Ecological Dynamics
Lab 1: Demographic models

Background

- Mathematical modeling is becoming increasingly important in biological disciplines.
- `R` is fast becoming the *de facto* language of mathematical modeling because it is flexible, free and relatively easy to understand.
- **The instructions assume that you have read chapter 2 of the course manual**.

Objectives

- Learn to use the `deSolve` and `manipulate` packages to develop and analyze continuous-time demographic models.
- Learn how to create `functions` and `for` loops to describe iterative processes such as discrete demographic models.
- Learn how to plot model output.

Instructions

- Launch RStudio, open a new script file and save it as `lab1-yourLastName.R`.
- Complete the activities below by adding the appropriate commands to your `R` script file.
- Embed your answers as comments in the `R` script file.
- At the end of your session, save your file onto a flash drive (files saved on lab computers are wiped when you log out).

A gentle introduction to `R`

Although `R` comes with its own **I**ntegrated **D**evelopment **E**nvironment or IDE, we will be using a new and better open-source IDE called RStudio. The job of the IDE is to make your life easier. You will be spending a lot of time writing code and analyzing results, so you need to make sure that you have the best environment available. IDEs provide a single application to program, access help functions, download packages, and run your code. Once you have launched RStudio, you will be greeted with a 4-panel window:

The different tabs in the top left panel show the content of the open **script** files. The different tabs in the top right panel show the **command history** (i.e., the commands that were run in the past) and the **workspace** (i.e., the variables and functions that were created during the `R` session). The bottom left panel shows the **console** where the output of the commands appears. The different tabs in the bottom right panel show plotted figures, a list of installed packages, **help** on specific functions, and the list of files in the current working directory. Keyboard shortcuts can be used to switch between the different panels (e.g., for Macs `control-1`: editor, `control-2`: console, `control-3`: help).

In RStudio, it is advisable to place commands directly in a new `R` script instead of typing them one at a time in the console. This is because commands placed in a script can be sent to the console

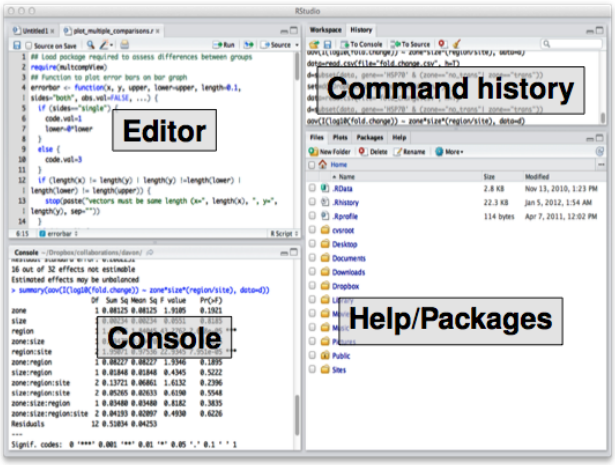


Figure 1: RStudio on a Mac

for execution via RStudio's graphical user interface (i.e., "Run" icon in the top right corner of the Editor panel) or keyboard shortcuts. Individual lines of code or commands can be sent to the console for execution by using the keyboard shortcut `command-return` on a Mac or `control-enter` on Windows and Linux. Alternatively, highlighting a few lines of code and using the same keyboard shortcut will send the whole block to the console for execution. Finally, to execute the entire script, click on the "Source" icon in the top right corner of the Editor window or use the keyboard shortcut `command-shift-return` on a Mac.

RStudio has a number of other extremely powerful features, most of which are easily discoverable. Here, I will focus on *the* killer feature: code **completion** via the keyboard shortcut `tab` in both the console and editor panels. After typing the first few leading letters of a function or variable name, hit the `tab` key to get a list of candidate functions and variables that start with those letters:

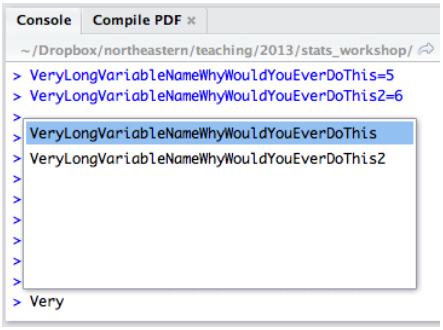


Figure 2: `tab` completion of variable name

Using the up and down keys on the keyboard will scroll through the list of candidate functions/variables and pressing `return` on a Mac or `enter` on Windows/Linux will select a command. For functions, tab completion shows the package that the function belongs to in curly braces and a small snippet of its documentation:

In this example, the `lm` function has quite a few arguments. To get additional details about the function, hit the `F1` key to bring-up the full documentation for the function in the help tab located in the bottom right panel of RStudio. Crucially, `tab` completion also works for function arguments.

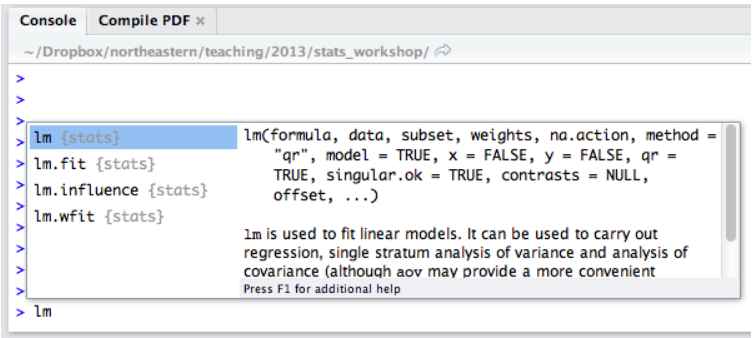


Figure 3: `tab` completion of function name

Once the list of arguments appears, use the up and down keys to scroll through the list of arguments and then press `return` to have the selection appear in the console:

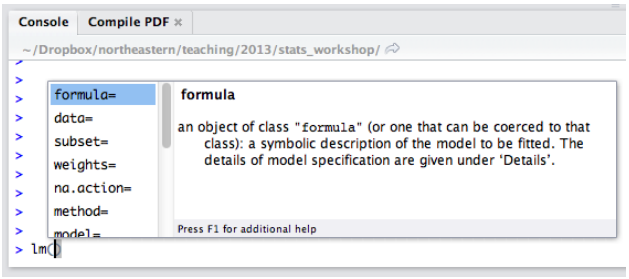


Figure 4: `tab` completion of function arguments

To get further help on any function or command, type `help.search("commandName")` or `?commandName` at the console to open the local `R` documentation within RStudio. To search online forums for help, type `RSiteSearch("commandName")` at the console. Doing so will open a web browser window. Note that the `F1` shortcut will only work on a Mac if the "Use all F1, F2, etc. keys as standard functions" option is selected in the **Keyboard** section of the **System Preferences**.

Variables and simple objects

In `R`, **variables** are used to store **objects** such as numerical values or text. Each object has a specific type called a `class`. The `class` defines the kinds of operations that can be conducted using the object. For instance, objects of the class `numeric` can be used to perform simple algebraic calculations by using mathematical **operators** such as `+,/,-`. The special assignment operator `<-` is used to store these algebraic calculations into variables. Variable names can contain a combination of letters, numbers and certain special characters (e.g., '.' and '_'). However, variable names cannot begin with special characters or numbers. `R` is case-sensitive so `var1` and `Var1` represent two distinct variables. Variable names should be short but informative. The following example can be typed directly in the console or typed in the script and executed:

```
# This comment documents the code and is ignored by R
my.sum <- 34 + 9
```

The above code block computes the sum of 34 and 9 and stores the result in variable `my.sum`. To see the value stored in variable `my.sum`, simply type `my.sum` at the console:

```
# Print the contents of the variable
my.sum
```

```
## [1] 43
```

Since `my.sum` is a variable of class `numeric`, you can perform algebraic calculations directly with it. For instance, to compute its square root or log base 10:

```
# Compute the square root and log10
sqrt(my.sum)
```

```
## [1] 6.557439
```

```
log10(my.sum)
```

```
## [1] 1.633468
```

Note that these calculations were not stored in `my.sum`, so its value has not changed:

```
# Print the contents of the variable
my.sum
```

```
## [1] 43
```

Manipulating compound objects

Variables can also be used to store **compound objects** containing multiple values (i.e., vectors, lists, matrices, etc...). The simplest compound object is called a **vector** and can be used to store a list of values in a 1-Dimensional structure. To create a vector by hand, one can use the `c` function to combine values into a vector:

```
# Create variable my.vec containing 5 numerical values
my.vec <- c(1, 3, 5, 6, 0)
```

Once a vector has been used, you can determine its length (via function `length`) or access specific parts of it because each value has a corresponding 1-based **index** or location. For instance, to retrieve the value stored in index 1 of `my.vec`:

```
# Get length of vector
length(my.vec)
```

```
## [1] 5
```

```
# Get value in location 1
my.vec[1]
```

```
## [1] 1
```

You can also retrieve multiple values over a non-contiguous range of indices:

```
# Get values at locations 2, 3, and 5
my.vec[c(2, 3, 5)]
```

```
## [1] 3 5 0
```

One can also create 2-Dimensional complex objects called **matrices** to store multiple vectors into a single variable. For instance, one can combine two vectors by column binding (`cbind`) or row-binding them (`rbind`) together. In this case, both vectors must have the same number of elements or length. If we use `rbind`, the matrix will consist of 2 rows and n columns, where n is the length of the vectors. Conversely, if we use `cbind`, the matrix will consist of 2 columns and n rows. We can

verify this by issuing the `dim(mat)` function, which determines the number of rows and columns in variable `mat`:

```
# Create new vectors
my.vec1 <- c(1, 3, 5, 4)
my.vec2 <- c(4, 8, 15, 2)
# Create matrix via cbind:
(my.mat1 <- cbind(my.vec1, my.vec2))
```

```
##      my.vec1 my.vec2
## [1,]       1       4
## [2,]       3       8
## [3,]       5      15
## [4,]       4       2
```

```
# my.mat1 is a 4 x 2 matrix
dim(my.mat1)
```

```
## [1] 4 2
```

```
# Create matrix via rbind:
(my.mat2 <- rbind(my.vec1, my.vec2))
```

```
##         [,1] [,2] [,3] [,4]
## my.vec1    1    3    5    4
## my.vec2    4    8   15    2
```

```
# my.mat2 is a 2 x 4 matrix
dim(my.mat2)
```

```
## [1] 2 4
```

Naturally, there is a 2-Dimensional indexing system for retrieving values from specific rows and columns of a matrix. In this case, `mat[row, col]` will retrieve the values of matrix `mat` stored in row `row` and column `col`:

```
# Get values in rows 1-3 and column 1
my.mat1[1:3, 1]
```

```
## [1] 1 3 5
```

```
# Get values in row 2 and columns 2-3
my.mat2[2, 2:3]
```

```
## [1]  8 15
```

Creating functions

Functions in `R` are just like functions in calculus (e.g., $y = f(x) = x^2$): they take in a parameter, perform a calculation and return the result. However, the functions in `R` can do much more than mere calculations; they can be used to perform entire analyses. The basic syntax for a function is simple:

```
f <- function(x) {
    return(x^2)
}
```

Here, we have created a function called `f` that takes in a parameter `x` and returns its square. Once the function is known to `R`, we can use it compute the square of single values or vectors:

```
# Compute the square of 6
f(6)
```

```
## [1] 36
```

```
# Compute the square of 1-5
f(1:5)
```

```
## [1]  1  4  9 16 25
```

We can create a more complex function `g` that expects two parameters `x, y` and returns the difference of their squares. Here we will also give the parameters default values of `x=0` and `y=2`. This means that if we call the function `g` without specifying the parameters, then it will assume that `x=1` and `y=2`:

```
g <- function(x = 1, y = 2) {
    return(x^2 - y^2)
}
# Compute g using default parameters
g()
```

```
## [1] -3
```

```
# Compute g for x=5, y=9 and store result in new variable my.calc
(my.calc <- g(5, 9))
```

```
## [1] -56
```

If you specify the parameter values without their names, then `R` will assume that you specified them in the right order (e.g., `x` first then `y` here). However, you can also specify the parameters out of order if you use their names explicitly:

```
# Compute g for x=5, y=9
g(y = 9, x = 5)
```

```
## [1] -56
```

```
# Compute g for x=9, y=5
g(9, 5)
```

```
## [1] 56
```

One limitation of functions in `R` (and most programming languages) is that they can only return a single variable. However, if you need your function to return multiple values, you can simply store the results in a `compound object` such as a vector or a matrix and return it. For example, we can create a simple function `h` that returns all the values between (but excluding) `x` and `y` in the form of a vector:

```
# Return values between x+1 and y-1 as a vector
h <- function(x, y) {
    result <- (x + 1):(y - 1)
    return(result)
}
# Get vector of values from 2 to 8
(my.seq <- h(1, 9))
```

```
## [1] 2 3 4 5 6 7 8
```

Loops

Loops are used to repeat pieces of code multiple times. They are particularly useful for programming iterative processes such as discrete demographic models where the abundance at time t depends on the abundance at time $t-1$. The syntax of the `for` loop in `R` is very basic and resembles that of a function:

```
for (i in 1:5) {
    # Useful code to repeat
}
```

Here, `i` is a parameter that specifies the number of times the code within the `for` loop should be repeated. The calculations from each iteration of `for` loops are often stored in `compound objects`. For example, let's create a `for` loop that will iteratively take the square of a value. To begin, we need to determine the number of iterations `n` the `for` loop will run and create an empty vector `result` of the same size to store the calculations. We can then set the initial value of `results` and execute the loop:

```
# Number of iterations
n <- 5
# Create vector of n zeros
result <- numeric(length = n)
# Initialize first position of results to 3
result[1] <- 3
for (i in 2:n) {
    # Current value is square of previous value
    result[i] <- result[i - 1]^2
}
# Print results
result
```

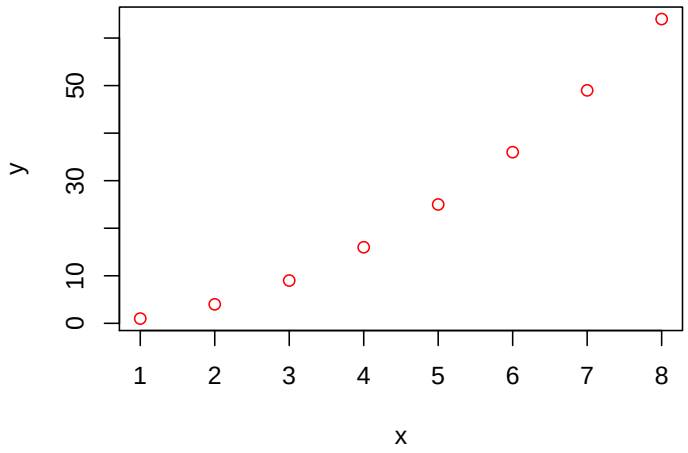
```
## [1]        3        9       81     6561 43046721
```

Note that we had to use the parameter `i` to identify the index of `results` in which to store the calculation from each iteration of the `for` loop. Can you guess why we started the `for` loop with `i=2`?

Plotting

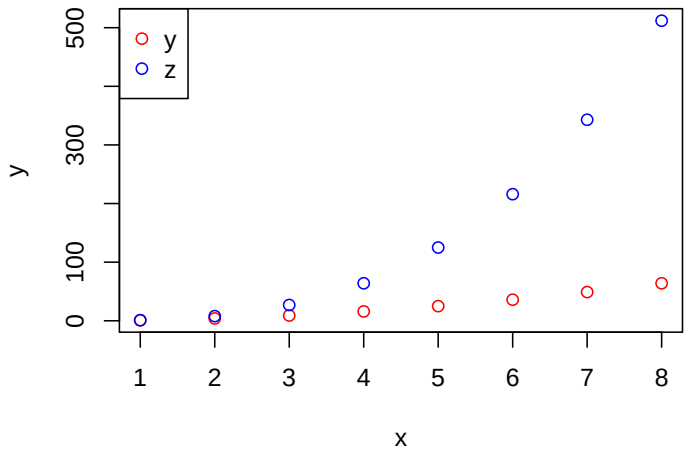
The plotting system is `R` is incredibly powerful but fairly complex. A full description of the plotting functionality is provided in chapter 2 of the course manual. Here, I will merely emphasize a few key concepts. The base graphic system in `R` embraces a layered approach for plotting. A figure is initiated by using the function `plot` and specifying a few key arguments (type `?plot` at the console of help and a full list of all the arguments). For example, to plot a variable y (y-axis) against a variable x (x-axis):

```
# Example of a dataset
x <- 1:8
y <- x^2
plot(x, y, ylab = "y", xlab = "x", type = "p", col = "red", pch = 1)
```



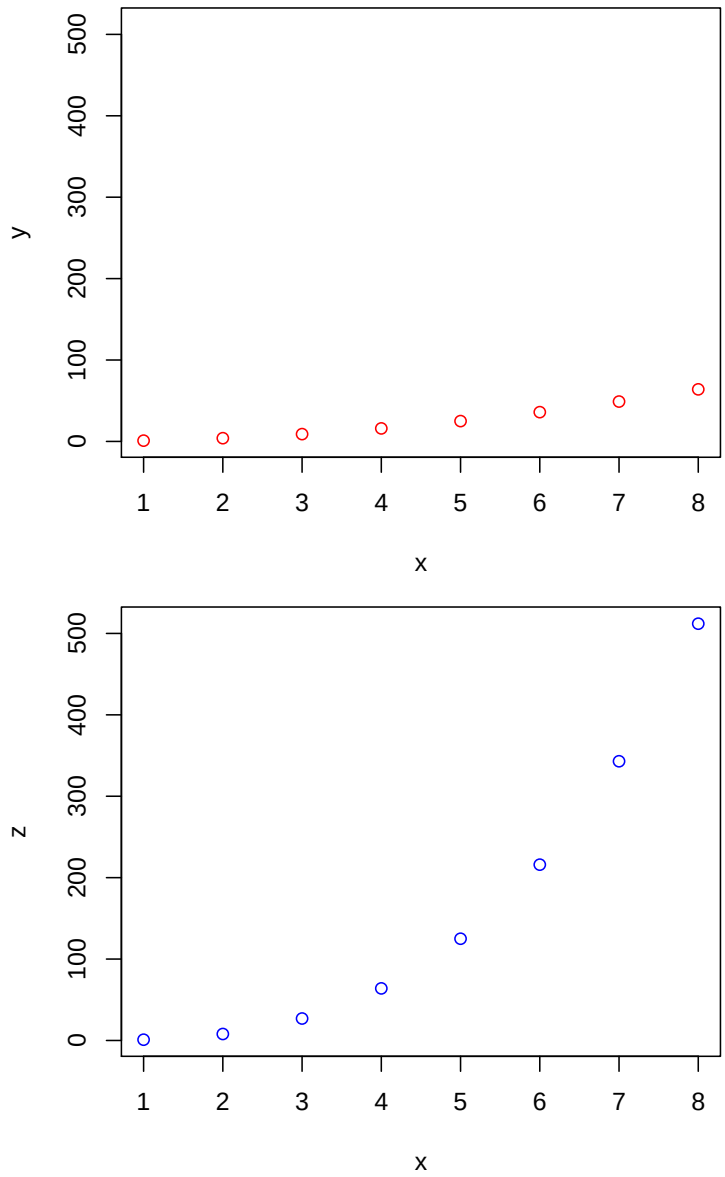
The `type="p"` argument means that we want to plot **points**. We could have selected `"l"` for **lines** or `"b"` for **both**. The `pch=1` argument means that we want to plot the data using open circles. To add data to the plot, you have to use functions `lines` to add lines and `points` to add points. For instance, to add `z` to the existing plot:

```
z <- x^3
plot(x, y, ylab = "y", xlab = "x", type = "p", col = "red", pch = 1, ylim = range(y,
    z))
points(x, z, col = "blue")
# Add a legend
legend(x = "topleft", legend = c("y", "z"), col = c("red", "blue"), pch = 1)
```



Finally, `R` can produce multi-panel figures using function `par(mfrow=c(nrows, ncols))`. For instance, to plot `y` and `z` in a 2-row x 1-column set of panels:

```
par(mfrow = c(2, 1), mar = c(4, 4, 1, 1))
plot(x, y, ylab = "y", xlab = "x", type = "p", col = "red", ylim = range(y,
    z), main = "")
plot(x, z, ylab = "z", xlab = "x", type = "p", col = "blue", ylim = range(y,
    z), main = "")
```



Each time the function `plot` is called, it will move to the next panel. Hence, to add multiple lines to a panel, you will have to use the `plot` function to initiate the panel and then add lines using the `lines` function. You will notice that the y-limits are identical for each panel. This is because argument `ylim` is based on the range (i.e., the min and max) found in both `y` and `z`.

Task 1: Numerically solving differential equations

To numerically solve differential equations, you must install an external package called `deSolve`:

```
# Only install the package once:
install.packages("deSolve")
```

Note that this should only be done once per computer. Once the package is installed you can simply load it by issuing the following command:

```
# Load the package every time you need it in your R session:
library(deSolve)
```

1. Once the package is installed and loaded, you are ready to solve any differential equation system. To do so, you will need to write a function that computes the change in the state variable over time as a function of the parameters of interest. For example, the function to compute the exponential growth would look like this:

   ```
   solve.expo <- function(t, y, parms) {
       dY <- parms$r * y
       return(list(dY))
   }
   ```

   Here, `t` is a vector containing the timesteps for which you would like to s̈olveÿour model, `Y` represents the state variable and `parms` is a `list` containing the parameters of your model. For the exponential growth model, there is only one parameter `r`. Once the function is defined, you can simulate the dynamics of the model by integrating the differential equation via the `ode` function from the `deSolve` package:

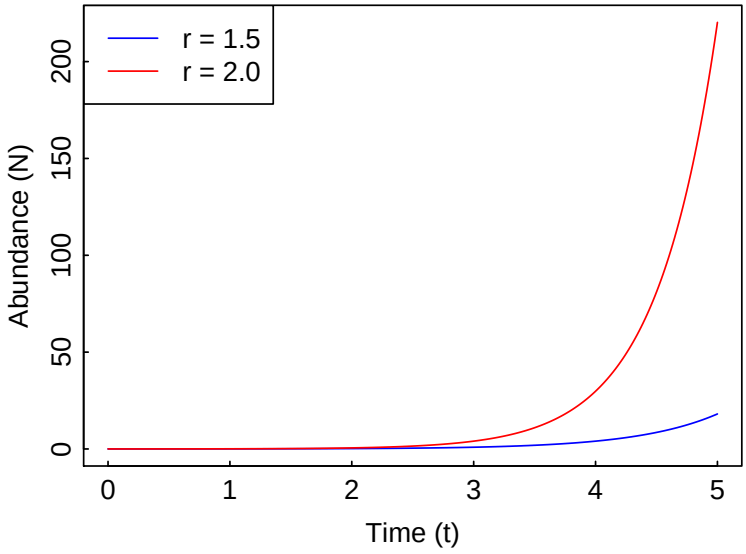
   ```
   # Initial abundance
   N0 <- 0.01
   # Total number of time steps
   total.time <- 3
   # Create list of parameters for model (in this case, only r=2)
   parms <- list(r = 2)
   # Here, function ode from package deSolve will numerically solve the
   # model via the Runge-Kutta 4-5 algorithm
   results <- ode(y = N0, times = seq(from = 0, to = total.time, len = 1000),
       func = solve.expo, parms, method = "ode45")
   ```

   The `times` vector specifies that the model should be simulated for 1000 time steps between 0 and `total.time`. The `results` variable will contain a matrix with the simulated dynamics; column 1 contains the timesteps and column 2 contains the corresponding abundances.

2. Simulate the model dynamics for 5 time steps with (i) `r` set to 1.5 and (ii) `r` set to 2. Then plot the abundances (y-axis) as a function of time (x-axis) using different colored lines for each value of `r`. Do not forget to label the axes and add a legend. Does the figure make sense based on your understanding of the exponential growth model?

   ```
   N0 <- 0.01
   total.time <- 5
   # Setup two values of r
   r <- c(1.5, 2)
   # Dynamics for r = 1.5
   parms <- list(r = r[1])
   results1 <- ode(y = N0, times = seq(from = 0, to = total.time, len = 1000),
       func = solve.expo, parms, method = "ode45")
   # Dynamics for r = 2
   parms <- list(r = r[2])
   results2 <- ode(y = N0, times = seq(from = 0, to = total.time, len = 1000),
       func = solve.expo, parms, method = "ode45")
   ```

```
plot(results1[, 1], results1[, 2], t = "l", xlab = "Time (t)", ylab = "Abundance (N)",
    col = "blue", ylim = range(results1[, 2], results2[, 2]))
lines(results2[, 1], results2[, 2], col = "red", ylim = range(results1[,
    2], results2[, 2]))
legend(x = "topleft", legend = c("r = 1.5", "r = 2.0"), col = c("blue",
    "red"), lty = 1)
```



Answer:

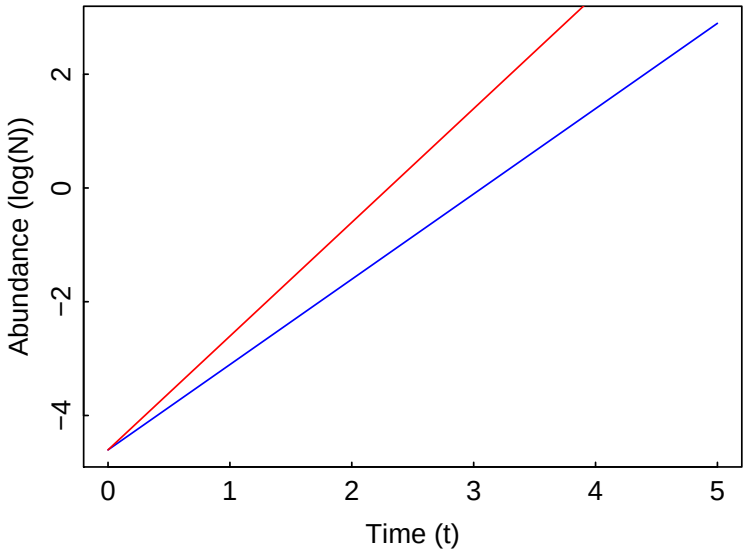
Yes, the figure makes sense: the line representing the higher intrinsic rate of growth lies above the line representing the lower intrinsic rate of growth.

3. To determine whether the simulations are correct, plot $\log(N)$ as a function of time t and see if it yields a line whose intercept is N_0 and whose slope is r. You can do this by fitting a simple regression model to the output of the model using the `lm` function. The coefficients from the regression should correspond to the model parameters used to simulate the dynamics (i.e., `r` and `N0`).

```
plot(results1[, 1], log(results1[, 2]), t = "l", xlab = "Time (t)", ylab = "Abundance (log(N))",
    col = "blue")
lines(results2[, 1], log(results2[, 2]), col = "red")
# N0 and r for r = 1.5
exp(coef(lm(log(results1[, 2]) ~ results1[, 1]))[1])

## (Intercept)
##        0.01

coef(lm(log(results1[, 2]) ~ results1[, 1]))[2]

## results1[, 1]
##           1.5

# N0 and r for r = 2.0
exp(coef(lm(log(results2[, 2]) ~ results2[, 1]))[1])

## (Intercept)
##        0.01
```

```
coef(lm(log(results2[, 2]) ~ results2[, 1]))[2]
```

```
## results2[, 1]
##             2
```



Answer:

Yes, the coefficients estimated from the simple regression correspond to the parameters used to simulate the dynamics.

4. We are now going to learn how to use the `manipulate` package to study the behavior of models. First load the package:

```
library(manipulate)
```

The package provides a really powerful function called `manipulate` that can be used to update your model by interactively changing the parameter values via a graphical user interface. The syntax is pretty basic:

```
manipulate ({
  # Code that needs to be manipulated
  parms <- list(r=rVal)
  times <- 0:tVal
  results <- ode (y = NVal, times=times, func=solve.expo, parms, method="ode45")
  plot(results[,1], results[, 2], xlab="Time (t)", ylab="Abundance (N)", type="l", col="blue")
},
# The graphical user interface elements used to change the parameter values
tVal=slider(min=20, max=500, initial=20, label="Total timesteps (t)"),
rVal=slider(min=0, max=4, initial=0.35, step=0.05, label="Growth rate (r)"),
NVal=slider(min=0, max=20, initial=1, label="Initial abundance (N0)"))
```

The code above will allow you to change the total number of time steps (`tVal`), the intrinsic rate of growth (`rVal`) and the initial abundance (`nVal`). Every time these parameters are changed, the code will simulate and plot the dynamics of the model in the background. This is a very rapid way of developing your intuition about the model's behavior.

5. Now it's your turn to grab the wheel. Use the code above to help you write a function to simulate the dynamics of the logistic model. Then, use the `manipulate` function to simulate and plot its behavior. How does changing the carrying capacity `K` affect the shape of the temporal dynamics? Can you explain this behavior?

```
solve.logi <- function(t, y, parms) {
    r <- parms$r
    K <- parms$K
    dY <- r * y * (1 - y/K)
    return(list(dY))
}
manipulate({
    parms <- list(r = rVal, K = KVal)
    times <- 0:tVal
    results <- ode(y = nVal, times = times, func = solve.logi, parms, method = "ode45")
    par(internal)
    plot(results[, 1], results[, 2], xlab = "Time", ylab = "Abundance",
        type = "l", col = "blue")
}, tVal = slider(min = 20, max = 500, initial = 20, label = "Total timesteps (t)"),
    rVal = slider(min = 0, max = 4, initial = 0.35, step = 0.05, label = "Growth rate (r)"),
    KVal = slider(min = 1, max = 50, initial = 2, step = 1, label = "Carrying capacity (K)"),
    nVal = slider(min = 0, max = 20, initial = 1, label = "Initial abundance (N0)"))
```

Answer:

Increasing the carrying capacity from 2 to 50 leads to a sharper rate of increase in population abundance at the onset of the simulations. This is because the larger the carrying capacity, the more the behavior of the model begins to resemble that of an exponential model. Indeed as $K \to \infty$, $\frac{N}{K} \to 0$ so $\frac{\mathrm{d}N}{\mathrm{d}t} = rN\left(1 - \frac{K}{N}\right) \to rN$

Task 2: Simulating the dynamics of discrete-time models

To numerically solve discrete-time models (difference equations), you need only use a `for` loop.

1. Using the help provided in the **Loops** section, write a `for` loop to simulate the dynamics of the geometric growth model for `R=1.3`, `N0=0.1` and 10 timesteps. Plot the results.

```
R <- 1.3
N0 <- 0.1
times <- 20
abund <- numeric(len = times)
abund[1] <- N0
for (i in 2:times) {
    abund[i] <- R * abund[i - 1]
}
plot(1:times, abund, t = "l", col = "blue", xlab = "Time (t)", ylab = "Abundance (N)")
points(1:times, abund, pch = 21, col = "blue", bg = "blue")
```

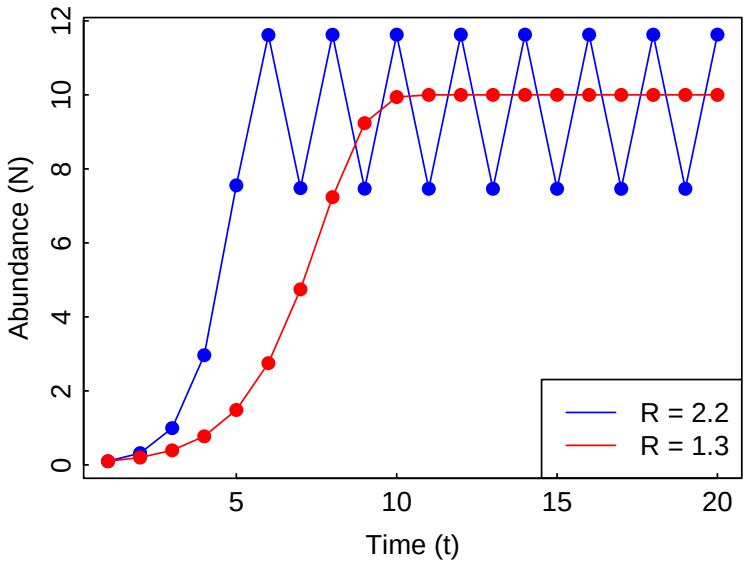
2. Now write a `for` loop to simulate the dynamics of the logistic map (discrete version of the logistic growth model) for `R=1`, `N0=0.1` and 20 time steps. Store the results for the next question.

```
R <- 1
N0 <- 0.1
K <- 10
times <- 20
```

```
abund1 <- numeric(len = times)
abund1[1] <- N0
for (i in 2:times) {
    abund1[i] <- (1 + R * (1 - abund1[i - 1]/K)) * abund1[i - 1]
}
```

3. Now rerun the model for `R=2.2`, plot the results from the previous question (in red) and the new simulation (in blue) on the same figure. Can you explain the difference in the dynamics?

```
R <- 2.2
N0 <- 0.1
K <- 10
times <- 20
abund2 <- numeric(len = times)
abund2[1] <- N0
for (i in 2:times) {
    abund2[i] <- (1 + R * (1 - abund2[i - 1]/K)) * abund2[i - 1]
}
plot(1:times, abund2, t = "l", col = "blue", xlab = "Time (t)", ylab = "Abundance (N)")
points(1:times, abund2, pch = 21, col = "blue", bg = "blue")
lines(1:times, abund1, col = "red")
points(1:times, abund1, pch = 21, col = "red", bg = "red")
legend(x = "bottomright", legend = c("R = 2.2", "R = 1.3"), col = c("blue",
    "red"), lty = 1)
```



Answer:

When `R=1`, the population attains and remains at its carrying capacity. However, when `R=2.2`, the population overshoots and undershoots its equilibrium because of its high growth rate. This leads to a limit cycle (here with one period).

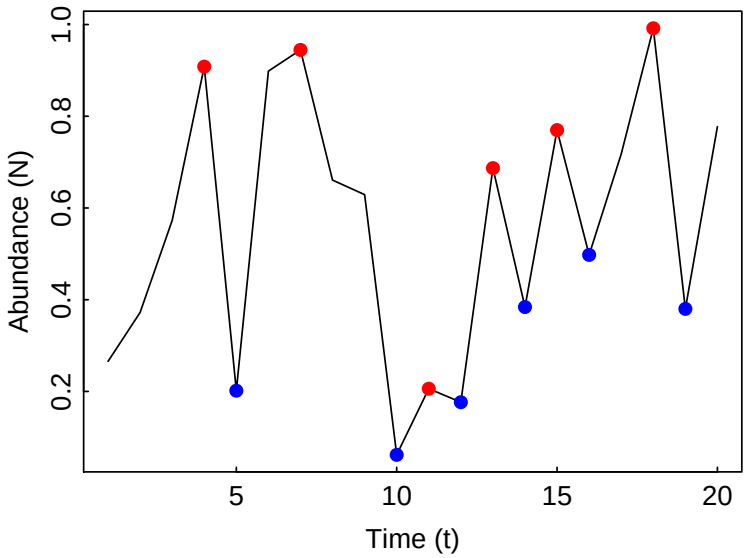
4. Create a function to simulate the logistic map. Use this function in conjunction with the `manipulate` function to visualize the dynamics of the model when you vary `R` from 1 to 4. Describe the behavior of the model as you increase `R` from 1 to 4.

```
simulate.discrete.logistic <- function(N0, t, K, R) {
    N <- numeric(len = t)
    N[1] <- N0
    for (i in 2:t) {
        N[i] <- (1 + R * (1 - N[i - 1]/K)) * N[i - 1]
    }
    return(data.frame(time = 1:t, N = N))
}
manipulate({
    results <- simulate.discrete.logistic(N0 = N0Val, t = tVal, R = rVal,
        KVal)
    plot(results$time, results$N, t = "l", xlab = "Time (t)", ylab = "Abundance (N)",
        ylim = c(0, max(results$N)))
}, tVal = slider(min = 10, max = 1000, initial = 100, label = "Total timesteps (t)"),
    rVal = slider(min = 0, max = 4, initial = 1.3, step = 0.1, label = "Growth rate (R)"),
    KVal = slider(min = 1, max = 20, initial = 10, step = 1, label = "Growth rate (K)"),
    N0Val = slider(min = 0, max = 10, initial = 0.1, label = "Initial abundance (N0)"))
```

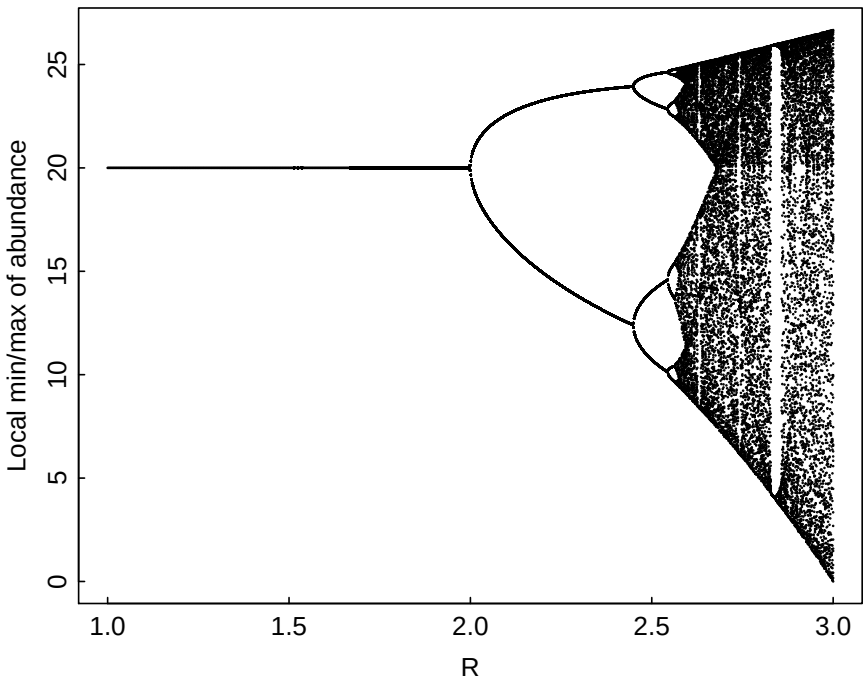
Answer:

Increasing `R` destabilizes the dynamics as the population increasingly over- and undershoots its carrying capacity `K`. The dynamics shift from a point equilibrium, to limit cycles and eventually chaos.

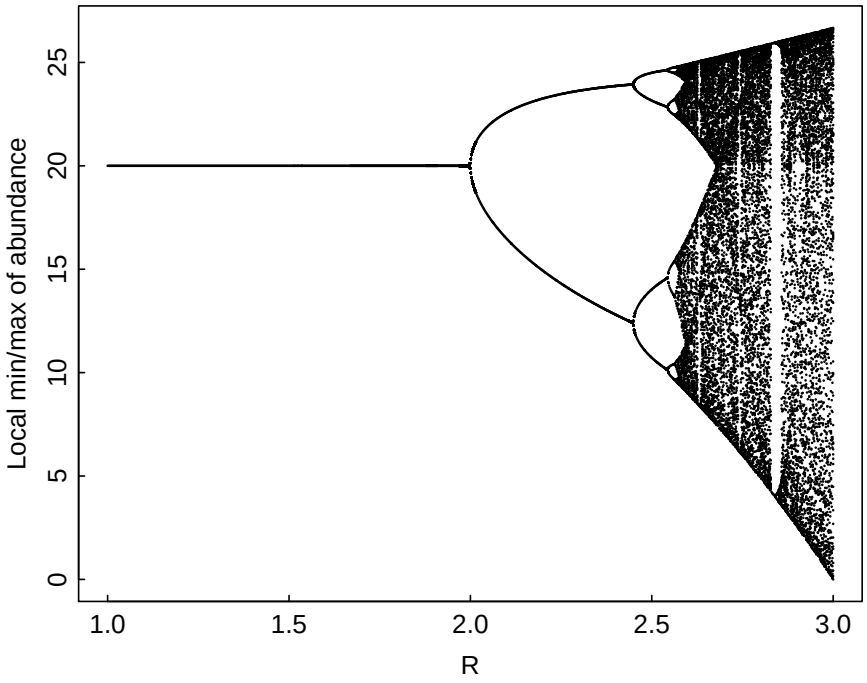
5. If you are feeling adventurous, you can attempt to generate the figure we saw in class for the logistic map with `R` on the x-axis and the min/max of abundance `N` on the y-axis. To do so, you will need to create a function that computes the local min and max of a time series. A local min/max exists when the slope of a time series changes sign. Plotting the local min/max for each `R` value between 1 and 3 will allow you to generate the figure. In this case, assume that `K=20`, `N0=0.5`, and `t=1000` time steps. Conduct the analysis for 1000 linearly spaced values of `R` between 1 and 3 (*Hint*: use function `seq`). Below is an example of local mins (blue) and maxs (red) for a random time series:



```
# First solution:
library(synchrony)
simulate.discrete.logistic <- function(N0, t, K, R) {
    N <- numeric(len = t)
    N[1] <- N0
    for (i in 2:t) {
        N[i] <- (1 + R * (1 - N[i - 1]/K)) * N[i - 1]
    }
    return(data.frame(time = 1:t, N = N))
}
timeval <- 1000
N0 <- 0.5
K <- 20
Rvals <- seq(from = 1, to = 3, len = 1000)
analysis.period <- (timeval - 100):timeval
results <- c()
for (j in 1:length(Rvals)) {
    abunds <- simulate.discrete.logistic(N0, timeval, K, Rvals[j])
    min.max <- try(find.minmax(abunds[analysis.period, ]), silent = TRUE)
    if (class(min.max) != "try-error") {
        new.results <- cbind(Rvals[j], c(min.max$mins$val, min.max$max$val))
        results <- rbind(results, new.results)
    } else {
        results <- rbind(results, cbind(Rvals[j], abunds$N[timeval]))
    }
}
results <- data.frame(R = results[, 1], vals = results[, 2])
plot(results$R, results$vals, xlab = "R", ylab = "Local min/max of abundance",
    cex = 0.08)
```



```r
# Second solution:
find.minmax <- function(t, y) {
    diffs <- sign(diff(y))
    # Return TRUE if local min/max found or FALSE if it's not
    locs <- ifelse(diffs[1:(length(diffs) - 1)] == diffs[2:length(diffs)],
        FALSE, TRUE)
    mins.maxs <- y[c(FALSE, locs)]
    mins.maxs.time <- t[c(FALSE, locs)]
    return(list(minmax = mins.maxs, times = mins.maxs.time))
}
simulate.discrete.logistic <- function(N0, t, K, R) {
    N <- numeric(len = t)
    N[1] <- N0
    for (i in 2:t) {
        N[i] <- (1 + R * (1 - N[i - 1]/K)) * N[i - 1]
    }
    return(data.frame(time = 1:t, N = N))
}
timeval <- 1000
N0 <- 0.5
K <- 20
Rvals <- seq(from = 1, to = 3, len = 1000)
analysis.period <- (timeval - 100):timeval
results <- c()
for (j in 1:length(Rvals)) {
    abunds <- simulate.discrete.logistic(N0, timeval, K, Rvals[j])
    min.max <- find.minmax(abunds$time[analysis.period], abunds$N[analysis.period])
    if (length(min.max$minmax) == 0)
        min.max$minmax <- abunds$N[NROW(abunds)]
    new.results <- cbind(Rvals[j], unique(min.max$minmax))
    results <- rbind(results, new.results)
}
results <- data.frame(R = results[, 1], vals = results[, 2])
plot(results$R, results$vals, xlab = "R", ylab = "Local min/max of abundance",
    cex = 0.08)
```


Local min/max of abundance
0
5
10
15
20
25
1.0
1.5
2.0
2.5
3.0
R