**Introduction to R for modelling**

Guillaume Latombe

*Note: In the following, we will use for loops because they are easier to understand, but see the optional questions at the end of the document.*

Now that we know some basics in R, let’s make our first simple population models. We will use a density-dependent logistic growth, using discrete and continuous models, as seen in class:

Eq.1

Eq. 2

**Discrete model (logistic map)**

We will first execute a series of commands to get you through the logic of the process, before actually implementing the model with a “for” loop:

* Initialise a numeric vector P of size 500 for the population, using the function numeric()
* Set the first value of P (i.e P[1]) to 10 individuals. This is your population at time t=0, i.e. your initial population size.
* Set the parameters r and K to 0.1 and 100
* Using Equation 1 above, compute P[2], the population at time t=1, from P[1].
* Using Equation 1 above, compute P[3], the population at time t=2, from P[2].
* Using Equation 1 above, compute P[4], the population at time t=3, from P[3].

Obviously if we want to simulate 500 time steps, we will not write 500 lines of code Instead, we will use a “for” loop:

* Set a for loop for 499 time steps (from 2 to 500).
* Within the for loop, use the same equation you used to compute P[2], P[3] and P[4], but this time, instead of specifying the index by an actual value, you will use the variable you iterate in the for loop.

Plot the output using the plot() function, with time on the x axis and population size on the y axis. Note that you can play with the options xlim, ylim and log in plot to improve the plot (type ?plot in the command window). Experiment.

Plot also P(t+1) vs P(t).

You can make two plots:

* one in which you only plot the points (this is the default setting when calling plot()
* one in which you plot it as a continuous line: plot(…, type = “l”)

You can also plot both in one plot, using the function lines() after calling plot. Check the help file for lines()to see what it does.

We will also use the function lines() to add a curve showing the whole possible set of [P(t),P(t+1)] combinations. The equivalent to draw points is points(). We cannot exactly show the whole possible set of [P(t),P(t+1)] combinations, but we can compute a large number of these combinations and plot the output. To do so, we simply need to compute Equation 1 for many values of P(t):

Pt <- seq(0,150,0.1) # that creates a vector from 0 to 150 with an increment of 0.1

Ptt <- r\*(1-Pt/K)\*Pt+Pt

lines(Pt, Ptt, col = “blue”)

lines(c(1,150), c(1,150), col = “red”) # that draws the diagonal, as we saw in the lecture, to show the equilibrium

lines(c(P[1],rep(P[2:99],each=2)), y = c(rep(P[2:99],each=2),P[100]),col="green") #that creates the path using the diagonal

Change the values of P(0), r and K. What do you observe? In particular, increase the value of r until you see more complex population behaviour, until reaching a chaotic behaviour. Does it change anything for the plot showing P(t+1) vs P(t)?

**Continuous model**

Here we will see how to use a function to find a continuous solution to the logistic growth equation. Start by installing the following packages: deSolve, tidyverse, and ggplot2.

deSolve is a package that contains the function ode() for solving the differential equation. It applies complex mathematical algorithms to do so, that you do not need to understand in detail at this stage.

tidyverse contains some useful functions to manipulate data.

ggplot2 makes beautiful graphs (you will see these in more details in the data visualization lecture in the Professional skills course).

Access the help file for the ode() function. We can see we need to define four parameters: y, times, func, parms (we don’t need to look at the other parameters at this stage, we will just use the default values).

* y – the initial (state) values for the ODE system, a vector. If y has a name attribute, the names will be used to label the output matrix.

Here there is only one variable with an initial value: P. We will therefore define:

state <- c(P=10)

* times – time sequence for which output is wanted; the first value of times must be the initial time.

Let’s simulate 500 time steps, and get values for every 0.1 interval:

times <- seq(0,100,by=0.01)

* parms – parameters passed to func.

These are the parameters r and K. We will store them in a vector:

parameters <- c(r=0.1, K=1000)

* func – either an R-function that computes the values of the derivatives in the ODE system (the model definition) at time t, or a character string giving the name of a compiled function in a dynamically loaded shared library.

If func is an R-function, it must be defined as: func <- function(t, y, parms,...). t is the current time point in the integration, y is the current estimate of the variables in the ODE system. If the initial values y has a names attribute, the names will be available inside func. parms is a vector or list of parameters; ... (optional) are any other arguments passed to the function.

The return value of func should be a list, whose first element is a vector containing the derivatives of y with respect to time, and whose next elements are global values that are required at each point in times. The derivatives must be specified in the same order as the state variables y.

func is a bit more complicated to define, but basically it is just an R function that implements the differential equation above:

LG <- function(t,state,parameters){ ##logistic grown function, that takes a set of parameter values, initial conditions and a time sequence

with(as.list(c(state, parameters)),{ ##"with" is a function that allows us to use the variable names directly - it looks for r, K and P in state and parameters

dP <- r\*(1-P/K)\*P ##this is our logistic equation governing the rate of change of P

return(list(dP)) ## return the rate of change - it needs to be a list

}) # end with(as.list ...

}

From there, we just need to call function ode with these four elements:

out <- ode(y=state, times = times, func = LG, parms = parameters)

Look at out in your workspace. What kind of element is it?

Some functions create elements that are in a specific format (here it is a deSolve object). This format is useful when calling other functions from the same package, but sometimes quite annoying for plotting the results. We therefore need to convert it to a more convenient data frame:

out.df <- data.frame(out)

We can now plot the output:

plot(out.df,type="l")

or if you want something fancier using ggplot:

ggplot(data = out.df)+

geom\_line(mapping=aes(x=time,y=P),color="blue") +

geom\_hline(yintercept=0,color="darkgrey") +

geom\_vline(xintercept=0,color="darkgrey") +

labs(x = "Time", y = "P")

Change the values of P(0), r and K. What do you observe this time?

**Model calibration**

We have implemented a population model following a logistic growth. This is all good and interesting, and it can be used to examine some theoretical behaviour (such as chaotic behaviours), but we may want to apply it to some real data, and estimate the values of parameters r and K for a population, as well as the initial conditions P(t=0). This is called model calibration. Here we will see a simple way to do it, for some data I have generated with a model (instead of real data, so that I know the real values of r and K!).

Here is the code I used (without the parameter values 😉). Can you see how it differs from the code you wrote for the discrete model, and why I made this modification?

tmax <- 300

x <- numeric(tmax+1)

for(i in 2:(tmax+1)){

x[i] <- r\*x[i-1]\*(1-x[i-1]/K)+x[i-1]+rnorm(1,0,x[i-1]/100)

}

Get the data

Read the file “pop\_LG\_simul\_noise\_small.csv” and store the values in a data frame. Plot and look at the values.

As we discussed in the introduction to modelling lecture, there are two ways to calibrate a model: directly from data, or by retro-fitting or inverse modelling, i.e. by adjusting the parameter values so that the output of the model fits the data. Ideally, we want to estimate most parameter values from real data, to avoid overfitting.

Initial conditions P(t=0)

How can you get this information from the data? Implement it in R.

Parameter K

Can you think of a way to estimate K directly from the data? Think of what K means in your model (look at the plots you have generated above). It is possible to estimate K in one line of code. Implement it in R.

Parameter r

Here we will use inverse modelling. The idea is the following: we will run multiple simulations, generating data for different values of r. We will then compare the model outputs to the data from the csv file. The r value generating the outputs closest to the data should be close to the real one.

First, we need to decide if we want to use all the data, or only part of it. What do you think? What part of the curve has r an effect on? Create a variable tmax representing the cutoff (the number of data points you will use).

We will then execute the following code. I commented it, but take the time to understand what each line of code is doing. Ask if you don’t get it.

out.df.list <- list() ##create a list in which we will store the model outputs for the different values of r

i <- 1 ##this is used to keep track of indices for out.df.list

for (r in seq(0.01,1,0.01)){ ##we vary r from 0.01 to 1, by steps of 0.01. Check ?seq for further details

parameters <- c(r=r, K=K) ##these are our parameters, as before

state <- c(P=pop$P[1]) ##this is our initial population value, as before

times <- seq(0,tmax,by=1) ##these are our time steps. This time, we want one value every time step, to match how the data was generated

out <- ode(y=state, times = times, func = LG, parms = parameters) ##we apply the ode() function, as before

out.df.list[[i]] <- data.frame(out) ##we store the data frame in the list out.df.list

i <- i+1 ##we need to increment i for the previous line of code

}

out.df.list is a list containing the data frames with the population values for the different values of r. We now need a measure of fit. Here we will simply take the sum of the absolute values of the differences between the simulated and the “real” data points. The “best” r value is therefore the one minimising this difference. You can compute it with the following code:

fit <- numeric(length(out.df.list))

for(i in 1:length(out.df.list)){

fit[i] <- sum(abs(out.df.list[[i]]$P-pop$P[1:(tmax+1)]))

}

ind.est <- which.min(fit) ##this is the index of fit which corresponds to the minimum value in fit

In the original mode, I used r <- 0.2. We can find the corresponding model outputs with the following line of code:

ind.real <- which(seq(0.01,1,0.01)==0.2)

Are ind.est and ind.real the same, or do they differ? By how much? What does it mean? To help you interpret the difference, we can plot both the values generated by the estimated and the real r value with the following code:

colors <- c("Estimated"="red","Real"="cyan4")

ggplot()+

geom\_line(mapping=aes(x=0:tmax,y=pop$P[1:(tmax+1)])) +

geom\_line(mapping=aes(x=0:tmax,y=out.df.list[[ind.est]]$P,color="Estimated")) +

geom\_line(mapping=aes(x=0:tmax,y=out.df.list[[ind.real]]$P,color="Real")) +

geom\_hline(yintercept=0,color="darkgrey") +

geom\_vline(xintercept=0,color="darkgrey") +

xlim(0,tmax\*1.1) +

ylim(0,K+100) +

labs(x = "Time", y = "P",color = "Legend") +

scale\_color\_manual(values = colors)

Change the value of tmax. Does it change anything?

Now, redo all of the above with the data from the csv file named “pop\_LG\_simul\_noise\_big.csv”, for which I used a noise 10 times as high as before. Does it change your results? How and why?

**That was too easy and you have some extra time?**

***Task 1***

Can you rewrite some of your code using apply / lapply / sapply instead of for loops?

***Task 2***

Try to produce a bifurcation plot from a discrete model output:

* Define the initial conditions
* Execute the model for r of increasing values, for 500 time-steps
* For each simulation, store the last 50 population values
* Plot the population values (y axis) against the corresponding r values, all in a same graph