiral will only act if it's called with a line length that's above a certain threshold [3 marks]

28.) Now, copy and paste the contents of the (fixed) ***spiral*** function and rename the copy ***tree***. Instead of having ***tree*** call itself only once (as ***spiral*** 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. Because the ***tree*** function should not open a new plot, use another function ***draw_tree*** to do this and to call tree with suitable parameters so that just running ***draw_tree*** does everything to make a nice tree plot. Please choose parameters so that the plot will draw in less than 30 seconds. [4 marks]

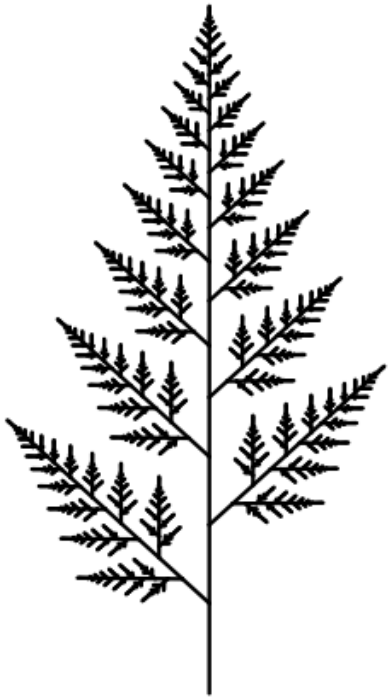
29.) Now write functions ***fern*** and ***draw_fern*** based on **tree** and **draw_tree**. Change your variables so that whilst one of the two branches goes 45 degrees to the left (as it did in

f.) the other goes straight on (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). Please choose parameters so that the plot will draw in less than 30 seconds. [3 marks]

30.) Now write functions ***fern2*** and ***draw_fern2*** based on ***fern*** and ***draw_fern***. This should have an additional 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. (Hint: spot the difference, look very carefully at your fern to check that it does look the same as the example in this worksheet below, you will not get full marks unless they are really the same) [4 marks]

Challenge question F: write a function ***Challenge_F*** which should return your plain text answer to the question "*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 as an additional (possibly multi panel) plot. Try using multiple colours – bonus points for being imaginative. Please keep the amount of time it takes to plot this under 1 minute.

Challenge question G: See how small you can make your code answer to question 30 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 file - you'll have to remove all your comments to shorten the code. In the past there has been 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 29 – 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.



List of thanks

If you spot any kind of mistake - I'll correct it and (if you wish) add your name to the list of thanks for following years. Apologies to people from previous years who helped before I had the idea of formally doing this!