EEB319 Lab 2: Grizzly Bear Population Dynamics

Name:

Lab Section:

Goals:

- Introduce you to R and RStudio, which you'll be using throughout this course to apply theoretical concepts from class and analyse data
- Analyse the population dynamics of grizzly bears, and in particular, their risk of extinction

Marking: Seven questions, totaling 100 points.

Introduction to R and RStudio

R is a computer programming language and environment for statistical computing and graphics. R is open-source, which means you can use it for free, and is maintained and updated by its users, who can write and upload "packages", which are units of sharable code that extend R's basic toolkit. As of June 2019, CRAN (Comprehensive R Archive Network), hosts over 14,000 packages, which means that if you're working on a problem, another R user likely has a solution! For example, you can use a package called "lme4" to fit linear or generalized linear mixed-effects models, "popbio" to construct and analyse projection matrix models, "ggmap" to visualize spatial data on maps, and "wesanderson" to make your plots look like they were taken from the set of Moonrise Kingdom!

Resources:
R Project
Big Book of R

Downloading R and RStudio

- To download R, go to www.r-project.org, and download the latest version of R that is compatible with your OS.
- To download RStudio, go to rstudio.org and download the latest version of RStudio that is compatible with your OS.

Once RStudio is downloaded and installed, open the application. You should see four panes: the top-left pane is your editor, where you can write, run, and save scripts. The bottom-left pane is your console/terminal, where you can run code without saving it in a script. The top-right pane is your environment, where you can find any objects you've defined. The bottom-right pane is where you can navigate between files in your working directory, view plots you've generated, or read package documentation.

R Markdown and .Rmd files

This document, along with most of the labs in this course, and your assignment submissions, will be written using R Markdown. R Markdown is a file format for making dynamic documents (documents that can be continually edited and updated) with R. R Markdown documents are written in markdown (an easy-to-write plain text format, similar to HTML and LaTex (*pronounced "law-tech"*)) and contain chunks of embedded R code. More generally, writing in markdown allows you to create documents that smoothly integrate chunks of text, mathematical equations, hyperlinks, chunks of code, and so on. You can open and edit R Markdown files (`.Rmd`) in R Studio. To learn more about R Markdown, go here, or here.

AN IMPORTANT NOTE: When we write Markdown documents, we can export `.Rmd` files to PDF, Word, or HTML via a process called "knitting". You should be able to knit this file to HTML by clicking "knit" at the top left of this page. Once R has finished knitting this file to HTML, a preview window will pop up. You can view the knitted document in HTML by clicking "Open in Browser". Opening this document in a browser will also show you how the formatting of a `.Rmd` file maps onto the knitted version of the document.

ANOTHER IMPORTANT NOTE: As previously mentioned, Markdown documents are neat because they can integrate chunks of text with chunks of code. To learn how we integrate a chunk of code, check of the example we use to set a working directory, below. Take a look at the "eval" argument. If we set `eval = TRUE`, R runs that chunk of code when knitting a document. If we set `eval = FALSE`, R won't run that chunk of code, but will include the chunk in the knitted document. In the `setwd()` example below, we've set `eval = FALSE` because we have different computers, and as such, the line we've written won't run on your computer. In upcoming labs, you'll notice some code chunks wherein we've set `eval = FALSE`. After filling in the missing code, denoted by "…", change `eval = TRUE` before knitting. When you submit assignments, you should make sure your code chunks are path independent (can be run on computers other than your own (you shouldn't have to worry about this if you've set your working directory correctly)), and set `eval = TRUE`. You can check if your script runs from top to bottom by clicking the drop-down menu, "Run", in the top-right of the editor window (top-left window in default layout), and finding "Run All".

Here's an R Markdown Cheatsheet, for all your formatting needs.

Getting started

Setting your working directory

We use working directories to tell R where it should look for and store files on our computers. We can start by creating a folder called "R_Projects", and a subfolder called "EEB319" on our desktops. If you right-click on that subfolder, you should be able to see its pathname (e.g. `/Users/mjarviscross/Desktop/R_Projects/EEB319`).

There are two ways to set your working directory:

`setwd()`: Use `setwd()` to set your working directory by including the pathname of your EEB319 folder:

```
setwd("/Users/mjarviscross/Desktop/R_Projects/EEB319")
```

This is the easier option, but not the best…

Start a "Project": If you set your working directory using `setwd()`, you'll be able to save all your files in the same place, but won't be able to save any of the objects or outputs you create. If you set your working directory by creating an R Project, you'll be able to save all your files in the same place *and* save all your objects and outputs. This will save you a lot of time in the long-run, and make your work more reproducible.

Go to the top-right corner of your R Studio window and click "Project: (None)". Then, click the first option in the drop-down menu, "New Project". A new window will pop-up, and give you three options: "New Directory", "Existing Directory", and "Version Control". Click "Existing Directory", and use the browse button to navigate to your "EEB319" folder. Finally, click "Create Project" in the bottom-right of the pop-up window, and you're ready to go! As a side-note, you should be able to see all the files and subfolders in your working directory in the "Files" tab of your "Files, Plots, Packages, Help, and Viewer" window of R Studio (bottom-right window in default layout).

NOTE: You can check if you've set your working directory correctly by typing `getwd()` in the console and clicking "return".

Objects

Like C++, Python, and Java, R is an object-oriented programming language. Object-oriented programming is a programming paradigm that organises software design around data, or objects, rather than functions and logic. As an example, types of objects can include:

- Variables: A single value or character
- Vectors: A list of values or characters
- Dataframes: A matrix of values
- Functions: Used in conjunction with arguemnts to perform an operation

Variables, vectors, and assignments

To assign a variable, use `<-`. You can generate this symbol using `option` + `-`. For example:

```
name_1 <- 8; name_2 <- 4
name_1; name_2
```

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

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

You can perform operations by calling variables, as follows:

```
name_3 <- name_1/name_2
name_3
```

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

You can assign a vector as follows:

```
num_vec <- c(1, 2, 3)
char_vec <- c("one", "two", "three")
num_vec; char_vec
```

```
## [1] 1 2 3
```

```
## [1] "one"   "two"   "three"
```

And perform operations as follows:

```
num_vec_transform <- num_vec*2
num_vec_transform
```

```
## [1] 2 4 6
```

```
num_vec_concat <- c(num_vec, c(4, 5, 6))
num_vec_concat
```

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

Functions

In R, functions are self-contained modules of code that accomplish a specific task. When you call a function and provide its required arguments, the function will complete its task, and gives you an output.

Let's use the function `mean()` to take the average of one of our vectors:

```
num_vec_concat_mean <- mean(num_vec_concat)
num_vec_concat_mean
```

```
## [1] 3.5
```

You can also write user-defined functions using the following general syntax:

```
function_name <- function(argument){
  statement
}
```

As an example:

```
LinFunc <- function(x){
  y <- 2*x + 4
  return(y)
}

# The function needs us to give a value of x. If we say x = 2, then the function calculates y according to the st
atment we wrote, y = 2x + 4
LinFunc(2)
```

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

```
# Applying the function to a vector
vec_x <- c(seq(1, 10, 1))
vec_y <- LinFunc(vec_x) # Store outputs in a new vector
vec_y
```

```
##  [1]  6  8 10 12 14 16 18 20 22 24
```

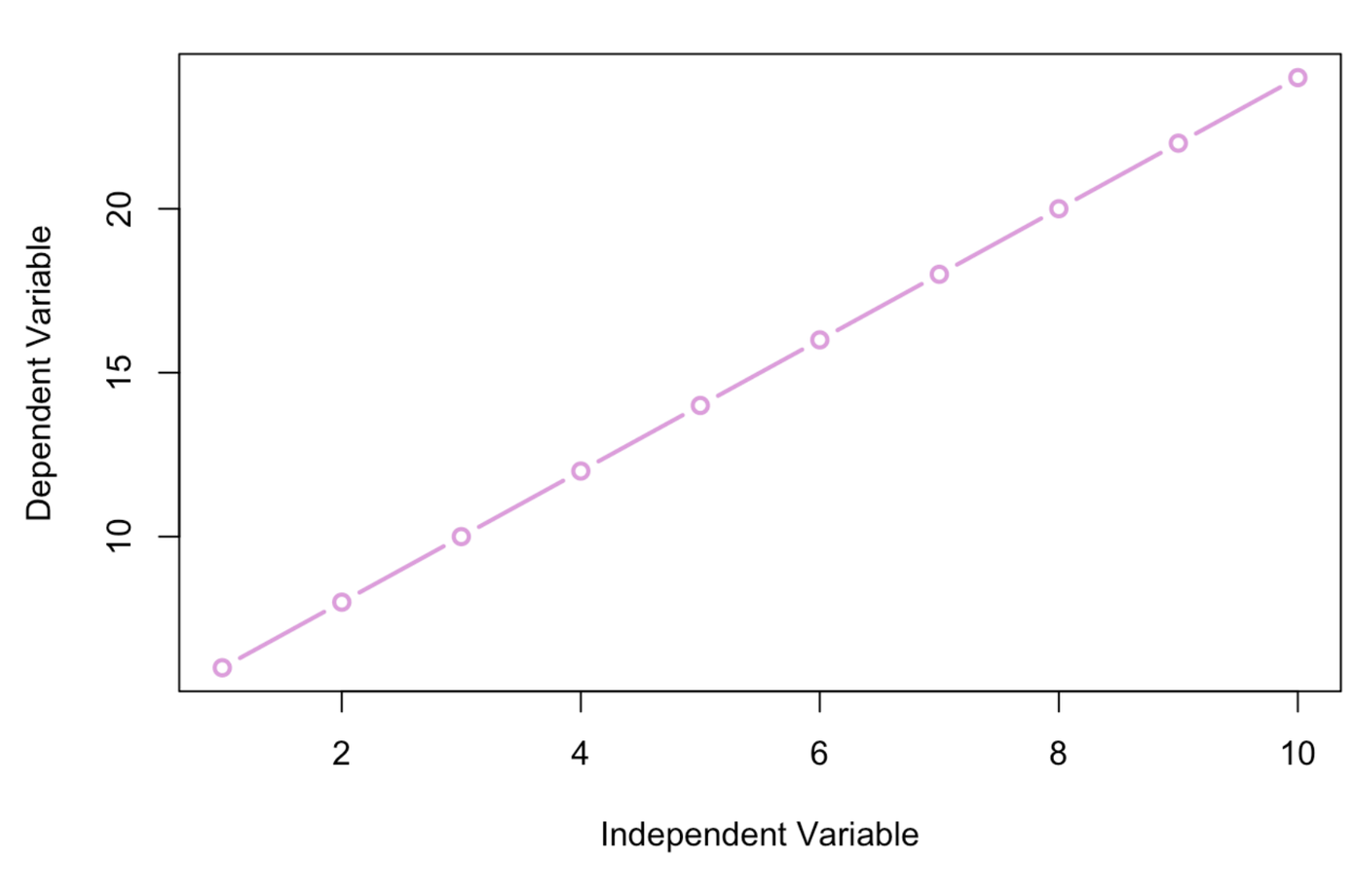
Plotting

We can plot our data using the function, `plot()`. We need to specify the data we want to use for the x-axis first, followed by the data we want to use for the y-axis.

We can use a few additional arguments to make aesthetic changes to our plot:

- `type` changes the line type
- `col` changes the colours of our data
- `lwd` changes the line width
- `xlab` and `ylab` change the names of our axes

```
plot(vec_x, vec_y, type = "b", col = "plum", lwd = 2, xlab = "Independent Variable", ylab = "Dependent Variable")
```



Loops

Loops are used to repeat a chunk of code some number of times. For loops are easy to write once you get the hang of them, and can be used to save you a lot of time! The general syntax for a for loop is as follows:

```
for (i in sequence){
  perform some operation
}
```

Here, `i` is what we call the "iterator", and `perform some operation` is the "iterable". As an example:

```
vec <- c() # Makes an empty vector to store the outputs of our for loop in
for (i in 1:10){
  vec[i] <- i + 1 # The value at position i in our vector is i + 1
}
# Means that the value at position 1 in our vector is 1 + 1, the value at position 2 in our vector is 2 + 1, etc.

vec # To show our resulting vector
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

if… else statements

if… else statements are conditional statements that are used to evaluate an input, and perform different tasks based on whether the input passes some test or not (a kind of Boolean (TRUE/FALSE) test). The general syntax for an if… else statement is as follows:

```
if (test_expression){
  statement
}

if (test_expression){
  statement
} else{
  statement2
}
```

As an example:

```
test <- 7 # Assign "test" some value
if (test > 2){
  print("It's a bones day")
}
```

```
## [1] "It's a bones day"
```

```
if (test == 5){
  print("It's a bones day!")
} else {
  print("It's a no bones day!")
}
```

```
## [1] "It's a no bones day!"
```

Combining for loops and if… else statements

For loops and if… else statements are often used together by nesting the if… else statement within the for loop.

```
vec_1 <- c(seq(1:10)) # Makes a vector with values 1-10
vec_2 <- c() # Another empty vector to store the outputs of our for loop in
for (i in vec_1){
  if (vec_1[i] <= 5){
    vec_2[i] <- "It's a bones day!"
  } else {
    vec_2[i] <- "It's a no bones day!"
  }
}
vec_2
```

```
##  [1] "It's a bones day!"    "It's a bones day!"    "It's a bones day!"
##  [4] "It's a bones day!"    "It's a bones day!"    "It's a no bones day!"
##  [7] "It's a no bones day!" "It's a no bones day!" "It's a no bones day!"
## [10] "It's a no bones day!"
```

Learning Resources:

- Introduction to R by DataCamp
  - A free interactive course with modules to teach you the basics, vectors, matricies, factors, data frames, and lists
  - 4 hour time commitment
- Tips and Tricks in R Studio and R Markdown
  - A really great article from Towards Data Science (generally a fantastic resource!) with some tips on how to write R scripts more efficiently
- Stack Overflow
  - A Q&A website for asking and answering coding questions
  - If you're having trouble figuring out how to do something, or can't figure out why your code isn't running, check Stack Overflow! It's likely that someone's already asked and answered your question!

Now, let's start the lab!

Downloading and importing data

Download the excel file, `Bears.xlsx` from Quercus. Open the file on your computer, and take a look. The file should contain a time series of the abundance of adult female grizzly bears in Yellowstone National Park, from 1959 to 1978. Re-save it as a `.csv`.

To import `Bears.csv` into R, use the following command. Make sure you include the correct pathname. On a PC (Windows OS), right click the file, and click "`Copy as path`". Alternatively, select the file and hold ctrl + shift + C. On a Mac, right click the file, hold down the "option" key, and click "`Copy "Bears.csv" as pathname`". There are a couple different ways to import data in R. Feel free to explore your options!

Change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

```
BearsData <- read.csv("Bears.csv")
```

Exercise 1

10 Points

Plot the abundance of female grizzly bears through time. Use black circles with white fill as your symbol on the plot and connect the symbols with a black line. Remember to label your axes, make your axis labels easily readable, and include a figure caption. Describe in words the trend in abundance. **HINT:** To select a column from a dataframe, use `name_of_dataframe$name_of_column`.

```
# Write your code here.
```

Exercise 2

10 Points

Using the code provided below, calculate a time series of annual geometric population growth rates, where:

$$\lambda_t = N_{t+1}/N_t$$

Next, calculate a time series of annual exponential population growth rates, where:

$$r_t = ln(\lambda_t)$$

Change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

Question:

1. What is the mean of the exponential population growth rates, and what is the standard deviation? **HINT:** Use the functions `mean()` and `sd()`. To learn how to use these functions, type `?mean()` / `?sd()` into your console.

Type your answer here.

```
## Annual geometric population growth rates
N <- BearsData$N # Vector of population abundances
lambda <- c() # Empty vector to store lambda values

for (i in 1:length(N)-1) {
  lambda[i] <- ((N[i + 1])/N[i])
}
lambda

## Annual exponential population growth rates
r <- log(lambda)
r

## Mean and standard deviation
# Write your code here
```

Exercise 3

15 Points

A simple exponential growth model (without stochasticity, density dependence) for grizzly bear population dynamics is as follows:

$$N_{t+1} = N_t e^r$$

Where r is the mean exponential population growth rate. Simulate this model for 50 years using an initial population size of 44 at t_0, and plot the resulting time series.

HINT: Include a horizontal line denoting a population size of 20 by pasting the following line of code below your `plot()` function: `abline(h = 20, col = "red")`.

Change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

Questions:

1. What is the final population size at t_{60}?
2. How long will it take the population to decline to less than 20 bears?

Type your answers here.

```
## Model simulation
N0 <- 44 # Initial population size
first_year <- BearsData$year[nrow(BearsData)] + 1
years <- seq(first_year, first_year + 59, 1) # Future years as a sequence
N <- c()
N[1] <- N0 # Assigning the initial population size to the first position in the empty vector

for (i in 2:length(years)) {
  N[i] = N[i - 1]*exp(r_mean)
}
N

## Plot your results here
```

Exercise 4

15 Points

Here, we'll analyze how the mean and variance of the predicted population size change through time. Plot the predicted mean population size +/- 1 standard deviation from t_1 to t_{60} with an initial population size of 44 at time t_0.

Change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

Question:

1. What is happening to the range of uncertainty around the predicted population size as time goes on? Why might we observe this phenomenon?

```
N0 <- 44
first_year <- BearsData$year[nrow(BearsData)] + 1
years <- seq(first_year, first_year + 59, 1)
N_1 <- c()
N_1[1] <- N0

for (i in 2:length(years)) {
  N_1[i] = N_1[i - 1]*exp(r_mean + r_sd)
}
N_1

N_2 <- c()
N_2[1] <- N0

for (i in 2:length(years)) {
  N_2[i] = N_2[i - 1]*exp(r_mean - r_sd)
}
N_2

## Plot your results here
```

Exercise 5

15 Points

A stochastic model of grizzly bear population dynamics may look something like this:

$$N_{t+1} = N_t e^{r+\epsilon_t}$$

Conduct a stochastic simulation of the model from t_1 to t_{60} with an initial population size of 44 at time t_0. We can use the function `rnorm()` to select a random variable, ϵ_t, from a distribution with a mean and standard deviation you can specify.

P.S. We can use the function, `set.seed()`, to standardize our selection between implementations.

Change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

```
set.seed(2) # Comment this out to see how model results change because of stochasticity

N0 <- 44
first_year <- BearsData$year[nrow(BearsData)] + 1
years <- seq(first_year, first_year + 59, 1)
N <- c()
N[1] <- N0

# Error is normally distributed with mean = 0 and standard deviation = sd
epsilon <- rnorm(length(years), mean = 0, sd = r_sd)

for (i in 2:length(years)) {
  N[i] = N[i - 1]*exp(r_mean + epsilon[i - 1])
}
N

## Plot your results here
```

Exercise 6

20 Points

Suppose we set 20 as a critical minimum population size, below which the population cannot persist. We are interested in estimating the probability that the population will drop below this threshold over a time period of 60 years. Using the model in Exercise (5), estimate this probability by conducting 100 simulations and counting the number of simulations within which the population size dips below 20.

Finally, use the provided code to plot the original empirical time series, and all 100 population trajectories.

Replace each "…" with code, and change `{r}, eval = FALSE` to `{r}, eval = TRUE` before knitting to submit.

```
epsilon_list <- list() # Empty list
for (i in 1:...) {
  # Select error terms for each year, in each of 100 simulations
  epsilon_list[[i]] <- rnorm(..., ..., ...)
}

N_list <- list() # Empty list
Below20 <- c()

## Simulation
for (i in 1:...){
  # Picking epsilon and setting up abundance vector
  N <- c(...)
  epsilon <- epsilon_list[[i]]
  # Simulation of population growth model
  for (j in 2:length(years)){
    N[j] <- N[j - 1] * exp(... + ...[j - 1])
    N_list[[i]] <- N
  }
  # If... else statement to test if population falls below 20 individuals
  if (TRUE %in% (N_list[[i]] < ...)) {
    Below20[i] <- "..."
  }
  else {
    Below20[i] <- "..."
  }
}

## Count the number of simulations in which population dips below 20
length(Below20[Below20 == "..."])

## Plotting
plot(BearsData$year, BearsData$N, type = "l", xlab = "Year", ylab = "Female Grizzly Bear Population Abundance", x
lim = c(1959, 2038), ylim = c(0, 200), lwd = 2, col = "red3")
for (i in 1:length(N_list)){
  lines(years, N_list[[i]])
}
legend("topleft", legend = c("Empirical: 1959 to 1978", "Simulated: 1979 to 2038"), lty = 1, lwd = 2, col = c("re
d3", "black"))
```

Exercise 7

15 Points

For the set of 100 simulations above, calculate the average population size at t_{60} and the 95% confidence intervals around this estimate.

HINT: You can use a for loop to find population sizes at t_{60} for each simulation. Remember that we use double square brackets, `[[i]]`, to index a list, and single square brackets, `[j]`, to index vectors. You can use the `quantile()` function to calculate 95% confidence intervals.

```
# Write your code here
```