EEB 201:
Introduction to Dynamic Models in R

Jamie Lloyd-Smith

September 15, 2015

1 Introduction

Earlier you got an introduction to the R language and computational environment. Now we will use R to get some experience with programming and analyzing mathematical models in population dynamics. All of the same work could be completed in Python or many other procedural programming languages – the basic ideas would be the same, but details of implementation would be a bit different. Goals for these sessions are to:

- Get experience writing scripts and functions in R, so you can make it do what you want!
- See how computer simulations can help build intuition about mathematical models (and later in the year, how they can complement insights from mathematical analysis of models).
- Go through a few worked examples and exercises, so you have templates you can build on for further work in courses and research.

Our aim is to give you a taste of the power and flexibility of scientific computing, so that you are motivated to learn more and to add this powerful tool to your repertoire in graduate school and beyond. This bootcamp is simply a teaser, but (a) more quantitative biology (theory, modeling, and programming) will come up in your core courses in EEB, (b) there are further courses available that focus on programming and modeling in EEB, and (c) there's lots of great information available on the web and in books. One excellent online reference which provides more details on programming dynamic models in R is `www.cam.cornell.edu/~dmb/DynamicModelsLabsInR.pdf` (and note that this is an accompaniment to an excellent book on dynamic modeling in biology).

2 Some modeling terminology

Before we start, here are some common terms *State variables* are quantities describing the current state of the system. A common state variable in models of population dynamics is the size of the population at time t, often denoted $N(t)$. State variables typically change through time in the model – indeed this is often the point of the model. You will need to make some decision about the starting values for the state variables, which are referred to as the *initial conditions*.

Parameters are quantities that describe the processes affecting the system. Often these will correspond to the rate or probability that a given process occurs, e.g. the birth rate or death rate. For simple modeling exercises, the parameter values are typically assumed to be fixed (but there is no reason why they can't change, due to changing environmental conditions for instance).

3 Coding a discrete-time model in R

First we will consider discrete-time models – that is, models where time advances in 'chunks' called timesteps. These models are often appropriate for systems with a strong biological rhythm, such as populations with a single annual birth pulse. One advantage of discrete-time models is that they are extremely easy to program. Consider the geometric growth model:

$$N(t+1) = RN(t) \tag{1}$$

This simple equation gives the update rule that is needed to step from one timestep to the next (this is sometimes called a *recursion relation*). To simulate the model, we just need to step through the desired number of timesteps, typically using a for loop, and use this rule to update the value of the state variable. Easy.

3.1 General layout of modeling scripts

Simple scripts used to simulate dynamic models often have four main sections (credit to Steve Ellner and John Guckenheimer for laying this out so simply):

1. Setup statements, if needed (e.g. loading libraries needed for the program).
2. Input data, set parameter values, and/or set initial conditions.
3. The actual calculations.
4. Display the results by plotting, saving, etc.

As with much in life, there's often more time spent in pre- and post-processing than in doing the calculations themselves. Using this layout isn't mandatory, but it is often a convenient way to structure your thoughts in writing these kinds of scripts.

3.2 Writing your first model: Geometric growth

Open a new file to write a script that runs the geometric growth model. I called mine `geometric_growth.R`. Enter the code shown below and save the file. Recall that the `#` symbol marks a comment, so all the text to the right of a `#` is invisible to R when it runs the script. I have added a lot of commenting to this script to relate it directly to the framework shown in the last section. You can add your own comments to the level that is helpful to you. As a general rule, don't skimp on comments when writing code! You'll thank yourself when six months goes by before you look at a program again.

```
# geometricGrowthScript.R
# a script to simulate and plot the discrete logistic model

# Setup
#   none needed, since program is so simple

# Set parameter values and initial conditions, and initialize variables for output
N0 <- 25
RR <- 1.05
ttMax <- 100   # total number of timesteps to simulate

NN <- matrix(NA, nrow=1, ncol=ttMax+1)  # initialize variable to a vector of NA values
NN[1] <- N0   # set first value to initial condition

# Loop over ttMax timesteps, using the model equation to update NN.
for (tt in 1:ttMax) {
    NN[tt+1] <- RR*NN[tt]
}

# Plot the results
plot(1:ttMax+1,NN, xlab="time", ylab="N", type="b", col='blue')
```

Note that we have used the variable `ttMax` to represent the total number of timesteps. This makes life easier if we want to run some short simulations and some longer ones, since you only need to change the parameter value once. There is a little trick needed if you want to store the value of the population size at each timestep, though. If you simulate 100 steps forward, you will need to store 101 values: the initial population size, plus the population size after each step forward. This is why we create a vector with `ttMax+1` elements to store all the values of `NN` for the simulation.

3.2.1 *Mini-exercise

Play with this script. Can you change the parameters to make the population decline? Does the qualitative behavior of the model depend on the initial population size? Can you find any other weird behavior? Commit a figure file showing your favorite output from your model.

3.2.2 *Exercise

Convert this script into a function so you can plot geometric growth curves simply by entering a statement like this at the command line:

```
geometricGrowthFun(N0=10, RR=0.95)
```

Unlike scripts, you can keep numerous functions in the same file (I used `bootcamp_functions.R`) and then when you `source` it you can load a whole set of functions into the workspace. Note that every time you add a new function you can just `source` this file again.

3.3 Logistic growth in discrete time

Make a new script called `discrete_logistic.R` that simulates the discrete logistic growth model:

$$N(t+1) = N(t)\left(1 + r_d\left(1 - \frac{N(t)}{K}\right)\right) \tag{2}$$

Do this by altering the script for the geometric model and saving it in a new file. Hint: Think about what each line in your geometric growth script does. Where is the calculation of population size happening? This is all you need to change.

3.3.1 *Mini-exercise

Now convert this script into a function, so you can run and plot the model from the command line by entering:

```
discreteLogisticFun(N0=10, rr=0.5, KK=100)
```

Explore the behavior of this model for different values of r_d. Notice anything strange? This model, which is also known as the logistic map, has a famous history in ecology and in the science of complex systems. We will learn more about it in EEB 200B.

3.3.2 *Mini-exercise (a bit harder)

Use your function to do a systematic exploration of the dynamics of the discrete logistic model. Keep $N(0) = 10$ and $K = 100$, and examine the behaviour of the model for r_d equalling -0.3, 0.3, 1.3, 1.9, 2.2 and 2.7. Use a for loop to conduct this sensitivity analysis. Plot all these examples in an array of subplots, by adding the line:

```
par(mfrow = c(2,3))
```

before the for loop. This command changes the graphing parameters for the current figure window. It creates an array of subplots (2 rows, 3 columns) which R will fill in one at a time as it encounters plotting commands. To reset the graphing parameters, either close that figure window or enter the command `par(mfrow` c(1,1))=, which will go back to a single plot per window. Hint: Create a vector that holds the desired values of r_d, and use a for loop to run the model (using your function from the last mini-exercise) for each value of r_d. Suppose you call your vector `rdVec`, then you will want to run your loop over the range `ii in 1:length(rdVec)`. Then you can get each value of r_d by using `ii` as an index to the vector.

3.3.3 Advanced exercise (optional)

Write a script to make a bifurcation plot for this model, showing how its long-term dynamics change as the parameter r_d is varied. You will need to collect a set of values reflecting the long-term dynamics of N for each value of r_d, where r_d falls between -1 and 3. Plot these N-values as points on the y-axis, versus the corresponding value of r_d on the x-axis. Hint: you may need to look up the R command `matplot`. If you get done with this and still have time, do it again using the `apply` function.

4 Writing a differential equation model in R

To work with differential equations in R, we need to import a special package of code that contains relevant functions. In the Windows R-GUI, this can be done using the menu system Packages/Install Package(s). Follow the prompts and download the deSolve package. If this is the first time you're doing this, you'll need to select a CRAN mirror which is the server you're taking the file from. Choose one of the ones in California – 'USA (CA 2)' is actually in the statistics department at UCLA.

Alternatively you can do it from the command prompt:

```
install.packages("deSolve")
```

Installing the package just downloads the files onto your computer. To make the commands in that package accessible during a given R session, you need to run another command. Type this at the command prompt:

```
library(deSolve)
```

Now we've got a whole library of functions for working with ordinary differential equations (or ODEs). (Note that an ordinary differential equation is a differential equation where you're considering changes with respect to only one independent variable. For dynamical models this is typically time, i.e. we're working with equations that look like $\frac{dN}{dt} = rN$. Partial differential equations or PDEs consider changes with respect to several independent variables, e.g. to time and space. We won't get into those for now.)

4.1 A first example of an ODE model: exponential growth

We're going to use a function called `lsoda` which is a versatile differential equation solver from the deSolve package. This is a great tool but requires that you interact with it in a very specific way. We'll look at an example of this format for the simplest ODE model for population dynamics, the exponential growth function:

$$\frac{dN}{dt} = rN \tag{3}$$

The first thing we need to do is make an R function that contains the equation for the model (or this can be extended to models with several equations, as we'll see). This function has a specific format:

```
expGrowthODE  <- function(tt, yy, pars) {
    derivs <- pars['rr'] * yy
    return(list(derivs))
}
```

These functions must always have the same input arguments, in the specific order shown. Here `tt` is the time, `yy` is the state variable, and `pars` is a vector of parameters used in the model. Note that, as always, the actual names of these variables don't matter, as long as the names are used consistently and contain the right type of information.

We also have to use a specific syntax to simulate the model using `lsoda`. The generic syntax is:

```
output <- lsoda(init, tseq, ODEfunction, pars)
```

where

- `init` is the initial value of the state variable
- `tseq` is a vector of the time points where the model will be evaluated
- `ODEfunction` is a place-holder for the name of the function holding the model equations (e.g. expGrowthODE, above)
- `pars` is a vector containing any parameters used in the model

Let's load some appropriate values into these variables:

```
init <- 1
tseq <- seq(0, 20, by=0.01)
pars <- c(rr = 0.1)
```

Note that we're using the convention of assigning names to the elements inside the vector, so the value of `rr` can be accessed by entering `pars['rr']`. We could also define the value of `pars` without specifying these names:

```
pars <- 0.1
```

or

```
pars <- c(0.1)
```

which will generalize better to situations where we have multiple parameters, but using the names helps to avoid confusion and means that we don't have to remember what order the elements are in. Note that if we use numeric indexing for the `pars['rr']` vector, we'll need to change the way we coded the `expGrowthODE` function so it's not looking for `pars['rr']`.

Now we're ready to run the model. The command to do this is:

```
expGrowthOutput <- lsoda( init, tseq, expGrowthODE, pars)
```

Look at the top few lines of the output by typing `head(expGrowthOutput)` at the prompt. It should look like:

```
> head(expGrowthOutput)
     time        1
[1,] 0.00 1.000000
[2,] 0.01 1.001001
[3,] 0.02 1.002002
[4,] 0.03 1.003005
[5,] 0.04 1.004008
[6,] 0.05 1.005013
```

The first column shows the row number for each row. The middle column displayed (which is actually the first column of the variable's contents) shows the values of the 'time' variable. The right-most column shows the value of the state variable at each time.

To plot the output of the model we just write:

```
plot(expGrowthOutput[,1], expGrowthOutput[,2], col='blue', type='l')
```

4.1.1 *Mini-exercise

Make a script that brings together the commands necessary to make and run this model. You can organize it using the four parts outlined earlier in Section 3.1, and remember to include the `library(deSolve)` command in the setup section so the necessary functions are always available. Play with the script to see what different behaviors you can get from the model for different parameters and initial conditions. Does it behave the way we expect from our memories of population ecology? Does it exhibit the same range of qualitative behaviors as the geometric growth model?

4.2 Logistic growth model

Now we're ready to tackle the logistic growth model in continuous time:

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right) \tag{4}$$

4.2.1 *Mini-exercise

Make a new script that runs the logistic growth model. Try to do this on your own by building on your exponential growth script. (One version is at the end of the document if you get stuck.) Change the parameters to get a feel for the model. Can you get all the behaviors you expect? Can you choose parameters to make the model give dynamics like the exponential growth model?

4.2.2 Advanced exercise

How will the dynamics of the logistic model be impacted by harvesting (e.g. by fishing or hunting of the population)? A simple model that incorporated a constant per capita risk of being harvested is:

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right) - hN \tag{5}$$

If you remember how, analyze this model to find its equilibria and their local stability conditions (i.e. using calculus). Check your findings with computer simulation. If you don't remember how, just explore the model equilibria and stability by simulation.

5 Sensitivity analysis and batch runs

5.1 Sensitivity to initial conditions

Local stability analysis describes how the system will behave in the neighborhood of its equilibria (or other attractors). We will learn mathematical approaches to stability analysis later in the year, but for now we can investigate the stability properties of our model by simulating to see how the system dynamics unfold from particular starting positions, depending on the parameter values. First we will just plot the time series for a range of different initial conditions, using a loop. (Note: to make multiple plots on one set of axes, we need to use the `lines()` function. This works well, but `lines()` only works if the plot has already been initialized using `plot()`. So we add a statement that sets up an empty set of axes before we start the loop.)

```
ICVec <- seq(0, 200, by=10)

plot(x=NULL, y=NULL, xlim=c(1,max(tseq)), ylim=c(0,2*KK), xlab='time',
    ylab='N', main='Logistic growth')

for (ii in 1:length(ICVec)) {
    init <- ICVec[ii]
    logisticOutput <- lsoda(init, tseq, logisticGrowthODE, pars)
    lines(logisticOutput[,1], logisticOutput[,2], col='red', type='l')
}
```

Make these plots for a few (well-chosen) values of r to investigate the stability of the different equilibria of this model.

Often it is preferable to extract some output measure from the model results, and to plot that measure as a function of another quantity that you are changing. For example, maybe we want to know how big a population we can get in 5 years, given different starting population sizes. First we can find what index corresponds to year 5 by applying the `which` command to the first column of the model output:

```
index <- which(logisticOutput[,1]==5)
```

Then we initialize a vector to hold the results from each model run. There will be one run per initial condition, hence:

```
nnVec <- rep(NA, 1, length(ICVec))
```

Now we use a loop to run the model for each initial condition, and collect the output of interest in our vector:

```
for (ii in 1:length(ICVec)) {
    init <- ICVec[ii]
    logisticOutput <- lsoda(init, tseq, logisticGrowthODE, pars)
    nnVec[ii] <- logistic.output[index,2]
}
```

Open a new plotting window (depending on your R environment, you may need to use a special command like `x11()` or `windows()` to open a new window), and plot the results:

```
x11()   # if needed
plot(ICVec, nnVec, xlab='Initial population size',
ylab='Population at time=5', type='b', col='red')
```

This process of writing a script to run a model many times and collect the output is called a *batch run*. You can repeat this procedure for other values of r, and plot the results on the same plot using the `lines` command.

5.2 Exercise: Sensitivity to parameter values

Very often we are interested in how different parameter values influence model results. Alter the code we've just been using to plot the population size at time = 5 for a range of values of r. (Hint: only two lines need to change.)

6 Writing a two-variable differential equation model

Just so you have a template, here's a script that runs a two-dimensional ODE model (the Lotka-Volterra predator-prey model) and plots the outcome. Type it in, and run it for different parameter values to see what dynamics you can get.

```
# source("LV_script.R")
# A script to run a simple 2-dimensional ODE system

# equations are
# dN/dt = r N - c N P
# dP/dt = e c N P - d P

library(deSolve)

# pred-prey cycles
pars <- c(rr = 2, cc = 0.1, ee = 0.1, dd = 0.2); init <- c(200, 20)

tseq <- seq(0, 50, by=0.02)

# define the model equations
# note that yy is now a vector with two elements for the two state variables
lvPredpreyODE  <- function(tt, yy, pars) {
    derivs <-rep(NA, 2)

    derivs[1] <- pars['rr'] * yy[1] - pars['cc'] * yy[1] * yy[2]
    derivs[2] <- pars['ee'] * pars['cc'] * yy[1] * yy[2] - pars['dd'] * yy[2]
    return(list(c(derivs)))
}

lvPredpreyOutput <- lsoda( init, tseq, lvPredpreyODE, pars)

# make array of two plots
par(mfrow=c(1,2))

# plot output as time series
plot(lvPredpreyOutput[,1], lvPredpreyOutput[,2],
    col="blue", type="l",
    xlab="time", ylab="# of individuals",
    ylim = c(0,1.2*max(lvPredpreyOutput[,2])))

# add a line to the plot with the predators
points(lvPredpreyOutput[,1], lvPredpreyOutput[,3], col="red", type="l")

# add a simple legend
legText = c("Prey", "Predator")
legend("topright", legText, lty=1, col = c('blue','red'))
```

```
# plot output as a phase plot, in second subplot
plot(lvPredpreyOutput[,2], lvPredpreyOutput[,3],
    xlab="Prey", ylab="Predators", type="l", col='black')
```