BIOL 3551: Population modelling in R

You have, previously, been introduced to the R environment and to some of the basic commands in R (getting data in or out, conducting simple mathematical operations, and using loops to iterate procedures). The purpose of this document is to show you that, without much additional knowledge, you can turn mathematical models into computer code. I hope this process will reinforce some of the ideas covered in the population modelling lectures.

1. RStudio

For this practical, we will work in RStudio (not least, because I can't find R on university PCs any more). RStudio is very similar to R but provides a slightly more user-friendly interface. It can be downloaded for free from https://www.rstudio.com/products/rstudio/download3/. On university PCs, RStudio can be found under "All apps", looking for those that are "Academic. Mathematical Sciences".

RStudio looks similar to R but with slight differences – see screenshot in last handout. A particular advantage of RStudio is the panel in the top right, which shows the functions and variables currently in R's "memory". In R itself, to view the contents of a data frame (which, as you might recall, is a bit like a spreadsheet), you would have to use the command **View(df)**. [Remember: I will use bold to illustrate code that you can type into the R console or into your script file. The name 'df' in the example I just used is just that – an example; it is intended to represent the name of any data frame that you might wish to view]. By contrast, in RStudio, you can simply click on df in the top right panel.

RStudio is also nice because you can view several useful windows all at once, as the default (whereas R opens new windows for different types of information, and can get a bit messy). For example, in RStudio you can view your script in the top left, your console in the bottom left, your variables and functions in the top right, and any graphics you produce or help files you view, in the bottom right.

Remember that, as a general rule of thumb, you should enter all commands into a script file and save it (somewhere sensible) at regular intervals. That will allow you to keep a copy of what you did and to view it again whenever you need to.

2. Simple population models

Although your previous workshop was not a comprehensive introduction to R, you are already well-equipped to code simple population models. As an example, consider the discrete time model:

$$N_{t+1} = N_t + RN_t$$

This can be run using the code:

```
Years <- 100                    # sets the duration of the simulation
R <- 0.2                        # sets the population's growth rate
N <- c(10,rep(0,Years-1))       # defines a vector to store the population sizes (an initial
                                # size of 10, followed by 0 repeated for every other year)
```

```
for (t in 2:Years) N[t] <- N[t-1]+N[t-1]*R# loops through the remaining years of the simulation,
                                          # calculating the new population size each time
plot(1:Years,N, bty= "l", xlab= "Year", ylab= "Population size", type= "l")
```

Notice that 'Years' is defined as a constant. It is good practice to use constants in code. You will see that 'Years' was used in 3 different lines of code. If you want to change a constant (such as the duration of the simulation) it is much easier to have it defined in a single place than to pick through the code, changing each line that depends on that value. A similar rationale can be applied to defining *R* as a constant, even though – in this model – *R* is only used once more. For further information on the rep() function, you can use **?rep**.

This code describes a population that grows exponentially at a rate of 20% per year. Try changing the variable *R* to see the effect on population growth. This model lacks almost any aspect of realism; it lacks ecology. We can improve the model by introducing a carrying capacity, *K*. For example:

```
Years <- 100                              # sets the duration of the simulation
R <- 0.2                                  # sets the population's maximum growth rate
K <- 100                                  # sets the carrying capacity
N <- c(10,rep(0,Years-1))                 # defines a vector to store the population sizes (an initial
                                          # size of 10, followed by 0 repeated for every other year)
for (t in 2:Years) N[t] <- N[t-1]+N[t-1]*R*(1 – N[t-1]/K)  # loops through the remaining years of the
                                                            # simulation, calculating the new
                                                            # population size each time
plot(1:Years,N, bty= "l", xlab= "Year", ylab= "Population size", type= "l")
```

This is equivalent to the density dependent model:

$$N_{t+1} = N_t + R\left(1 - \frac{N_t}{K}\right)N_t$$

3. Density dependence and population cycles

For heuristic purposes, we're going to look at a result that was very fashionable for a while in the latter half of the last century. Specifically, we're going to take a look at the way that population growth rate affects population stability. To start with, try adjusting the value of *R*, the population growth rate. Try numbers in the range $0.05 \leq R \leq 3.0$. What do you notice about the stability of the population? How would you describe the population when $R = 3.0$? Why do you think these things can happen simply as a consequence of adjusting *R*?

We can examine the transition from a steady approach to equilibrium to complete chaos by reproducing (Lord) Bob May's famous bifurcation plot. To do that, we want to know how many stable population sizes there are after the population reaches a semi-stable state (i.e. after it is no longer affected by the specific population size at which we started the simulation. To make sure that we only consider those population sizes, we'll give the population slightly longer to settle down (by increasing Years to 250) and we'll only record the population sizes that the population exhibits over the last 100 years of each simulation. What we want is to be able to plot these 'stable' population sizes against the population's growth rate. If the population is at perfect equilibrium, every population size over the last 100 years will be the same and we will get only one point (or 100 points in exactly the same place) in relation to growth rate. If the population oscillates between what are known as "stable limit cycles", we'll get a number of points equal to the number of population sizes between which the population is oscillating. This is probably most easily demonstrated by running the model.

To simulate this, we need to loop through a range of values of *R*, funning our simulation for each one and recording, again for each one, the population sizes in the last 100 years of the simulation. We could use a loop to do this (as discussed in the previous workshop). However, it turns out that there is a more efficient way of iterating things in R, and this is with the **apply** family of functions. First, we need to set up the vector of population growth rates:

Rs <- seq(1.5,3,0.01)

Next, we want to do something with each value in this list (or, more properly, vector) of numbers. To do that, I'm going to introduce two new concepts: functions and the **lapply** statement. I will start with the **lapply** statement.

The purpose of **lapply** is to apply some set of commands or instructions to every item in a list. Students sometimes struggle with how this works. I have pondered how to explain this with reference to the every day and I decided to illustrate it with reference to a theme in this life that I live of domestic drudgery. Consider washing dishes. In the average kitchen, there will be a formidable tower of items to the left of the sink. You want to take each one, subject it to various procedures (to get it clean and soap free), then place it on the draining rack on the right of the sink. Now, consider that even academics can dream. Suppose I wanted to instruct someone else to do the washing up for me whilst I put my feet up elsewhere. Suppose, further, that they had limited experience of such a task. I could say to them that I want them to take each item in turn from the left of the sink, plunge it into soapy water, scrub it until it's clean, plunge it back into soapy water, check it's cleanliness, rinse it and place it on the rack to the right. Now, I don't want to repeat these instructions for every new item. Rather, I want them to apply the same instructions to each item. Consequently, I would say that **items.to.the.left** is the complete list of items to the left of the sink, whilst **items.on.draining.rack** is what I want to see when I return to the kitchen. If I wanted to write this in R code, I would write:

```
items.on.draining.rack <- lapply(items.to.the.left, function(item){
        plunge.into.soapy.water(item)
        scrub(item)
        plunge.into.soapy.water(item)
        check.cleanliness(item)
        rinse(item)
        return(clean.item)
})
```

Some of you might be wondering if I've taken leave of my senses. However, I will persevere. Notice that: (1) the set of instructions for each item is the same and is enclosed in curly brackets {} to ensure that every step within the curly brackets is applied to every item; (2) we don't have to refer to each item by name because, in the first line, we say "**function(item)**", which means that each will be referred to as "item" in turn; and (3) by using "**return(clean.item)**" at the end, we ensure that each clean item is passed back from the function to the object defined by this **lapply** statement (which is **items.on.draining.rack**).

The more perspicacious among you might notice that "**check.cleanliness(item)**" is a slightly pointless gesture, because we don't do anything about it if it isn't clean. Technically, therefore, we should probably use:

```
items.on.draining.rack <- lapply(items.to.the.left, function(item){
        while(item == dirty){
                plunge.into.soapy.water(item)
                scrub(item)
                plunge.into.soapy.water(item)
                check.cleanliness(item)
                }
        rinse(item)
        return(clean.item)
})
```

- however, I don't want to get too carried away with this analogy. The important point is that this lapply statement should work, whatever is to the left of the sink, regardless of whether **items.to.the.left** = c("wine.glass","cheese.knife","side.plate"), or **items.to.the.left** = c("pint.glass","grill.pan","chip.pan").

So, what I have illustrated here is 3 concepts: the **lapply** statement to apply a set of instructions to every item in a list or vector; the use of a local (anonymous) function to describe the set of things I want to do with items in that list); and the return() command to ensure that the outcome of the function is passed back to the 'global environment'.

Well, now that I, myself, have returned to the global environment, I should probably refocus on Bob May's bifurcation plot. To produce that, I would write the following:

```
Years <- 250                # sets the duration of the simulation
K <- 100                    # sets the carrying capacity

stable.sizes <- lapply(Rs,function(R){
        N <- c(10,rep(0,Years-1))          # defines a vector to store the population sizes (an initial
                                           # size of 10, followed by 0 repeated for every other year)
        for (t in 2:Years) N[t] <- N[t-1]+N[t-1]*R*(1 - N[t-1]/K)   # loops through the remaining
                                                        # years of the simulation, calculating the
                                                        # new population size each time
        sizes <- N[(Years-99):Years]                    # record only the last 100 years' worth of
                                                        # population sizes
        return(data.frame(PGR=R,Pop.sizes=sizes))       # store those population sizes as a data
                                                        # frame within the "stable.sizes" object;
                                                        # the first column contains the population
                                                        # growth rate
})
stable.sizes <- do.call('rbind',stable.sizes)           # bind all the data frames together as a
                                                        # single object
stable.sizes$colour <- 'red'                            # make another column in the data frame,
                                                        # with the word 'red' in each row
stable.sizes$colour[stable.sizes$Pop.sizes < K] <- 'blue' # change the colour column to be 'blue' for
                                                        # all rows in which the population sizes are
                                                        # less than the carrying capacity

par(las=1)          # change the graphical parameter that controls axis tick-mark label orientation
                    # and plot the graph:
plot(stable.sizes$PGR,stable.sizes$Pop.sizes,cex=0.3,pch=19,col=stable.sizes$colour,
                              xlab='Population growth rate',ylab='Stable sizes')
```

Here, the list of items that I want to do something with is the vector of population growth rates that we defined earlier (**Rs**). The things we're going to do with every item in the list are given by the instructions within the curly brackets following function(R). Notice, therefore, that each value in the vector **Rs** will be referred to as 'R' whilst it is our focus. Finally, the line "**return(data.frame(PGR=R,Pop.sizes=sizes))**" defines what will be returned to the global environment after each iteration of the function (which, instead of being a clean dish, will be a data frame to be stored in **stable.sizes**).

This creates a pretty plot that, as I mentioned earlier, caused much excitement among ecologists. Could chaos and total unpredictability be a natural and inevitable feature of even simple ecological processes? This continued to cause much excitement until people pointed out that virtually nothing (except, maybe, a few irruptive species of invertebrate) was likely to have population growth rates of the magnitude required to cause chaos. With the formulation I've used, R = 0 gives stability and R = 1 implies a population that is doubling in each time step. Our bifurcation plot doesn't even start until R = 1.5, and chaos isn't seen until R = 2.5. This would only apply to populations that increase by 250% in each time step. After they'd realised that, ecologists calmed down a bit.

4. Matrices, simulations and population characteristics

Up to this point, we've been focusing on tools to allow you to use R effectively and to run simple models of population growth in which population size is treated as a scalar (a single number). In this section, I'll focus on the kinds of operations required to interrogate matrices for the sorts of purposes discussed in conservation biology lectures. You might like to save your script, close it down and begin a new one.

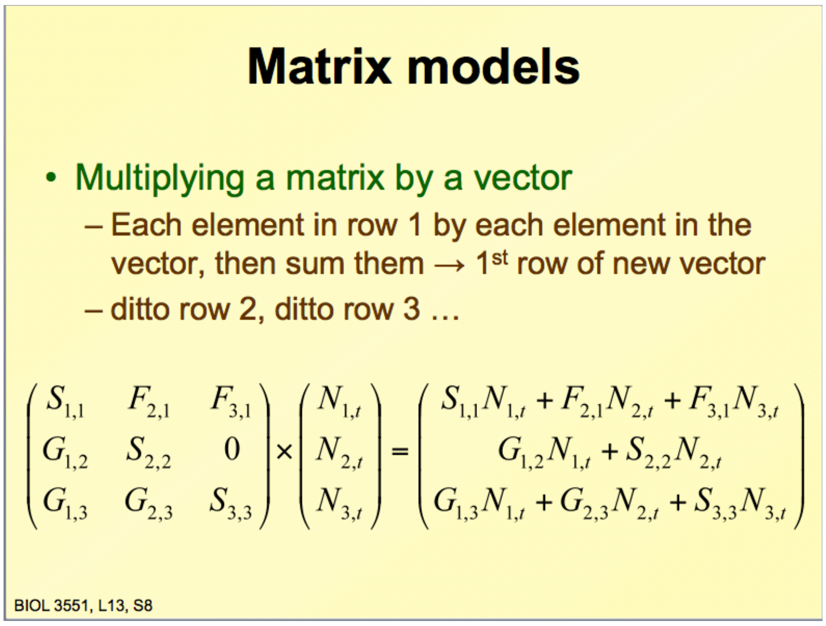
It's good practice to begin a new script by clearing out R's memory and setting the working directory. You can do that with the following commands:

```
rm(list=ls())
setwd("C:/My documents/R")   # or whatever represents a good directory for you to work in
```

In a previous workshop, we saw how values can be entered in to a matrix:

```
M <- matrix(c(0,0.024,0,52,0.08,0.25,279.5,0,0.43),nrow=3)
```

This produces a matrix called M with parameter values that make it the transition matrix for the common frog. This is where R really comes into its own. Recall that matrix multiplication requires a number of operations – e.g.



Matrix models

- Multiplying a matrix by a vector
  - Each element in row 1 by each element in the vector, then sum them → 1st row of new vector
  - ditto row 2, ditto row 3 …

$$\begin{pmatrix} S_{1,1} & F_{2,1} & F_{3,1} \\ G_{1,2} & S_{2,2} & 0 \\ G_{1,3} & G_{2,3} & S_{3,3} \end{pmatrix} \times \begin{pmatrix} N_{1,t} \\ N_{2,t} \\ N_{3,t} \end{pmatrix} = \begin{pmatrix} S_{1,1}N_{1,t} + F_{2,1}N_{2,t} + F_{3,1}N_{3,t} \\ G_{1,2}N_{1,t} + S_{2,2}N_{2,t} \\ G_{1,3}N_{1,t} + G_{2,3}N_{2,t} + S_{3,3}N_{3,t} \end{pmatrix}$$

BIOL 3551, L13, S8

To do this in R, we need only specify the vector of population sizes, and then use one operation to multiple it by our matrix – e.g.

```
N<-c(10,10,10)          # note that this can also be written N<-rep(10,3)
N<-M%*%N                # NB. Matrix multiplication is sensitive to order, unlike normal
                        # multiplication (i.e. M%*%N ≠ N%*%M)
```

We can use the same kind of notation that we saw earlier to run this as a population model:

```
# set the duration of the simulation
Years <-15
# define the transition matrix
M <- matrix(c(0,0.024,0,52,0.08,0.25,279.5,0,0.43),nrow=3)
# define a matrix with 3 rows and 'Years' columns to store population sizes.  Make the first 3
# numbers (to fill the first column) 10 (the initial population sizes) and all subsequent numbers
# zero (because we'll calculate what those numbers should be in subsequent steps)
N <-matrix( c(rep(10,3), rep(0,(Years-1)*3) ), nrow=3)
# compute population sizes
for (t in 2:Years) N[,t] <- M%*%N[,t-1]
# plot the outcome
par(las=1)
plot(c(0,Years),c(1,max(N)),xlab="Time",ylab="Number",type="n",log="y",bty="l")
cols=c("black","brown","darkgreen")
for (L in 1:3) lines(1:Years,N[L,],col=cols[L],lwd=3)
```

Several points are worth making about the code above. First, although it looks cumbersome, much of the text is comment. Second, it contains a number of new graphical tricks. The first is setting a graphical parameter using the par() function. par(las=1) ensures that all axis text is horizontal (which makes the numbers easier to read). The next graphical trick is using the plot() function to set up the plot but with no data in it. That is achieved by setting type="n" in the plot function, so that the points are suppressed. The reason is that we'd like the axes to have a scale appropriate to the maximum value that they'll have to plot. We therefore tell R to plot some 'null' points with x-values at zero and Years (c(0,Years), which takes in the full range of the x-axis) and y-values at 1 and the maximum value in the population size matrix (c(1,max(N)), which takes in the full range of the y-axis). Other tricks include logging the y-axis (using log="y") and looping through the three rows of the population size matrix to plot each row (N[L,]) against the full range of time steps (1:Years). We can add a legend to this graph using the legend() function:

```
# add a legend
labs <- c("Pre-juveniles","Juveniles","Adults")
legend( x=0,          # an x coordinate for the legend's top left corner
        y=3000000,    # a y coordinate for the legend's top left corner
        legend=labs,  # tells R that the text for the legend is in the vector 'labs'
        lty=1, lwd=3, # specifies the line characteristics
        col=cols,     # tells R that the colours for the legend are in the vector 'cols'
        box.lty=0)    # suppresses a surrounding box
```

Look at the graph and consider its meaning. Notice that, after some initial jitters, the population settles down so that the numbers in each stage class are increasing at the same exponential rate and, hence, the whole population is growing at an exponential rate. This is the 'asymptotic growth rate', λ, and can be estimated by looking at the ratio of the population size in its final year to the population size in its penultimate year:

```
lambda.est = sum(N[,Years])/sum(N[,Years-1])
round(lambda.est,2)                # reduces the number of decimal places to 2
```

The resultant number (λ=1.68) suggests that populations of the common frog have the potential to grow very rapidly (at 68% per year). Of course, this models the population without density dependence or stochasticity, so is very ecologically naïve. For this reason, we do not typically use matrix models for forecasting, only for understanding the characteristics of the population *under the current circumstances*. In that context, λ tells us only the current trajectory of the population: $\lambda < 1$ implies a declining population, $\lambda = 1$ is a stable population, and $\lambda > 1$ is a growing population.

Another thing to notice is that, after the original jitters, the trajectories of each stage class are parallel. This means that dividing the number in one stage class by the other would produce a steady ratio. This is known as the stable stage distribution (SSD): the distribution of individuals between stage classes when the population is growing at a steady rate. We can estimate the SSD by finding out the ratio of individuals in the 3 classes in the final year of our simulation:

```
SSD.est = N[,Years]/sum(N[,Years])
round(SSD.est,3)
```

Notice that the population is dominated by pre-juveniles. The 'reproductive value' of the three stage classes could also be estimated by simulation. For example, we could try sequentially performing a simulation, starting with just 1 individual in one age class. The total population at the end of the simulation would tell us how many individuals the initial individual contributed to the future population size. The ratio of these contributions tells us the relative 'reproductive values' of the three stages:

```
Final <- numeric(3)  # this is one way to declare an empty vector of numbers
for (s in 1:3) {
        N[,1] <- c(0,0,0)
        N[s,1] = 1
        for (t in 2:Years) N[,t] = M%*%N[,t-1]
        Final[s] = sum(N[,Years])
        }
RV.est <- Final/Final[1]
round(RV.est,2)
```

5. Population characteristics and matrix properties

The code discussed in section 12 is straightforward. In fact, however, it is even simpler to find out what we want to know about the present state of a population according to its transition matrix. This is because matrices have standard properties (known as eigenvalues and eigenvectors) that tell us almost everything we might otherwise need to find out by simulation. All of these numbers can be derived using the function eigen(). You don't need to know what eigenvalues or eigenvectors are in order to make use of them. All you need to know is how they relate to population characteristics. For example, the asymptotic growth rate of the population, λ, can be determined from the 'dominant eigenvalue' of the matrix. This is extracted using the code:

```
lambda.true<-eigen(M)$values[1]
```

In fact, this number is a complex number but we are only interested in its rational part. To get that, we can use the function Re(). We can also limit the number of decimal places using the function round() – i.e.

```
( lambda.true<-round(Re(eigen(M)$values[1]),2) )
```

In the same way, the stable stage distribution can be found from the first column of eigenvectors produced by the eigen() function. We can 'normalise' these values, so that they add up to 1.

```
SSD.true <- eigen(M)$vectors[,1]/sum(eigen(M)$vectors[,1])
( SSD.true<-round(Re(SSD.true),3) )
```

Remember: by placing the second line of code in brackets, we see the outcome of the computation at the same time as doing the calculation. For what it's worth, the left eigenvectors of a matrix M are the same as the eigenvectors of its transpose, t(M). Thus, we can find the reproductive values of each stage using:

```
RV.true <- eigen(t(M))$vectors[,1]
RV.true <- Re(RV.true/RV.true[1])        # this makes all reproductive values relative to that of the
                                          # 1st stage class
round(RV.true,3)
```

6. Sensitivities, elasticities and other uses of matrix models

Sensitivities can be calculated using:

```
M<-matrix(c(0,0.024,0,52,0.08,0.25,279.5,0,0.43),nrow=3)     # the matrix
w <- eigen(M)$vectors[,1]/sum(eigen(M)$vectors[,1])            # the SSD
v <- eigen(t(M))$vectors[,1]/eigen(t(M))$vectors[1,1]          # reproductive values
s = 3                                                          # number of life stages
# Summed product of RV and SSD for each stage class:
v.w <- sum(v*w)
# Now calculate sensitivity for each matrix element:
sensitivity <- matrix(nrow=s,ncol=s,0)
for (i in 1:s)
        for (j in 1:s)
                sensitivity[i,j] = v[i]*w[j]/v.w
( sensitivity<-round(Re(sensitivity),4) )
```

Elasticities can then be computed by:

Lambda <- eigen(M)$values[1]
elasticity <- sensitivity * M / Lambda
(elasticity <- round(Re(elasticity),4))

If you have typed in this code correctly, you should get something like:

Sensitivities:

```
     [,1]    [,2]    [,3]
[1,]  0.3628 0.0055 0.0011
[2,] 25.0531 0.3812 0.0776
[3,] 82.6139 1.2570 0.2560
```

Elasticities:

```
     [,1]    [,2]    [,3]
[1,] 0.0000 0.1726 0.1855
[2,] 0.3628 0.0184 0.0000
[3,] 0.0000 0.1896 0.0664
```

Some questions to consider:

1. What are the management implications of large sensitivities and elasticities?
2. Why is the sensitivity[3,1] large but the corresponding elasticity is zero?
3. In reality, what else should dictate management efforts, in addition to the elasticities of the different vital rates (i.e. elements of the transition matrix)?
4. If we had some indication of uncertainty in vital rates, what technique might we use to explore the implications of that uncertainty and the sensitivity of λ to each vital rate *within the bounds of its uncertainty*?

You should consider the second half of the lecture on matrix models to help you to answer these questions.