



THE UNIVERSITY OF
CHICAGO BIOLOGICAL SCIENCES

BSD qBio⁵ Boot Camp @ MBL



September 8–15, 2019
Marine Biological Laboratory
Woods Hole, MA

Group logos by Stephanie Palmer

MBL Village Campus



- | | |
|-----------|--|
| 1 | Candle House (Administration) |
| 2 | 100 Water Street (Pierce Exhibit Center, Satellite Club, MBL Club) |
| 3 | Marine Resources Center |
| 4 | Collection Support Facility |
| 5 | Crane Wing (Labs, Shipping) |
| 6 | Lillie Laboratory (Labs, Service Shops, MBLWHOI Library) |
| 7 | Rowe Laboratory (Labs, Speck Auditorium) |
| 8 | Environmental Sciences Laboratory |
| 8a | Homestead Administration (Human Resources, Education Dept.) |
| 9 | Loeb Laboratory (Research and Teaching Labs, Lecture Rooms) |
| 10 | Brick Apartment House |
| 11 | Veeder House Dormitory |
| 12 | David House Dormitory |
| 13 | Broderick House (IT) |
| 14 | Crane House |
| 15 | Swope Center (Registration, Cafeteria, Dormitory, Meigs Room) |
| 16 | Ebert Hall Dormitory |
| 17 | Drew House Dormitory |
| 18 | 15 North Street (Carpentry Shop) |
| 19 | Smith Cottage and Barneck Road Property |
| 20 | C.V. Starr Environmental Sciences Laboratory |
| 21 | 11 North Street |
| | MBL Parking |

Contact Information

Directors

- **Stefano Allesina**
Professor, Ecology & Evolution
sallesina@uchicago.edu
- **Stephanie Palmer**
Associate Professor, Organismal Biology and Anatomy
sepalmer@uchicago.edu
- **Vicky Prince**
Dean for Graduate Affairs/Professor, Organismal Biology and Anatomy
vprince@uchicago.edu

Administrators

- **Diane Hall**
Associate Dean for Graduate Affairs
djh8@uchicago.edu (773-383-4930)
- **Melissa Lindberg**
Graduate Student Affairs Administrator
mlindber@bsd.uchicago.edu (773-817-5751)

Instructors

- **Peter Carbonetto**
Computational scientist, Research Computing Center
pcarbo@uchicago.edu
- **A. Murat Eren**
Assistant Professor, Medicine
meren@uchicago.edu
- **Patrick J. La Riviere**
Associate Professor, Radiology
pjlarivi@uchicago.edu
- **Matthias Steinrücken**
Assistant Professor, Ecology & Evolution
steinrue@uchicago.edu

Course Assistants

- **Evan Kiefl**
Biophysics
ekiefl@uchicago.edu
- **Graham Smith**
Computational neuroscience
grahams@uchicago.edu
- **Grace Hansen**
Genetics, Genomics and Systems Biology
gthansen@uchicago.edu
- **Jill Rosenberg**
Cancer biology
jnrosenberg@uchicago.edu
- **Marie Greaney**
Neurobiology
mgreaney@uchicago.edu
- **Paula Lemos Da Costa**
Postdoc, Ecology & Evolution
plemos@uchicago.edu
- **Zachary Miller**
Ecology & Evolution
zachmiller@uchicago.edu

BSD qBio⁵ @ MBL

General Schedule

Sunday, September 8

3:00-4:30	Check-in
4:45-5:30	Team building activities
5:30-6:30	Dinner
7:15-8:30	Welcome to qBio5 - Allesina and Palmer

Monday, September 9

8:30-10:00	Basic comp. I
10:00-10:30	Coffee break
10:30-12:00	Basic comp. I
12:00-12:45	MBL Orientation + Lunch
1:00-2:30	Basic comp. II
2:30-2:45	Coffee break
2:45-4:30	Basic comp. II
5:00-6:30	Dinner
7:15-8:00	Welcome to UChicago Biosciences - Prince

Tuesday, September 10

8:30-10:00	Boat tour (<i>A.thaliana C.elegans</i>) Free time (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>) Microscopy and ImageJ (<i>S.aegyptiacus T.roseae T.thermophila X.laevis</i>) Visit Marine Resources Center (<i>D.melanogaster D.rerio</i>)
10:00-10:30	Coffee break
10:30-12:00	Boat tour (<i>D.melanogaster D.rerio</i>) Free time (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>) Microscopy and ImageJ (<i>S.aegyptiacus T.roseae T.thermophila X.laevis</i>) Visit Marine Resources Center (<i>A.thaliana C.elegans</i>)
12:00-1:00	Lunch
1:00-2:30	Data visualization (<i>C.jacchus D.rerio P.polytes T.thermophila</i>) Defensive programming (<i>D.melanogaster M.musculus O.bimaculoides S.aegyptiacus</i>) Reproducibility of data analysis (<i>A.thaliana C.elegans T.roseae X.laevis</i>)
2:30-2:45	Coffee break
2:45-4:30	Data visualization (<i>C.jacchus D.rerio P.polytes T.thermophila</i>) Defensive programming (<i>D.melanogaster M.musculus O.bimaculoides S.aegyptiacus</i>) Reproducibility of data analysis (<i>A.thaliana C.elegans T.roseae X.laevis</i>)
5:00-6:30	Dinner
7:15-9:00	MBL Lightening Talks

Wednesday, September 11

8:30-10:00	Boat tour (<i>C.jacchus M.musculus</i>) Free time (<i>S.aegyptiacus T.roseae T.thermophila X.laevis</i>) Microscopy and ImageJ (<i>A.thaliana C.elegans D.melanogaster D.rerio</i>) Visit Marine Resources Center (<i>O.bimaculoides P.polytes</i>)
10:00-10:30	Coffee break

10:30-12:00	Boat tour (<i>O.bimaculoides P.polytes</i>) Free time (<i>S.aegyptiacus T.roseae T.thermophila X.laevis</i>) Microscopy and ImageJ (<i>A.thaliana C.elegans D.melanogaster D.rerio</i>) Visit Marine Resources Center (<i>C.jacchus M.musculus</i>)
12:00-1:00	Lunch
1:00-2:30	Data visualization (<i>A.thaliana M.musculus O.bimaculoides X.laevis</i>) Defensive programming (<i>C.elegans C.jacchus P.polytes T.roseae</i>) Reproducibility of data analysis (<i>D.melanogaster D.rerio S.aegyptiacus T.thermophila</i>)
2:30-2:45	Coffee break
2:45-4:30	Data visualization (<i>A.thaliana M.musculus O.bimaculoides X.laevis</i>) Defensive programming (<i>C.elegans C.jacchus P.polytes T.roseae</i>) Reproducibility of data analysis (<i>D.melanogaster D.rerio S.aegyptiacus T.thermophila</i>)
5:00-6:30	Dinner
7:15-8:00	Free time

Thursday, September 12

8:30-10:00	Boat tour (<i>S.aegyptiacus T.thermophila</i>) Free time (<i>A.thaliana C.elegans D.melanogaster D.rerio</i>) Microscopy and ImageJ (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>) Visit Marine Resources Center (<i>T.roseae X.laevis</i>)
10:00-10:30	Coffee break
10:30-12:00	Boat tour (<i>T.roseae X.laevis</i>) Free time (<i>A.thaliana C.elegans D.melanogaster D.rerio</i>) Microscopy and ImageJ (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>) Visit Marine Resources Center (<i>S.aegyptiacus T.thermophila</i>)
12:00-1:00	Lunch
1:00-2:30	Data visualization (<i>C.elegans D.melanogaster S.aegyptiacus T.roseae</i>) Defensive programming (<i>A.thaliana D.rerio T.thermophila X.laevis</i>) Reproducibility of data analysis (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>)
2:30-2:45	Coffee break
2:45-4:30	Data visualization (<i>C.elegans D.melanogaster S.aegyptiacus T.roseae</i>) Defensive programming (<i>A.thaliana D.rerio T.thermophila X.laevis</i>) Reproducibility of data analysis (<i>C.jacchus M.musculus O.bimaculoides P.polytes</i>)
5:00-6:30	Dinner
7:15-8:30	Talk Patel

Friday, September 13

8:30-10:00	Statistics for a data rich world (<i>T.roseae T.thermophila X.laevis</i>) Workshop Eren (<i>A.thaliana C.elegans D.rerio</i>) Workshop Palmer (<i>C.jacchus D.melanogaster M.musculus</i>) Workshop Steinrücken (<i>O.bimaculoides P.polytes S.aegyptiacus</i>)
10:00-10:30	Coffee break
10:30-12:00	Statistics for a data rich world (<i>T.roseae T.thermophila X.laevis</i>) Workshop Eren (<i>A.thaliana C.elegans D.rerio</i>) Workshop Palmer (<i>C.jacchus D.melanogaster M.musculus</i>) Workshop Steinrücken (<i>O.bimaculoides P.polytes S.aegyptiacus</i>)
12:00-1:00	Lunch
1:00-2:30	Statistics for a data rich world (<i>D.rerio M.musculus S.aegyptiacus</i>) Workshop Eren (<i>D.melanogaster O.bimaculoides T.thermophila</i>) Workshop Palmer (<i>A.thaliana P.polytes T.roseae</i>) Workshop Steinrücken (<i>C.elegans C.jacchus X.laevis</i>)
2:30-2:45	Coffee break

2:45-4:30	Statistics for a data rich world (<i>D.rerio M.musculus S.aegyptiacus</i>) Workshop Eren (<i>D.melanogaster O.bimaculoides T.thermophila</i>) Workshop Palmer (<i>A.thaliana P.polytes T.roseae</i>) Workshop Steinrücken (<i>C.elegans C.jacchus X.laevis</i>)
5:00-6:30	Dinner
7:15-8:30	Professional Development Night with CAs

Saturday, September 14

8:30-10:00	Statistics for a data rich world (<i>C.elegans D.melanogaster P.polytes</i>) Workshop Eren (<i>C.jacchus S.aegyptiacus T.roseae</i>) Workshop Palmer (<i>D.rerio O.bimaculoides X.laevis</i>) Workshop Steinrücken (<i>A.thaliana M.musculus T.thermophila</i>)
10:00-10:30	Coffee break
10:30-12:00	Statistics for a data rich world (<i>C.elegans D.melanogaster P.polytes</i>) Workshop Eren (<i>C.jacchus S.aegyptiacus T.roseae</i>) Workshop Palmer (<i>D.rerio O.bimaculoides X.laevis</i>) Workshop Steinrücken (<i>A.thaliana M.musculus T.thermophila</i>)
12:00-1:00	Lunch
1:00-2:30	Statistics for a data rich world (<i>A.thaliana C.jacchus O.bimaculoides</i>) Workshop Eren (<i>M.musculus P.polytes X.laevis</i>) Workshop Palmer (<i>C.elegans S.aegyptiacus T.thermophila</i>) Workshop Steinrücken (<i>D.melanogaster D.rerio T.roseae</i>)
2:30-2:45	Coffee break
2:45-4:30	Statistics for a data rich world (<i>A.thaliana C.jacchus O.bimaculoides</i>) Workshop Eren (<i>M.musculus P.polytes X.laevis</i>) Workshop Palmer (<i>C.elegans S.aegyptiacus T.thermophila</i>) Workshop Steinrücken (<i>D.melanogaster D.rerio T.roseae</i>)
5:15-8:15	Wrap up and BBQ

Sunday, September 15

10:00-10:30 Leave for Chicago

MBL Talks

Tuesday, Sept 10th

7:15 - 7:45pm Lightning talks!

- Irina Arkhipova
- Karen Echeverri
- Kristin Gribble
- Jessica Mark Welch
- Emil Ruff
- Joe Vallino

7:45 - 8:30pm

David Remsen

Friday, Sept 13th

7:15 - 8:15pm

Nipam Patel

Tutorials

Microscopy and ImageJ

Basic computing I

Basic computing II

Advanced computing I

Advanced computing II

Defensive programming

Data visualization

Reproducibility of data analysis

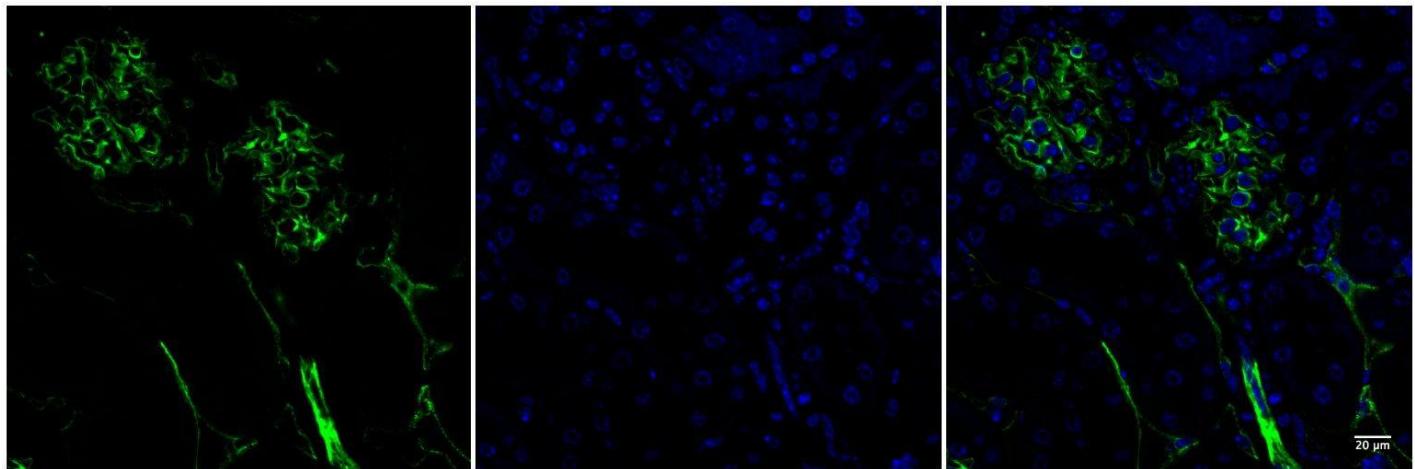
Statistics for a data-rich world

Image Processing with ImageJ Exercises

Using the information in the ImageJ Tutorials and the built-in ImageJ Command Finder tool (open ImageJ / Fiji and type L to get it to pop up), complete the following exercises. You may not be able to complete these exercises in the time you have, but do your best.

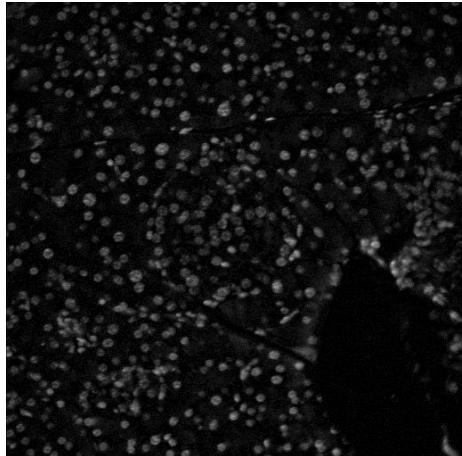
Exercise 1

Use the cd31 glomeruli.tif file to create the montage shown below. The image comes as two images grouped together in a stack, so your first job is to figure out how to separate them. The pixel size for the image is 0.25 microns. Don't forget the 20um scale bar in the lower right hand corner!

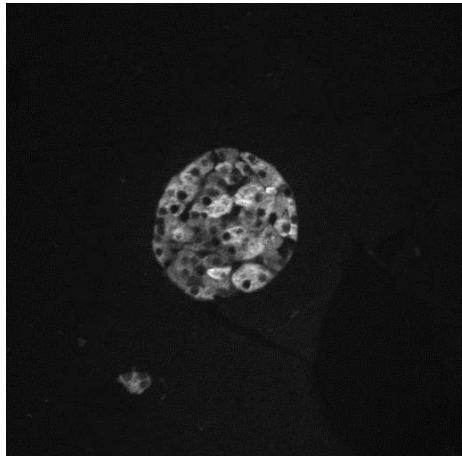


Exercise 2

Use the pancreatic_islet.tif to answer the following questions:



- a) How many cells are in the entire image? One nucleus per cell is marked by the DAPI stain in the first image.
- b) What is the average nuclear area for all the cells in the image? Readout should be in um^2 NOT cm^2 !
- c) How would you create a data table that includes the mean gray value for every nucleus in the image? They are not all 255!
- d) How many beta cells are in the image? Beta cells are represented by the insulin-seeking stain in the second image. Hint: you can still count the nuclei if you find a way to restrict your count to nuclei in an islet.



Bonus Questions (if you have time)

Bonus 1: if there were multiple islets in the pancreatic_islet.tif image, how could you count the number of cells PER islet?

Bonus 2: Create a macro to automate the creation of the cd31 glomeruli montage.

Basic Computing 1 — Introduction to R*

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aim of this workshop is to introduce R (and RStudio), and show how it can be used to analyze data in an automated, replicable way. We will illustrate the notion of assignment and present the main data structures available in R. We will learn how to read and write data, how to execute simple programs, and how to modify the stream of execution of a program through conditional branching and looping. *This workshop is intended for biologists with little to no background in programming.*

Setup

To complete the tutorial, we will need to install R and RStudio. To install R, follow instructions at cran.rstudio.com. Then install Rstudio following the instructions at bit.ly/2JARd4v.

Motivation

When it comes to analyzing data, there are two competing paradigms. First, one could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; second, one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is preferred because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a lab mate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce R, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

What is R?

R is a statistical software that is completely programmable. This means that one can write a program (“script”) containing a series of commands for the analysis of data, and execute them auto-

*This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, MBL, 2019. Current version: July 31, 2019; Corresponding author: sallesina@uchicago.edu.

matically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is free software: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. You can find a list of official packages (which have been vetted by R core developers) at ggplot2.org/S0SDWA; many more are available on GitHub, Bioconductor, and other websites.

The main hurdle new users face when approaching R is that it is based on a command-line interface; when you launch R, you simply open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example,

```
2 * 3  
# [1] 6
```

A little history

R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. The main advantage is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

The main RStudio interface is split up into "panels". The most important panels are:

1. **Console:** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
2. **Source:** In this panel, you can write a program and save it to a file pressing control + S (or command + S). The code can also be executed by pressing control + shift + S (or command + shift + S).
3. **Environment:** This panel lists all the variables you created (more on this later).
4. **History:** This gives you the history of the commands you typed.
5. **Plots:** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (just type `help(name_of_command)` in the Console) and packages.

How to write an R program

An R program is simply a list of commands, which are executed one after the other. The commands are written in a text file (with extension `.R`). When R executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time R encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. Writing a program is advantageous, however, because it can be automated and shared with other researchers. Moreover, after a while you will have accumulated a lot of code, at which point you can reuse much of your code for new projects.

Activity: Collaborative data collection

In this activity, you and the rest of the class will collaborate on creating a data set. Specifically, we will collect data about the laptops of incoming BSD graduate students. We will use these data to learn about R, and how it can be used to analyze a data set; we will write some R code that can be used to answer basic questions such as, what is the most common operating system among the population (where “population” is all the current incoming BSD grad students)? This is also an opportunity to learn about your laptop’s hardware.

The instructor will provide you with further instructions.

Activity: Formulating the questions

Before writing any R code to import and analyze the data set, you first need to determine the aims of working with this data set. In particular, we need to formulate some *guiding questions*. For example, some very basic questions we might want to know are:

- How many samples have been collected?
- What are the different operating systems used?

Or, some more complicated questions we could ask are:

- Which is the most common (and least common) operating system?
- On average, do Mac laptops have more (or less) memory than Windows laptops, and how much more (or less)?

Write down a few more questions guiding your data analysis:

Reading in the data

Your R environment should be empty:

```
ls()
```

Your first step is to create a new object in our R environment that contains the data from the CSV file. Let's call this object "laptops". Write down your R code for reading in the data:

If this was successful, your environment should now contain a single object, "laptops":

```
ls()
```

Inspecting the data

What kind of object is it?

```
class(laptops)
```

It should be a "data frame".

Data frames contain data organized in a table or spreadsheet. The columns typically represent different measurements, and they can be of different types (e.g., date of measurement, weight of individual, volume of cell, treatment of sample). The rows typically represent different individuals or samples.

When you read a spreadsheet or CSV file in R, it is automatically stored as a data frame.

Run some basic commands to inspect the table, and check that the data were read correctly. After running each line of code, add a note next to the line of code summarizing what it did:

```
nrow(laptops)
ncol(laptops)
length(laptops)
head(laptops)
tail(laptops)
summary(laptops)
str(laptops)
```

Accessing rows and columns

There are many ways to inspect parts of the data frame. Here are some examples—as before, add a note next to each line of code explaining what it did.

```
laptops[4, 2]
laptops[4, "os"]
laptops[4, ]
laptops[, 2]
laptops[2]
laptops[, "os"]
laptops$os
laptops["os"]
laptops$os[4]
laptops[4, ]$os
```

Here are some slightly more complicated examples:

```
laptops[1:4, ]
laptops[, 2:3]
laptops[, c("os", "mem")]
laptops[c("os", "mem")]
laptops[c(1:4, 8, 12), c("os", "cores")]
```

Creating—and overwriting—objects

The most basic operation in any programming language is assignment. In R, assignment is marked by the operator `<-`. Above, we used the assignment operator to create a new object, `laptops`, containing the data from the CSV file.

When you type a command in R, it is executed, and the output is printed in the **Console**. For example, run:

```
sqrt(9)
```

If we want to save the result of this operation, we can assign the result to an object, or variable. Let's call this new object “`x`”:

```
x <- sqrt(9)  
x
```

What happened? We wrote a command containing an assignment operator (`<-`). R evaluated the right-hand side of the command (`sqrt(9)`), and stored the result (3) in a newly created variable called `x`. Now we can use `x` in our commands: every time the command needs to be evaluated, the program will look up which value is associated with the variable `x`, and substitute it. For example, we can run:

```
2 * x
```

We can then use `x` to create a new object storing the result of multiplication:

```
y <- 2 * x  
y
```

Or we can *overwrite* the existing value of `x`:

```
x <- 2 * x  
x
```

Observe that both assignment (creating new objects) and overwriting both use `<-`. So be careful—*there is no undo command in R!*

Next, we will use the assignment operator to extract individual columns from the data frame, and understand these columns in more detail.

Side note: “`<-`” versus “`=`”

A common point of confusion is that the equality symbol (`=`) *can also be used for creating and overwriting objects*. And many people prefer to use this.

But we strongly recommend against using `=` for assignment. We have two reasons:

1. The `=` is easily confused with `==`, which does something completely different (as we will see soon)—it is used to check that two values are equal.
2. The `=` is also used elsewhere for named arguments to functions (we saw one example of this when we called `read.csv`). This use of “`=`” in this case is *not* assignment (consider that no objects are created or overwritten in your environment).

Basic data types and operations: the basic building blocks of R

A data frame is a *composite* data structure—it is made of simpler data objects which can be thought of as R “building blocks”. Here we will take a close look at these building blocks.

Text data

What columns of `laptops` contains text data? Write a line of code to select a text column, and assign it to a new variable, `x`:

Then check the data type of `x`:

```
class(x)
```

In R, the `character` data type is used to store text.

You can access individual elements by their index: the first element is indexed at 1, the second at 2, etc.

```
x[1]  
x[5]
```

R has many built-in functions for operating on text data. Here are some examples (add your notes explaining what each line of code does):

```
nchar(x)  
toupper(x)  
sort(x)  
unique(x)  
table(x)  
paste(x[1], x[2])
```

Just as data types can be combined to form more complex data structures, operations can also be combined:

```
unique(toupper(x))
```

Numeric data

What columns of `laptops` contain numeric data? Write code to assign a numeric column to `x`, and check its type:

The `integer` data type stores whole numbers.

There is another data type, `numeric`, used to store real numbers (or rather their approximations, as computers have limited memory and thus cannot store numbers like π , or even 0.2), e.g.,

```
y <- x / 10  
class(y)
```

The numeric data type can store very small or very large numbers (what does the `^` operator do?):

```
x^10
```

Like characters, R has many built-in functions for working with numbers, such as:

```
abs(x)  
sqrt(x)  
cos(x)  
log(x)
```

Some functions allow different data types:

```
sort(laptops$mem)  
sort(laptops$os)  
table(laptops$mem)  
table(laptops$os)
```

Given that R was born for statistics, there are many statistical functions you can perform on numeric data (add notes next to these lines explaining what they do):

```
min(x)  
max(x)  
range(x)  
sum(x)  
median(x)  
mean(x)  
summary(x)
```

Finally, standard mathematical operations such as `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation) can be applied to numbers. A possibly unfamiliar operator is the modulo `(%%)`, calculating the remainder of an integer division:

```
x %% 3
```

Question: What is the modulo operator, and could it be useful?

Exercise: Combine some of the functions and/or operators mentioned above to compute the *standard deviation* of `x`, then compare your answer against R's built-in solution, `sd(x)`. *Hint:* Use the definition of variance. Write your answer in the box below.

Logical data

The `laptops` data set does not contain any logical data, but we can easily generate logical data from one of the columns. Write code to create an object `x` containing logical data, and check the type of `x`:

The “logical” data type takes only two possible values `TRUE` and `FALSE`.

Question: Why are two equal signs used to check that two numbers are equal?

Exercises:

1. Besides equality, other comparison operators include `>` (greater than), `<` (less than), `!=` (differs) and `>=` and `<=`. Write code using one or more of these operators to create some other (hopefully useful!) logical data:

2. Additionally, you can formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write code using these operators, as well as the ones above, to generate some new (and hopefully useful!) logical data:

Using numeric data to select rows of a data frame

Above, we saw that rows and columns can be selected by number`x`. The row and column numbers are just numeric data—therefore, *you can think of the selection of rows as data*. Let’s take an example: suppose you would like to inspect only data on Mac laptops. You can do this by using the `which` function to create a new numeric data object containing the row numbers corresponding to Mac laptops:

```
rows <- which(laptops$os == "mac")
laptops[rows, ]
```

Exercise: Create a new `rows` object based on a logical expression involving two columns of the `laptops` data frame. Write two different ways of implementing this row selection:

Categorical data: factors

There is one last basic data type that isn't present in the `laptops` data. But we can easily create it using a column of the data frame. This last data type is a *factor*. Write down your R code for creating a factor:

Inspect the factor:

```
x  
summary(x)
```

Question: What is a factor? When would a factor be useful?

Manipulating data

So far, after reading in the data, we have treated the data frame as a static object. But the data frame is an object like any other in R—it can be modified and overwritten. This is a very powerful—but also very dangerous—idea! Let's illustrate a few of the many kinds of data manipulations we can make, but before doing so, let's create a copy of the data frame in case we make a mistake along the way:

```
laptops2 <- laptops
```

You can overwrite individual entries of the data frame, or several entries at once:

```
laptops2[c(1, 2), "mem"] <- 2
```

Write your code to replace all “mac” values in the “os” column with “macOS”:

Above, we found that factors are sometimes useful. Replace the text data in the “os” column with categorical data:

```
laptops2$os <- factor(laptops2$os)
```

If you don't like the column names, you can change them:

```
colnames(laptops2) <- c("name", "OS", "GB", "CPUs")
```

You can reorder the rows or columns:

```
rows      <- order(laptops2$name)
laptops2 <- laptops2[rows, ]
laptops2 <- laptops2[c("name", "CPUs", "GB", "OS")]
```

Or you can even create new columns:

```
n <- nrow(laptops2)
laptops2$sample <- seq(1, n)
```

Save the state of your environment

It is important to periodically save the state of your R environment. Let's do this now.

To save your environment, go to **Session > Save Workspace As** in RStudio, or run this code:

```
save.image("mbl_tutorial.RData")
```

Later, to restore your environment, select **Session > Load Workspace** in RStudio, or run this code:

```
load("mbl_tutorial.RData")
```

Question: In RStudio, what is the difference between **File > Save As** and **Session > Save Workspace As**? Record your answer here:

Group activity: Data analysis, Part 1

You now have the basic tools you need to analyze a data set in R. Return to the guiding questions you (and others) have formulated. In this activity, you will apply your newly acquired R skills to answer some of these questions. (Some of these questions may have already been answered as we worked through the examples above.) Record your answers, as well as the code you wrote to generate these answers.

Group activity: Data analysis, Part 2

Use your code from Part 1 to create a *script*—a text file containing all the code necessary to run the complete data analysis, starting with the `read.csv` call to import the data, and ending with the code used to generate the answers to your guiding questions. Give it a useful name such as `analyze_laptops_data.R`. The purpose of a script is to record your code so that you can easily repeat, or *automate*, the analysis later.

Once you have written this script, make sure that it runs, and save the file somewhere on your computer. Please also add comments (lines starting with "#") explaining in plain language what the code does. The instructor will give you instructions for sharing your script with the rest of the class.

Other topics

Creating data

Above, we imported data into R. In addition to viewing and manipulating existing data, R also has many facilities for creating data structures, either from scratch, or from existing data. (We already saw a few examples of this.)

The most basic tool is the `c` function, which is short for “combine”, and can combine multiple objects or values:

```
x      <- c(2, 3, 5, 27, 31, 13, 17, 19)
sex    <- c("M", "M", "F", "M", "F")           # Sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # Weight (in mg)
```

You can generate sequences of numbers using `seq`. Write R code that uses the `seq` function to generate all odd numbers from 1 to 100:

For simpler sequences of numbers, use the “colon” operator:

```
x <- 1:10
```

To repeat a value (or values) several times, use `rep`:

```
x <- rep("treated", 5) # Treatment status
x <- rep(c(1, 2, 3), 4)
```

Finally, there are many functions for generating random numbers. For example,

```
x <- runif(100)
```

What does `runif(100)` return?

Matrices

A matrix is like a data frame—it also has rows and columns—but a key difference is that all the columns must be of the same type. Here's an example of a 2×2 matrix:

```
A <- matrix(c(1, 2, 3, 4), 2, 2) # Inputs are values, nrows, ncols.
```

In the case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, etc).

```
A %*% A      # Matrix product
solve(A)     # Matrix inverse
t(A)         # Transpose
A %*% solve(A) # This should return the identity matrix.
```

Determine the dimensions of a matrix:

```
nrow(A)
ncol(A)
dim(A)
```

Use indices to access a particular row or column of a matrix:

```
Z <- matrix(1:9, 3, 3)
Z[1, ]          # First row.
Z[, 2]          # Second column.
Z[1:2, 2:3]    # Submatrix with coefficients in rows 1 & 2, and columns 2 & 3.
Z[c(1, 3), c(1, 3)] # Indexing non-adjacent rows and columns.
```

Some operations apply to all the elements of the matrix:

```
sum(Z)
mean(Z)
```

Question: When is it better to store data in a matrix instead of a data frame?

Arrays

If you need tables with more than two dimensions, use arrays:

```
A <- array(1:24, c(4, 3, 2))
```

A matrix is a special case of an array with only 2 dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing you should be paying attention to: R drops dimensions that are not needed. So, if you access a “slice” of a 3-dimensional array, you should obtain a matrix:

```
A[, 2, ]  
dim(A[, 2, ])  
class(A[, 2, ])
```

Lists

You have already worked with a list object (probably without realizing it). The `laptops` data frame is also a list:

```
is.list(laptops)
```

A data frame is a specific type of list—it is a list in which every list item is a vector of the same length. Lists also allow vectors of different lengths, for example, here we add an additional item (“date”) that has a length of 1:

```
mylist <- as.list(laptops)  
mylist$last_modified <- "September 9, 2019"
```

We could not do this in a data frame.

A list is also a very general data structure for creating complex objects, and the list elements can be used to store almost anything.

Missing data

Finally, R allows most types of data—text, numeric, factors, *etc*—to take on a special value, `NA`. This is short for “not available” or “not assigned”, and is commonly called “missing data”. One special feature of R is that it often gracefully handles data sets containing missing data.

Exercise: Set a few entries in the `laptops` data frame to `NA`, then run `summary(laptops)`. How are the missing data reported in the summary?

Group activity: Analyzing genetic data (time permitting)

In this activity, you will apply the tools we have developed so far to look at an example data set from human genetics: genotype data on chromosome 6 collected from European individuals. (Data adapted from the [Human Genome Diversity Project](#) by John Novembre.)

```
chr6 <- read.table("H938_Euro_chr6.geno", header = TRUE)
```

Setting `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
nrow(chr6)  
ncol(chr6)
```

It contains 7 columns, and more than 40,000 rows!

The table reports the number of homozygotes ($nA1A1$, $nA2A2$) and heterozygotes ($nA1A2$), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals. The other columns are:

- CHR: The chromosome (in this case, 6).
- SNP: The identifier of the Single Nucleotide Polymorphism.
- A1: One of the observed alleles.
- A2: The other allele.

Write R code to answer the following questions:

1. How many individuals were sampled? Find the maximum of the sum $nA1A1 + nA1A2 + nA2A2$. *Hint:* Recall that you can access the columns by index (e.g., `chr6[, 5]`), or by name (e.g., `chr6$nA1A1`, or `chr6[, "nA1A1"]`).
2. Use the `rowSums` function to obtain the same answer.
3. How many SNPs have no heterozygotes (*i.e.*, no “A1A2”)?
4. How many SNPs have less than 1% heterozygotes?

Programming Challenge

Instructions

You will work with your group to solve the following exercise. When you have found the solution, go to [stefanoallesina.github.io/BSD-QBio5](#) and follow the link “Submit solution to challenge 1” to submit your answer (alternatively, you can go directly to [goo.gl/forms/dDJKvF0dUi7KUDqp1](#)). At the end of the boot camp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Nobel nominations

The file `nobel_nominations.csv` contains data on Nobel Prize nominations from 1901 to 1964. There are three columns (the file has no “header”): (1) the field (e.g., “Phy” for physics), (2) the year of the nomination, and (3) the id and name of the nominee.

1. Take Chemistry (Che). Who received the most nominations?
2. Find all researchers who received nominations in more than one field.
3. Take Physics (Phy). Which year had the largest number of nominees?
4. For each field, what is the average number of nominees per year? Calculate the number of nominees in each field and year, and take the average across all years.

Hints:

- You will need to subset the data. To make operations clearer, it is helpful to give names to the columns. For example, suppose you imported the data into a data frame called `nobel`. Then `colnames(nobel) <- c("field", "year", "nominee")` should do the trick.
- The simplest way to obtain a count from a vector is to use the `table` function. For example, the command `sort(table(x))` produces a table of the occurrences in `x`, in which the counts are sorted from smallest to largest.
- You can also use the same function, `table`, to build a table using more than one vector. For example, suppose `x` and `y` are vectors of the same length. Then `table(x, y)` will build a table with counts for each unique pair of occurrences in `x` and `y`.
- Some other functions you may find useful for the challenge: `colMeans`, `factor`, `head`, `length`, `max`, `read.csv`, `subset`, `tail`, `tapply`, `unique`, `which` and `which.max`.
- Save your solution code for each exercise in a file.

Basic Computing 2 — Packages, functions and documenting code*

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aims of this workshop are to: (1) learn how to install, load and use the many of the freely available R packages; (2) illustrate how to write user-defined functions, and how to organize your code; use basic plotting functions; and (3) introduce the package `knitr` for writing beautiful reports. *This workshop is intended for biologists with basic knowledge of R.*

Setup

To produce well-documented code, you need to instal the `knitr` package. You will also use the package `MASS` for statistics.

Packages

R is the most popular statistical computing software among biologists due to its highly specialized packages. They are often written by biologists for biologists. You can contribute a package too! The RStudio website (goo.gl/harVqF) provides guidance on how to start developing R packages. See also Hadley Wickham's free online book (r-pkgs.had.co.nz).

You can often find highly specialized packages to address your research questions. Here are some suggestions for finding an appropriate package. The Comprehensive R Archive Network (CRAN) offers several ways to find specific packages for your task. You can browse the full list of CRAN packages (goo.gl/7oVyKC). Or you can go to the CRAN Task Views goo.gl/0WdIcu) and browse a compilation of packages related to a topic or discipline.

From within R or RStudio, you can also call the function `RSiteSearch("KEYWORD")`, which submits a search query to the website search.r-project.org. The website rseek.org casts an even wider net, as it not only includes package names and their documentation, but also blogs and mailing lists related to R. If your research interests relate to high-throughput genomic data, or other topics in bioinformatics or computational biology, you should search the packages provided by Bioconductor (goo.gl/7dwQ1q).

Installing a package

Suppose you want to install the `rsvd` package, which is a package for performing singular value decompositions (SVD) and principal components analysis (PCA) on large data sets. To install the package, run:

*This document is included as part of the Basic Computing 2—Introduction to R tutorial packet for the BSD qBio Bootcamp, MBL, 2019. Current version: July 31, 2019; Corresponding author: sallesina@uchicago.edu.

```
install.packages("rsvd")
```

Or, in RStudio, select the **Packages** panel, then click on *Install*.

Loading a package

Once it is successfully installed, to load the rsvd package into your R environment, run:

```
library(rsvd)
```

Another example

Suppose you would like to access the “bacteria” data set, which reports the incidence of *H. influenzae* in Australian children. The data set is included in the MASS package. If you try to access the data set before loading the package, you will get an error:

```
data(bacteria)
```

First, you need to load the package:

```
library(MASS)
```

Now the data set is available, and you can load it:

```
data(bacteria)
head(bacteria)
```

Random numbers

For many of your analyses, you will need to draw random numbers. (These are actually “pseudo-random” numbers because they are not perfectly random.) In fact, you will need random numbers to implement the analysis of citation data in the “case study” below.

R has many functions to sample random numbers from different statistical distributions. For example, use `runif` to create a vector containing 10 random numbers:

```
runif(10)
```

Question: What kind of random numbers are generated by `runif`? How could you check this?

To sample from a set of values, use `sample`:

```
v <- c("a", "b", "c", "d")
sample(v, 2)                  # Sample without replacement.
sample(v, 6, replace = TRUE)  # Sample with replacement.
sample(v)                     # Shuffle the elements.
```

The normal distribution is one of the most commonly used distributions, so naturally there is a function in R for simulating from the normal distribution:

```
rnorm(3)          # Three draws from the standard normal.  
rnorm(3, mean = 5, sd = 4) # Change the mean and standard deviation.
```

Exercise: The normal distribution has a very recognizable shape. Use `rnorm` to generate a large number of values from the standard normal, then use `hist` to draw a histogram of these values (you can adjust the number of bins in the histogram with the `n` argument). Is the histogram “bell shaped”? Use `mean`, `median` and `sd` to verify that the random numbers recover the expected properties of the normal distribution. Write all your code here:

Writing functions

It is good practice to subdivide your analysis into functions, and then write a short “master” program that calls the functions and performs the analysis. In this way, the code will be more legible, easier to debug, and you will be able to recycle the functions for your other projects.

In R, every function has this form:

```
my_function_name <- function (arg1, arg2, arg3) {  
  #  
  # Body of the function.  
  #  
  return(return_value) # Not required, but most functions output something.  
}
```

Here is a very simple example:

```
sum_two_numbers <- function (a, b) {  
  s <- a + b  
  return(s)  
}  
sum_two_numbers(5, 7.2)
```

In R, a function can return only one object. If you need to return multiple values, organize them into a vector, matrix or list, and return that; e.g.,

```
sum_and_prod <- function (a, b) {  
  s <- a + b  
  p <- a * b  
  return(c(s,p))  
}  
sum_and_prod(5, 7.2)
```

Activity: Probability density function (PDF) of the normal distribution

Write a function, `normpdf`, that accepts two arguments, `x` and `s`, and returns the density at `x` for the normal distribution with zero mean and standard deviation `s`.

Write down the formula for the normal density at x with mean zero and standard deviation s :

Now write your code for the `normpdf` function:

Check that this function gives the correct answers by comparing to the built-in function `dnorm`.

Conditional branching

When we want a block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met) {  
  # Execute this block of code.  
} else {  
  # Execute this other block of code.  
}
```

For example, try running these lines of code (you might want to try running them a few times):

```
x <- rnorm(1)  
if (x < 0) {  
  msg <- paste(x, "is less than zero")  
} else if (x > 0) {
```

```
msg <- paste(x, "is greater than zero")
} else {
  msg <- paste(x, "is equal to zero")
}
print(msg)
```

We have created a conditional branching point, so that the value of `msg` changes depending on whether `x` is less than zero, greater than zero, or equal to zero.

Activity: Improved normal probability density function

The probability density function of the normal distribution is not defined if the standard deviation is less than zero. When the standard deviation is exactly zero, the density is a “spike” at zero; it is `Inf` exactly at $x = 0$, and zero everywhere else. Use the `if` and `else` keywords, following the example above, to write an improved `normpdf` function that returns `NaN` whenever the standard deviation is negative, and returns zero when the standard deviation is exactly zero (unless $x = 0$, in which case it should return `Inf`):

Looping

Another way to change the flow of your program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

R provides you with two ways to loop over code blocks: the `for` loop and the `while` loop. Let’s start with the `for` loop, which is used to iterate over a vector or list; for each value of the vector (or list), a series of commands will be run, as shown by the following example:

```
v <- 1:10
for (i in v) {
  a <- i ^ 2
  print(a)
}
```

In the code above, the variable `i` takes the value of each element of `v` in sequence. Inside the block within the `for` loop, you can use the variable `i` to perform operations.

The anatomy of the `for` statement:

```
for (variable in list_or_vector) {
  execute these commands
} # Automatically moves to the next value.
```

You should use a `for` loop when you know that you want to perform the analysis over a given set of values (e.g., files of a directory, rows in your data frames, sequences of a fasta file, etc).

The `while` loop is used when the commands need to be repeated while a certain condition is true, as shown by the following example:

```
i <- 1
while (i <= 10) {
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

The script gives exactly the same result as the `for` loop above. A key difference is that you need to include a step to update the value of `i`, (using `i <- i + 1`), whereas in the `for` loop it is done for you automatically. The anatomy of the `while` statement:

```
while (condition is met) {
  execute these commands
} # Beware of infinite loops... remember to update the condition!
```

You can break a loop using `break`. For example:

```
i <- 1
while (TRUE) {
  if (i > 10) {
    break
  }
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

Question: Above, we ran three different loops that achieved the exact same outcome. Is one approach better than the others?

Activity: Plotting the normal density

Using the `normpdf` function you wrote above, write code to compute the normal density for 1,000 equally spaced numbers between between -3 and 3 (insert your code for the loop above the call to `plot`):

```
n <- 1000
x <- seq(-3, 3, length.out = n)
y <- rep(0, n)

plot(x, y, type = "l")
```

Footnote: Vectorization

A nice surprise is that you should be able perform the above computation *without a loop*. This is because R has an advanced feature called *vectorization*—many operations in R, including most basic mathematical operations, are automatically applied to all values in a vector. To check whether R automatically vectorized your `normpdf` function, try running this code:

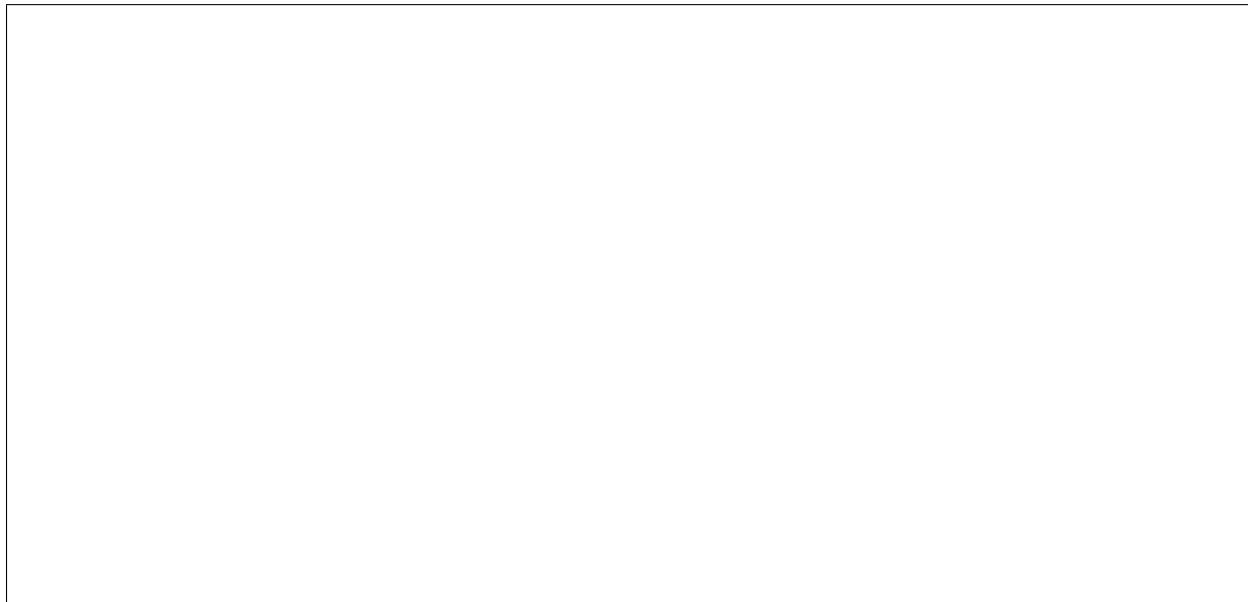
```
n <- 1000
x <- seq(-3, 3, length.out = n)
y <- normpdf(x, 1)
plot(x, y, type = "l")
```

Vectorization is very powerful, but it may take time to get comfortable with it, and know when it works well.

Group activity: The quadratic formula

You may remember that there is a formula for solving for the unknown, x , in a quadratic equation, $ax^2 + bx + c = 0$. It is commonly called the *quadratic formula*. Write down the quadratic formula:

In this exercise, you will work with your team to write a function, `solvequad`, that takes three numbers as input (a , b and c) and returns the solution(s) x that are real (*i.e.*, not complex numbers). **Hint:** Recall that a quadratic equation may have more than one real solution—or it may have none! You will need to make use of the `if` and `else` keywords to handle all possible cases. Write the code for your `solvequad` function here:



After creating `solvequad`, check that it does the right thing by running these tests:

```
solvequad(4,4,1)    # Should return -1/2.  
solvequad(1,-1,-2)  # Should return 2 and -1.  
solvequad(1,1,1)    # Should return no solutions.
```

Come up with a few more test cases at www.wolframalpha.com.

Case study: Do shorter titles lead to more citations?

To keep learning about R, we study the following question:

Is the length of a paper title related to the number of citations?

This is what Letchford *et al* claimed ([doi:10.1098/rsos.150266](https://doi.org/10.1098/rsos.150266)). In 2015, they analyzed 140,000 papers, and they found that *shorter titles* were associated with a larger number of citations.

In the folder containing the Basic Computing 2 tutorial materials, you will find data on scientific articles published between 2004 and 2013 in three top disciplinary journals, *Nature Neuroscience*, *Nature Genetics* and *Ecology Letters*. These data are contained in three CSV files. We are going to use these data to explore this question.

Load data

Start by reading in the data:

```
data.file <- file.path("citations", "nature_neuroscience.csv")
papers     <- read.csv(data.file, stringsAsFactors = FALSE)
```

Next, take a peek at the data. How large is it?

```
nrow(papers)
ncol(papers)
```

Let's see the first few rows:

```
head(papers)
```

The goal is to test whether papers with longer titles accrue fewer (or perhaps more?) citations than those with shorter titles. The first step is to add another column to the data containing the length of the title for each paper:

```
papers$TitleLength <- nchar(papers>Title)
```

Basic statistics in R

In the original paper, Letchford *et al* used rank-correlation: rank all the papers according to their title length and the number of citations. If Kendall's τ ("rank correlation") is positive, then longer titles are associated with *more* citations; if τ is negative, longer titles are associated with *fewer* citations. In R, you can compute rank correlation using `cor`:

```
k <- cor(papers>TitleLength, papers$Cited.by, method = "kendall")
```

To perform a significance test for correlation, use `cor.test`:

```
k.test <- cor.test(papers>TitleLength, papers$Cited.by, method = "kendall")
```

Does the output of `cor.test` show that the correlation between the ranks is positive or negative? Is this positive or negative correlation significant? You should find that the correlation is opposite of the one reported by Letchford *et al*—longer titles are associated with **more** citations!

Now we are going to examine the data in a different way to test whether this result is robust.

Basic plotting in R

To plot title length vs. number of citations, we need to learn about plotting in R. To produce a simple scatterplot using the base plotting functions, simply run:

```
plot(papers>TitleLength, papers$Cited.by)
```

The problem with this very simple plot is that it is hard to detect any trend because a few papers have many more citations than the rest, obscuring the data at the bottom of the plot. This suggests that plotting the data on the *logarithmic scale* is a better approach:

```
plot(papers$TitleLength, log10(papers$Cited.by))
```

Question: This is a better plot, but there is one problem with it. What is the problem? How can we fix it? Write your code to fix the problem:

Again, it is hard to see any trend in here. Maybe we should plot the best fitting line and overlay it on top of the graph. To do so, we first need to learn about linear regressions in R.

Linear regression in R

R was born for statistics — the fact that it is very easy to fit a linear regression in is not surprising! To build a linear model comparing columns x and y in data frame dat, use `lm`, which is the “Swiss army knife” for linear regression in R:

```
model <- lm(y ~ x, dat) # y = a + b*x + error
```

Let’s perform a linear regression of the number of citations (on the log-scale) vs. title length. To do so, first create a new column in the data frame containing the counts on the log-scale:

```
papers$LogCits <- log10(papers$Cited.by + 1)
```

Now you can perform a linear regression:

```
model_citations <- lm(LogCits ~ TitleLength, papers)
model_citations # Gives best-fit line.
summary(model_citations) # Gives more info.
```

And we can easily add this best-fit line to our plot:

```
plot(papers$TitleLength, papers$LogCits)
abline(model_citations, col = "red", lty = "dotted")
```

Once we add the best-fit line, the positive trend is more clear.

One thing to consider is that this data set spans a decade. Naturally, older papers have had more time to accrue citations. In our models, we should control for this effect. But first, we should explore whether this is an important factor to consider.

First, let’s plot the distribution of the number of citations for all the papers. To produce an histogram in R, use `hist`:

```
hist(papers$LogCits)
```

You can control the number of histogram bins with the `n` argument:

```
hist(papers$LogCits, n = 50)
```

Alternatively, estimate the density using `density`, then plot it:

```
plot(density(papers$LogCits))
```

Next, compare the distributions for papers published in 2004, 2009 and 2013:

```
plot(density(papers$LogCits[papers$Year == 2004]), col = "black")
lines(density(papers$LogCits[papers$Year == 2009]), col = "blue")
lines(density(papers$LogCits[papers$Year == 2013]), col = "red")
```

More recent papers should have fewer citations. Does your plot support this hypothesis? You can account for this in your regression model by incorporating the year of publication into the linear regression:

```
model_citations_better <- lm(LogCits ~ TitleLength + Year, papers)
summary(model_citations_better)
```

Does the regression coefficient (slope) for year confirm that older papers have more citations?

This is an improvement, but might be even better to have a separate “baseline” for each year. This can be done by converting the “Year” column to a factor:

```
papers$Year           <- factor(papers$Year)
model_citations_better <- lm(LogCits ~ Year + TitleLength, papers)
summary(model_citations_better)
```

This model has a different baseline for each year, and then title length influences this baseline. In this new model, are longer titles still associated with more citations?

Computing *p*-values using randomization

Kendall’s τ takes as input two rankings, x and y , both of the same length, n . It calculates the number of “concordant pairs” (if $x_i > x_j$, then $y_i > y_j$) and the number of “discordant pairs”. The final value is

$$\tau = \frac{n_{\text{concordant}} - n_{\text{discordant}}}{\frac{n(n-1)}{2}}$$

If x and y are completely independent, we would expect τ to have a distribution centered at zero. The variance of the “null” distribution of τ depends on the data. It is typically approximated as

a normal distribution. If you want to have a stronger result that does not rely on a normality assumption, you can use randomizations to calculate a p -value. Simply, compute τ for the actual data, as well as for many “fake” datasets obtained by randomizing the data. Your p -value is then the proportion of τ values for the randomized sets that exceed the τ value for the actual data.

Here, we will try to implement this randomization to calculate a p -value for papers published in 2006, and then we will compare against the p -value obtained from running `cor.test`. To do this, we will use a for-loop for the randomization.

First, subset the data:

```
dat <- papers[papers$Year == 2006, ]
```

Compute τ from these data:

```
k <- cor(dat>TitleLength, dat$Cited.by, method = "kendall")
```

Now calculate τ in “fake” data sets by randomly scrambling the citation counts. Begin by doing this for one fake data set:

```
shuffled_citation_counts <- sample(dat$Cited.by)
k.fake <- cor(dat>TitleLength, shuffled_citation_counts, method = "kendall")
```

Is the value of τ closer to zero in this “shuffled” data set?

To get an accurate p -value, we should compute τ for a large number of shuffled data sets. Let’s try 1,000 of them. This and similar randomization techniques are known as “bootstrapping”.

```
nr      <- 1000      # Number of fake data sets.
k.fake <- rep(0, nr) # Storage all the "fake" taus.
```

Since this computation involves lots of repetition, a for-loop makes a lot of sense here. Write your for-loop here:

After running your code, you should have 1,000 correlations calculated from 1,000 fake data sets. You have just generated a “null” distribution for τ . What does this null distribution look like? Try plotting it:

```
hist(k.fake, n = 50)
```

What proportion of the fake data sets have a correlation that exceeds the correlation in the actual data? This is the *p*-value. Write code to compute the *p*-value:

How does your new *p*-value compare to the *p*-value computed by `cor.test`? Is it smaller or larger?

```
cor.test(dat>TitleLength, dat$Cited.by, method = "kendall")
```

Question: Did you get the same result as the instructor, or your neighbours? If not, why? How could you ensure that your result is more similar, or the same?

In summary, whenever possible, use randomizations rather than relying on classical tests. They are more difficult to implement, and more computationally expensive, but they allow you to avoid making assumptions about your data.

Repeating the analysis for each year

Up until this point, we have only analyzed the citation data for 2006. Does the result we obtained for 2006 hold up in other years? Let's explore this question—we will use a for-loop to repeat the analysis for 2004 to 2013. Let's be smart about designing our code and use a *function* to decompose the problem into parts. The code for the final analysis will look like this:

```
years <- 2004:2013
for (i in years){
  dat <- papers[papers$Year == i, ]
  out <- analyze_citations(dat)
  cat("year:", i, "tau:", out$k, "pvalue:", out$pvalue, "\n")
}
```

The missing piece is the R code for function `analyze_citations`. You can re-use your code above to write this function.



Activity: Organizing and running your code

Now we would like to be able to automate the analysis, such that we can repeat it for each journal. This is a good place to pause and introduce how to go about writing programs that are well-organized, easy to write, easy to debug, and easier to reuse.

1. Take the problem, and divide it into smaller tasks (these are the functions).
2. Write the code for each task (function) separately, and make sure it does what it is supposed to do.
3. Document the code so that you can later understand what you did, how you did it, and why.
4. Combine the functions into a master program.

For example, let's say we want to write a program that takes as input the name of files containing citation data. The program should first fit a linear regression model,

```
log(citations + 1) ~ as.factor(Year) + TitleLength
```

then output the coefficient associated with TitleLength, and its *p*-value.

We could split the program into the following tasks:

1. A function to load and prepare the data for a linear regression analysis.
2. A function to run the linear regression analysis.
3. A master code that puts it all together.

Let's begin with the master code—the bulk of the code is a for-loop that repeats the regression analysis for each journal:

```
files <- list.files("citations", full.names = TRUE)
for (i in files) {
  cat("Analyzing data from", i, "\n")
  papers <- load_citation_data(i)
  out    <- fit_citation_model(papers)
```

```

  cat("coefficient:", out$estimate, "p-value:", out$pvalue, "\n")
}

```

This code doesn't work yet because you haven't written the functions that are called inside the loop. (What error message do you get when you try to run the code?)

The `load_citation_data` function reads in the data from the CSV file, then prepares the data for the linear regression analysis:

```

load_citation_data <- function (filename) {
  dat <- read.csv(filename, stringsAsFactors = FALSE)
  dat>TitleLength <- nchar(dat>Title)
  dat$LogCits     <- log10(dat$Cited.by + 1)
  dat$Year        <- as.factor(dat$Year)
  return(dat)
}

```

Before continuing, check that it works by running it on one of the CSV files:

```

papers <- load_citation_data("citations/nature_neuroscience.csv")

```

The `fit_citation_model` function fits a linear regression model to the input data, then extracts the quantities from the regression analysis we are most interested in (the “best-fit” slope and the *p*-value corresponding to “TitleLength”).

```

fit_citation_model <- function (papers) {
  model <- lm(LogCits ~ Year + TitleLength, papers)
  terms <- summary(model)$coefficients
  return(list(estimate = terms["TitleLength", "Estimate"],
             pvalue   = terms["TitleLength", "Pr(>|t|)"]))
}

```

Check that this function runs, and does what it is supposed to do:

```

out <- fit_citation_model(papers)

```

Now that you have defined the necessary functions, try running the master code above.

Question: Suppose you download a fourth CSV file containing data on papers from the *American Journal of Human Genetics*. Would any changes need to be made to your R code above to run it on the four citation data sets?

Creating computational notebooks using R Markdown

Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.

Donald E. Knuth, *Literate Programming*, 1984

When doing experiments, it is important to develop the habit of writing down everything you do in a laboratory notebook. That way, when writing your manuscript, responding to queries or discussing progress with your advisor, you can go back to your notes to find exactly what you did, how you did it, and possibly *why* you did it. The same should be true for computational work.

RStudio makes it very easy to build a computational laboratory notebook. First, create a new R Markdown file. (Choose **File > New File > R Markdown** from the RStudio menu bar.)

An R Markdown file is simply a text file. But it is interpreted in a special way that allows the text to be transformed it into a webpage (.html) or PDF file. You can use special syntax to render the text in different ways. Here are a few examples of R Markdown syntax:

```
*Italic text* **Bold text**
```

```
# Very large header
```

```
## Large header
```

```
### Smaller header
```

Unordered and ordered lists:

```
+ First  
+ Second  
    + Second 1  
    + Second 2
```

```
1. This is a  
2. Numbered list
```

When rendered as a PDF, the above R Markdown looks like this:

Very large header

Large header

Smaller header

Italic text **Bold text**

Unordered and ordered lists:

- First
- Second
 - Second 1
 - Second 2

1. This is a
2. Numbered list

You can also insert inline code by enclosing it in backticks.

The most important feature of R Markdown is that you can include blocks of code, and they will be interpreted and executed by R. You can therefore combine effectively the code itself with the description of what you are doing.

For example, including a code chunk in your R Markdown file,

```
```{r hello-world}
cat("Hello world!")
```
```

will render a document containing both the results and the code that run to generate those results:

```
cat("Hello world!")
# Hello world!
```

If you don't want to run the R code, but just display it, use `{r hello-world, eval = FALSE}`; if you want to show the output but not the code, use `{r hello-world, echo = FALSE}`.

You can include plots, tables, and even mathematical equations using LaTeX. In summary, when exploring your data, or describing the methods for your paper, give R Markdown a try!

You can find inspiration in this Boot Camp; the materials for Basic and Advanced Computing were written in R Markdown.

Programming challenge

Instructions

You will work with your group to solve the exercises below. When you have found the solutions, go to <https://stefanoallesina.github.io/BSD-QBio4> and follow the link "Submit solution to challenge 2" to submit your answer (alternatively, you can go directly to goo.gl/forms/QJhKmdGqRCIuGNPa2). At the end of the bootcamp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Google Flu Trends

Google Flu started strong, with a paper in *Nature* (Ginsberg *et al*, 2009, [doi:10.1038/nature07634](https://doi.org/10.1038/nature07634)) showing that, using data on Web search engine queries, one could predict the number of physician visits for influenza-like symptoms. Over time, the quality of predictions degraded considerably, requiring many adjustments to the model. Now defunct, Google Flu Trends has been proposed as a poster child of "Big Data hubris" (Lanzer *et al*, *Science*, 2014, [doi:10.1126/science.1248506](https://doi.org/10.1126/science.1248506)). In the folder containing the Basic Computing 2 tutorial materials, you will find the data used by Preis and Moat in their 2014 paper ([doi:10.1098/rsos.140095](https://doi.org/10.1098/rsos.140095)) to show that, after accounting for some additional historical data, Google Flu Trends are correlated with outpatient visits due to influenza-like illnesses.

1. Read the data using function `read.csv`, and plot the number of weekly outpatient visits versus the Google Flu Trends estimates.
2. Calculate the (Pearson's) correlation using the `cor` function.
3. The data span 2010–2013. In August 2013, Google Flu changed their algorithm. Did this lead to improvements? Compare the data from August and September 2013 with the same months in 2010, 2011 and 2012. For each, calculate the correlation, and see whether the correlation is higher for 2013.

Hint: You will need to extract the year from a string for each row. This can be done using `substr(gf$WeekCommencing, 1, 4)`, in which `gf` is the data frame containing the Google Flu data.

Advanced Computing 1 – Data wrangling and plotting*

Stefano Allesina *University of Chicago*

- **Goal:** learn how to manipulate large data sets by writing efficient, consistent, and compact code. Introduce the use of `dplyr`, `tidyR`, and the pipeline operator `%>%`. Produce beautiful graphs and figures for scientific publications using `ggplot2`.
 - **Audience:** experienced R users, familiar with the data type `data.frame`, loops, functions, and having some notions of data bases.
 - **Installation:** the following packages need to be installed: `ggplot2`, `dplyr`, `tidyR`, `lubridate`, `ggthemes`
-

Data wrangling

As biologists living in the XXI century, we are often faced with tons of data, possibly replicated over several organisms, treatments, or locations. We would like to streamline and automate our analysis as much as possible, writing scripts that are easy to read, fast to run, and easy to debug. Base R can get the job done, but often the code contains complicated operations (think of the cases in which you used `lapply` only because of its speed), and a lot of \$ signs and brackets.

To start, we need to import `tidyverse`:

```
library(tidyverse)
```

`tidyverse` is a fantastic bundle of packages: a collection of R packages designed to manipulate large data frames in a simple and straightforward way. These tools are also much faster than the corresponding base R commands, and allow you to write compact code by concatenating commands to build “pipelines”. Moreover, all of the packages in the bundle share the same philosophy, and are seamlessly integrated. By default, calling `library(tidyverse)` loads the packages `readr`, `tidyR` and `dplyr` (to read, organize and manipulate data), `ggplot2` (data plotting), `stringr` (string manipulation) and a few others; many others ancillary packages that are part of the `tidyverse` can be loaded if needed.

Then, we need a dataset to play with. We take a dataset containing all the papers published by UofC researchers in *Nature* or *Science* between 1999 and July 2019:

```
pubs <- read.csv("../data/UC_Nat_Sci_1999-2019.csv")
```

A new data type, `tibble`

The data are stored in a `data.frame`:

*This document is included as part of the Advanced Computing I packet for the U Chicago BSD qBio5 boot camp, @MBL, 2019. Current version: July 31, 2019; Corresponding author: sallesina@uchicago.edu.

```
is.data.frame(pubs)
```

tidyverse ships with a new data type, called a `tibble`. It also comes with its improved function to read data:

```
pubs <- read_csv("../data/UC_Nat_Sci_1999-2019.csv")
pubs
```

which automatically reads the data as a `tibble`. The nice feature of `tibble` objects is that they will print only what fits on the screen, and also give you useful information on the size of the data, as well as the type of data in each column. Other than that, a `tibble` object behaves very much like a `data.frame`. If you want to transform the `tibble` back into a `data.frame`, use the function `as.data.frame(my_tibble)`; the function `as_tibble(my_data_frame)` transforms a `data.frame` into a `tibble`.

We can take a look at the data using one of several functions:

- `head(pubs)` shows the first few rows
- `tail(pubs)` shows the last few rows
- `glimpse(pubs)` a summary of the data (similar to `str` in base R)
- `View(pubs)` open data in spreadsheet-like window

Selecting rows and columns

There are many ways to subset the data, either by row (subsetting the *observations*), or by column (subsetting the *variables*). For example, let's select only articles published after 2009:

```
filter(pubs, Year > 2009)
```

You can see that 515 of the 953 documents were published in the last 10 years. We have used the command `filter(tbl, conditions)` to select certain observations. We can combine several conditions, by listing them side by side, possibly using logical operators.

Exercise: what does this do?

```
filter(pubs, Year == 2008, `Source title` == "Nature", `Cited by` > 100)
```

Note that the “back ticks” can be used to type column names that contain spaces and non-standard characters. This is nice, because otherwise the name of the column would need to be altered (as done automatically by `read.csv`, sometimes creating column names that are difficult to interpret or type).

We can also select particular variables using the function `select(tbl, cols to select)`. For example, select only Authors and Title:

```
select(pubs, Authors, Title)
```

How many years are represented in the data set? We can use the function `distinct(tbl)` to retain only the rows that differ from each other:

```
distinct(select(pubs, Year))
```

Where we first extracted only the column `Year`, and then retained only distinct values.

Other ways to subset observations:

- `sample_n(tbl, howmany, replace = TRUE)` sample `howmany` rows at random with replacement
- `sample_frac(tbl, proportion, replace = FALSE)` sample a certain proportion (e.g. 0.2 for 20%) of rows at random without replacement
- `slice(tbl, 50:100)` extract the rows between 50 and 100
- `top_n(tbl, 10, Year)` extract the first 10 rows, once ordered by `Year`

More ways to select columns:

- `select(pubs, contains("Cited"))` select all columns containing the word `Cited`
- `select(pubs, -Authors, -Year)` exclude the columns `Authors` and `Year`
- `select(pubs, matches("astring|anotherstring"))` select all columns whose names match a regular expression.

Creating pipelines using `%>%`

We've been calling nested functions, such as `distinct(select(pubs, ...))`. If you have to add another layer or two, the code would become unreadable. `dplyr` allows you to "un-nest" these functions and create a "pipeline", in which you concatenate commands separated by the special operator `%>%`. For example:

```
pubs %>% # take a data table
  select(Year) %>% # select a column
  distinct() # remove duplicates
```

does exactly the same as the command we've run above, but is much more readable. By concatenating many commands, you can create incredibly complex pipelines while retaining readability.

Producing summaries

Sometimes we need to calculate statistics on certain columns. For example, calculate the average number of citations. We can do this using `summarise`:

```
pubs %>% summarise(avg = mean(`Cited by`))
```

which returns a `tibble` object with just the average number of citations. You can combine multiple statistics (use `first`, `last`, `min`, `max`, `n` [count the number of rows], `n_distinct` [count the number of distinct rows], `mean`, `median`, `var`, `sd`, etc.):

```
pubs %>% summarise(avg = mean(`Cited by`),  
                     sd = sd(`Cited by`),  
                     median = median(`Cited by`))
```

Summaries by group

One of the most useful features of `dplyr` is the ability to produce statistics for the data once subsetted by *groups*. For example, we would like to compute the average number of citations by journal and year:

```
pubs %>%  
  group_by(`Source title`, Year) %>%  
  summarise(avg = mean(`Cited by`))
```

Exercise: count the number of articles by UofC researcher in *Nature* and *Science* by Source title and Year.

Ordering the data

To order the data according to one or more variables, use `arrange()`:

```
pubs %>% select(Title, `Cited by`) %>% arrange(`Cited by`)  
pubs %>% select(Title, `Cited by`) %>% arrange(desc(`Cited by`))
```

Renaming columns

To rename one or more columns, use `rename()`:

```
pubs %>% rename(Cites = `Cited by`)
```

If you want to retain the new name(s), simply overwrite the object:

```
pubs <- pubs %>% rename(Cites = `Cited by`, Journal = `Source title`)
```

Adding new variables using `mutate`

If you want to add one or more new columns, use the function `mutate`. For example, suppose we want to count the number of authors for each document. Authors are separated by commas (with small errors, but let's disregard that), and therefore a strategy would be to first count the number of commas, and then add 1:

```
pubs <- pubs %>% mutate(Num_authors = str_count(Authors, ",") + 1)
```

use the function `transmute()` to create a new column and drop the original columns. You can also use `mutate` and `transmute` on grouped data.

When writing code, it is good practice to separate the operations by line:

```
# A more complex example: for each paper,  
# compute the percentile rank of citations  
# compared to other papers of the same year  
pubs %>%  
  group_by(Year) %>% # group papers according to year  
  mutate(pr = percent_rank(Cites)) %>% # compute % rank by Citations  
  ungroup() %>% # remove group information  
  arrange(Year, desc(pr), Authors) %>% # order by Year then % rank (decreasing)  
  head(20) # display first 20 rows
```

in this way, you can easily comment out a part of the pipeline (or add another piece in the middle).

Data plotting

The most salient feature of scientific graphs should be clarity. Each figure should make crystal-clear a) what is being plotted; b) what are the axes; c) what do colors, shapes, and sizes represent; d) the message the figure wants to convey. Each figure is accompanied by a (sometimes long) caption, where the details can be explained further, but the main message should be clear from glancing at the figure (often, figures are the first thing editors and referees look at).

Many scientific publications contain very poor graphics: labels are missing, scales are unintelligible, there is no explanation of some graphical elements. Moreover, some color graphs are impossible to understand if printed in black and white, or difficult to discern for color-blind people (8% of men, 0.5% of women).

Given the effort that you put in your science, you want to ensure that it is well presented and accessible. The investment to master some plotting software will be rewarded by pleasing graphics that convey a clear message.

In this section, we introduce `ggplot2`, a plotting package for R. This package was developed by Hadley Wickham who contributed many important packages to R (including `dplyr`), and who is the force behind `tidyverse`. Unlike many other plotting systems, `ggplot2` is deeply rooted in a “philosophical” vision. The goal is to conceive a grammar for all graphical representation of data. Leland Wilkinson and collaborators proposed The Grammar of Graphics. It follows the idea of a well-formed sentence that is composed of a subject, a predicate, and an object. The Grammar of Graphics likewise aims at describing a well-formed graph by a grammar that captures a very wide range of statistical and scientific graphics. This might be more clear with an example – Take a simple two-dimensional scatterplot. How can we describe it? We have:

- **Data** The data we want to plot.
- **Mapping** What part of the data is associated with a particular visual feature? For example: Which column is associated with the x-axis? Which with the y-axis? Which column corre-

sponds to the shape or the color of the points? In `ggplot2` lingo, these are called *aesthetic mappings* (`aes`).

- **Geometry** Do we want to draw points? Lines? In `ggplot2` we speak of *geometries* (`geom`).
- **Scale** Do we want the sizes and shapes of the points to scale according to some value? Linearly? Logarithmically? Which palette of colors do we want to use?
- **Coordinate** We need to choose a coordinate system (e.g., Cartesian, polar).
- **Faceting** Do we want to produce different panels, partitioning the data according to one (or more) of the variables?

This basic grammar can be extended by adding statistical transformations of the data (e.g., regression, smoothing), multiple layers, adjustment of position (e.g., stack bars instead of plotting them side-by-side), annotations, and so on.

Exactly like in the grammar of a natural language, we can easily change the meaning of a “sentence” by adding or removing parts. Also, it is very easy to completely change the type of geometry if we are moving from say a histogram to a boxplot or a violin plot, as these types of plots are meant to describe one-dimensional distributions. Similarly, we can go from points to lines, changing one “word” in our code. Finally, the look and feel of the graphs is controlled by a theming system, separating the content from the presentation.

Basic `ggplot2`

`ggplot2` ships with a simplified graphing function, called `qplot`. In this introduction we are not going to use it, and we concentrate instead on the function `ggplot`, which gives you complete control over your plotting. First, we need to load the package (note that `ggplot2` is automatically loaded by `tidyverse`). While we are at it, let’s also load a package extending its theming system:

```
library(ggplot2)
library(ggthemes)
```

A particularity of `ggplot2` is that it accepts exclusively data organized in tables (a `data.frame` or a `tibble` object). Thus, all of your data needs to be converted into a table format for plotting.

For our first plot, we’re going to produce a barplot showing the number of papers in `Science` and `Nature` by UoC researcher for each Year. To start:

```
ggplot(data = pubs)
```

As you can see, nothing is drawn: we need to specify what we would like to associate to the `x` axis (i.e., we want to set the *aesthetic mappings*):

```
ggplot(data = pubs) + aes(x = Year)
```

Note that we concatenate pieces of our “sentence” using the `+` sign! We’ve got the axes, but still no graph... we need to specify a geometry. Let’s use `barplot`:

```
ggplot(data = pubs) + aes(x = Year) + geom_bar()
```

As you can see, we wrote a well-formed sentence, composed of **data** + **mapping** + **geometry**, and this has produced a well-formed plot. We can add other mappings, for example, showing the journal in which the paper was published:

```
ggplot(data = pubs) + aes(x = Year, fill = Journal) + geom_bar()
```

Scatterplots

Using ggplot2, one can produce very many types of graphs. The package works very well for 2D graphs (or 3D rendered in two dimensions), while it lacks capabilities to draw proper 3D graphs, or networks.

The main feature of ggplot2 is that you can tinker with your graph fairly easily, and with a common grammar. You don't have to settle on a certain presentation of the data until you're ready, and it is very easy to switch from one type of graph to another.

For example, let's plot the number of citations in the y axis, the year in the x axis. We want a scatterplot, which is produced by the geometry `geom_point`:

```
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = Cites) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

This does not look very good, because some papers have a much larger number of citations than others. We can attempt plotting the $\log(Cites + 1)$ instead (the +1 is added because some papers might have 0 citations):

```
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = log(Cites + 1)) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

Much nicer! Now we can add a smoother by typing:

```
pl + geom_smooth() # spline by default
pl + geom_smooth(method = "lm", se = FALSE) # linear model, no standard errors
```

Exercise: repeat the plot of the citations, but showing a different colour for each journal; add a smoother for each journal separately. Do papers receive more citations when they're published in *Nature* or *Science*?

Histograms, density and boxplots

What is the distribution of citations?

```
ggplot(data = pubs) + aes(x = Cites) + geom_histogram()
```

You can see that there are some papers with many more citations than others. Try log-transforming the data:

```
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_histogram()
```

Now we observe an histogram much closer to a Normal distribution, meaning that the number of citations is approximately log-normally distributed. You can switch to a density plot quite easily (just change the geometry!):

```
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_density()
```

Similarly, we can produce boxplots, for example showing the number of citations for papers in *Nature* and *Science* (log transformed):

```
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_boxplot()
```

It is very easy to change geometry, for example switching to a violin plot:

```
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_violin()
```

Exercise:

- Produce a boxplot showing the number of authors (in log) per year (use `factor(Year)` for the x axis). Is science becoming more collaborative?
- Now produce a scatterplot showing the same trend, and add a smoothing function.

Scales

We can use scales to determine how the aesthetic mappings are displayed. For example, we could set the x axis to be in logarithmic scale, or we can choose how the colors, shapes and sizes are used. `ggplot2` uses two types of scales: continuous scales are used for continuous variables (e.g., real numbers); discrete scales for variables that can only take a certain number of values (e.g., treatments, labels, factors, etc.).

For example, let's plot a histogram showing the number of authors per paper:

```
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() # no transformation  
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +  
  scale_x_continuous(trans = "log") # natural log
```

```

ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "log10") # base 10 log
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "sqrt", name = "Number of authors")
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() + scale_x_log10() # shorthand

```

We can use different color scales. For example:

```

pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = Num_authors, y = Cites, colour = factor(Year)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
pl + scale_colour_brewer()
pl + scale_colour_brewer(palette = "Spectral")
pl + scale_colour_brewer(palette = "Set1")
pl + scale_colour_brewer("year of publication", palette = "Paired")

```

Or use the number of authors a continuous variable:

```

pl <- ggplot(data = pubs) +
  aes(x = Year, y = log(Cites + 1), colour = log(Num_authors)) +
  geom_point()
pl + scale_colour_gradient()
pl + scale_colour_gradient(low = "red", high = "green")
pl + scale_colour_gradientn(colours = c("blue", "white", "red"))

```

Similarly, you can use scales to modify the display of the shapes of the points (`scale_shape_continuous`, `scale_shape_discrete`), their size (`scale_size_continuous`, `scale_size_discrete`), etc. To set values manually (useful typically for discrete scales of colors or shapes), use `scale_colour_manual`, `scale_shape_manual` etc.

Themes

Themes allow you to manipulate the look and feel of a graph with just one command. The package `ggthemes` extends the themes collection of `ggplot2` considerably. For example:

```

library(ggthemes)
pl + theme_bw() # white background
pl + theme_economist() # like in the magazine "The Economist"
pl + theme_wsj() # like "The Wall Street Journal"

```

Faceting

In many cases, we would like to produce a multi-panel graph, in which each panel shows the data for a certain combination of parameters. In `ggplot` this is called *faceting*: the command `facet_grid`

is used when you want to produce a grid of panels, in which all the panels in the same row (column) have axis-ranges in common; `facet_wrap` is used when the different panels do not have axis-ranges in common.

For example:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +  
  aes(x = log10(Cites + 1)) +  
  geom_histogram()  
show(pl)  
pl + facet_grid(~Year) # in the same row  
pl + facet_grid(Year~.) # col  
pl + facet_grid(Journal ~ Year) # two facet variables  
pl + facet_wrap(Journal ~ Year, scales = "free") # just wrap around
```

Setting features

Often, you want to simply set a feature (e.g., the color of the points, or their shape), rather than using it to display information (i.e., mapping some aesthetic). In such cases, simply declare the feature outside the `aes`:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +  
  aes(x = log10(Num_authors))  
pl + geom_histogram()  
pl + geom_histogram(colour = "red", fill = "lightblue")
```

Saving graphs

You can either save graphs as done normally in R:

```
# save to pdf format  
pdf("my_output.pdf", width = 6, height = 4)  
print(my_plot)  
dev.off()  
# save to svg format  
svg("my_output.svg", width = 6, height = 4)  
print(my_plot)  
dev.off()
```

or use the function `ggsave`

```
# save current graph  
ggsave("my_output.pdf")  
# save a graph stored in ggplot object  
ggsave(plot = my_plot, filename = "my_output.svg")
```

Multiple layers

Finally, you can overlay different data sets, using different geometries. For example, suppose that we have two data sets: one for papers with few authors (say <10) and one for large collaborations:

```
small_collab <- pubs %>% filter(Num_authors < 10)
large_collab <- pubs %>% filter(Num_authors >= 10)
```

We can overlay different geometries for the same data set:

```
ggplot(data = small_collab) +
  aes(x = factor(Num_authors), y = log(Cites + 1)) +
  geom_boxplot(fill = "lightblue") +
  geom_violin(fill = "NA") +
  geom_point(alpha = 0.25) # alpha stands for transparency
```

Or combine different data sets (with the same aes!):

```
ggplot(data = small_collab) +
  aes(x = Year) +
  geom_bar(fill = "red", alpha = 0.5) +
  geom_bar(data = large_collab, fill = "blue", alpha = 0.5)
```

Tidying up data

The best way to organize data for plotting and computing is the *tidy form*, meaning that a) each variable has its own column, and b) each observation has its own row. When data are not in tidy form, you can use the package `tidyverse` to reshape them.

For example, suppose we want to produce a table in which for each journal and year, we report the average number of authors. First, we need to compute the values:

```
avg_authors <- pubs %>%
  group_by(Journal, Year) %>%
  summarise(avg_au = mean(Num_authors))
```

This table is in tidy format (also called “narrow” format); we want to create columns for each journal, and report the average in the corresponding cell. To do so, we “spread” the journals into columns:

```
avg_authors <- avg_authors %>% spread(Journal, avg_au)
```

Note that this is not in tidy form, as two observations are in the same row (also called “messy” or “wide” format). While this is not ideal for computing, it is great for human consumption, as we can easily compare the two numbers in the same row.

If we want to go back to tidy form, we can “gather” the column names, and return to tidy:

```
# gather(where to store col names,
#        where to store values,
#        which columns to gather)
avg_authors %>% gather(Journal, Average_num_authors, 2:3)
# alternatively, if it's cleaner
avg_authors %>% gather(Journal, Average_num_authors, -Year)
```

Joining tables

If you have multiple data frames or tibble objects with shared columns, it is easy to join them (as in a database). To showcase this, we are going to extract papers by very prolific authors. First, we want to compute how many papers are in the data for each “author” (actually, last-name initial combinations, which might represent different authors with common names...). First, we need a data set in which the authors have been separated:

```
by_author <- pubs %>%
  select(Authors, Title) %>%
  separate_rows(sep = " ", , Authors) %>%
  rename(Focal_author = Authors)
```

Now we can count the number of appearances of each name:

```
by_author <- by_author %>%
  group_by(Focal_author) %>%
  mutate(Tot = n())
```

Where we have created a new column (Tot) by calling `mutate` on grouped data. Who are the authors most represented in the data?

```
tot_author <- by_author %>%
  select(Focal_author, Tot) %>%
  distinct() %>%
  arrange(desc(Tot))
```

You can see that common Chinese name combinations are in the top few rows (meaning that probably we conflated several authors...). Let’s plot an histogram:

```
tot_author %>% ggplot() + aes(x = Tot) + geom_histogram() + scale_y_log10()
```

As you can see, the vast majority of authors appears only once, and very few, appear 15 or more times. We want to extract the papers of the most prolific authors from the data that we have stored in `pubs`. For example, we want to consider authors that are represented 10 or more times in these papers. To do so, we first extract the prolific authors:

```
prolific <- by_author %>% filter(Tot >= 10)
```

and now we can join pubs and prolific. By calling `inner_join`, only rows that are present in both tables will be retained; because the two tables share a column (`Title`), `dplyr` can proceed automatically:

```
pubs %>% inner_join(prolific)
```

We can use this table to compute the number of citations received by each prolific author:

```
pubs %>% inner_join(prolific) %>% ggplot() +  
  aes(x = Focal_author, y = Cites) +  
  geom_col() # similar to bar plot  
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) # rotate labels
```

Besides `inner_join(x, y)`, you can use:

- `left_join(x, y)`: return all rows from `x`, and all columns from `x` and `y` (those with no match will show NA);
- `right_join(x, y)`: return all rows from `y`, and all columns from `x` and `y`;
- `full_join(x, y)`: return all rows and all columns from both `x` and `y`. Where there are not matching values, returns NA for the one missing;
- `anti_join(x, y)`: return all rows from `x` where there are not matching values in `y`.

Exercises in groups

The file `data/Chicago_Crimes_May2016.csv` contains a list of all the crimes reported in Chicago in May 2016. Form small groups and work on the following exercises:

- **Crime map** write a function that takes as input a crime's Primary Type (e.g., ASSAULT), and draws a map of all the occurrences. Mark a point for each occurrence using Latitude and Longitude. Set the alpha to something like 0.1 to show brighter colors in areas with many occurrences.
- **Crimes by community** write a function that takes as input a crime's Primary Type, and produces a barplot showing the number of crimes per Community area. The names of the community areas are found in the file `data/Chicago_Crimes_CommunityAreas.csv`. You will need to join the tables before plotting.
- **Violent crimes** add a new column to the dataset specifying whether the crime is considered violent (e.g., HOMICIDE, ASSAULT, KIDNAPPING, BATTERY, CRIM SEXUAL ASSAULT, etc.)
- **Crimes in time** plot the number of violent crimes against time, facetting by community areas. (Hint: use the package `lubridate` to work with days, dates, time)
- **Dangerous day** which day of the week is the most dangerous? (Extract day of the week from Date)
- **Dangerous time** which time of the day is the most dangerous (Extract the hour of the day from Date).

- **Correlation between crimes** [difficult] which crimes tend to have the same pattern? Divide the crimes by day and type, and plot the correlation between crimes using `geom_tile` and colouring the cells according to the correlation (see `cor` for a function that computes the correlation between different columns).

Advanced Computing 2 – UNIX and RegEx*

Stefano Allesina *University of Chicago*

- **Goal:** Learn to write pipelines for data manipulation and analysis using the UNIX shell. Show how to interface the UNIX shell and R. Introduce the use of regular expressions.
 - **Audience:** experienced R users, familiar with dplyr and ggplot2.
 - **Installation:** Windows users need to install a UNIX shell emulator (e.g., Git Bash); for regular expressions, we are going to use the R package stringr.
-

The UNIX Shell

UNIX is an operating system (i.e., the software that lets you interface with the computer) developed in the 1970s by a group of programmers at the AT&T Bell laboratories. Among them were Brian Kernighan and Dennis Ritchie, who also developed the programming language C. The new operating system was an immediate success in academic circles, with many scientists writing new programs to extend its features. This mix of commercial and academic interest led to the many variants of UNIX available today (e.g., OpenBSD, Sun Solaris, Apple OS X), collectively denoted as *nix systems. Linux is the open source UNIX clone whose “engine” (kernel) was written from scratch by Linus Torvalds with the assistance of a loosely-knit team of hackers from across the internet.

All *nix systems are multi-user, network-oriented, and store data as plain text files that can be exchanged between interconnected computer systems. Another characteristic is the use of a strictly hierarchical file system.

Why use UNIX?

Many biologists are not familiar with coding in *nix systems and, given that the learning curve is initially fairly steep, we start by listing the main advantages of these systems over possible alternatives.

First, UNIX is an operating system written by programmers for programmers. This means that it is an ideal environment for developing your code and storing your data.

Second, hundreds of small programs are available to perform simple tasks. These small programs can be strung together efficiently so that a single line of UNIX commands can perform complex operations, which otherwise would require writing a long and complex program. The possibility of creating these pipelines for data analysis is especially important for biologists, as modern research groups produce large and complex data sets, whose analysis requires a level of automation that would be hard to achieve otherwise. For instance, imagine working with millions of files by having to open each one of them manually to perform an identical task, or try opening your single 80Gb whole-genome sequencing file in a software with a graphical user interface! In UNIX, you can

*This document is included as part of the Advanced Computing II packet for the U Chicago BSD qBio5 boot camp, @MBL, 2019. Current version: July 28, 2019; Corresponding author: sallesina@uchicago.edu.

string a number of small programs together, each performing a simple task, and create a complex pipeline that can be stored in a script (a text file containing all the commands). Then, you can let the computer analyze all of your data while you’re having a cup of coffee.

Third, text is the rule: almost anything (including the screen, the mouse, etc.) in UNIX is represented as a text file. Using text files means that all of your data can be read and written by any machine, and without the need for sophisticated (and expensive) proprietary software. Text files are (and always will be) supported by any operating system and you will still be able to access your data decades from today (while this is not the case for most commercial software). The text-based nature of UNIX might seem unusual at first, especially if you are used to graphical interfaces and proprietary software. However, remember that UNIX has been around since the early 1970s and will likely be around at the end of your career. Thus, the hard work you are putting into learning UNIX will pay off over a lifetime.

The long history of UNIX means that a large body of tutorials and support web sites are readily available online. Last but not least, UNIX is very stable, robust, secure, and—in the case of Linux—freely available.

In the end, entirely avoiding to work in a UNIX “shell” is almost impossible for a professional scientist: basically all resources for High-Performance Computing (computer clusters, large workstations, etc.) run a UNIX or Linux operating system. Similarly, the transfer of large files, websites, and data between machines is typically accomplished through command-line interfaces.

Directory structure

In UNIX we speak of “directories”, while in a graphical environment the term “folder” is more common. These two terms are interchangeable and refer to a structure that may contain sub-directories and files. The UNIX directory structure is organized hierarchically in a tree. As a biologist, you can think of this structure as a phylogenetic tree. The common ancestor of all directories is also called the “root” directory and is denoted by an individual slash (/). From the root directory, several important directories branch:

- /bin contains several basic programs.
- /etc contains configuration files.
- /dev contains the files connecting to devices such as the keyboard, mouse and screen.
- /home contains the home directory of each user (e.g., /home/yourname; in OS X, your home directory is stored in /Users/yourname).
- /tmp contains temporary files.

You will typically work in your home directory. From there, you can access the Desktop, Downloads, Documents, and other directories you are likely familiar with. When you navigate the system you are in one directory and can move deeper in the tree or upward towards the root.

Using the terminal

Terminal refers to the interface that you use to communicate with the kernel (the core of the operating system). The terminal is also called *shell*, or *command-line interface* (CLI). It processes the commands you type, translates them for the kernel, and shows you the results of your operations. There are several shells available. Here, we concentrate on the most popular one, the bash shell, which is the default shell in both Ubuntu and OS X.

In Ubuntu, you can open a shell by pressing `Ctrl + Alt + t` or by opening the dash (press the Super key) and typing Terminal. In OS X, you want to open the application `Terminal.app`, which is located in the folder *Utilities* within *Applications*. Alternatively, you can type Terminal in *Spotlight*. In either system, the shell will automatically start in your home directory. Windows users can launch Git Bash or another terminal emulator.

The command line prompt ends with a “dollar” (\$) sign. This means the terminal is ready to accept your commands. Here, a \$ sign at the beginning of a line of code signals that the command has to be executed in your terminal. You do not need to type the \$ sign in your terminal, just copy the command that follows it.

In UNIX, you can use the Tab key to reduce the amount you have to type, which in turn reduces errors caused by typos. When you press Tab in a (properly configured) shell, it will try to automatically complete your command, directory or file name (if multiple completions are possible, you can display them all by hitting the Tab key twice). Additionally, you can navigate the history of commands you typed by using the up/down arrows (you do not need to re-type a command that you recently executed). There are also shortcuts that help when dealing with long lines of code:

- `Ctrl + A` Go to the beginning of the line
- `Ctrl + E` Go to the end of the line
- `Ctrl + L` Clear the screen
- `Ctrl + U` Clear the line before the cursor position
- `Ctrl + K` Clear the line after the cursor
- `Ctrl + C` Kill the command that is currently running
- `Ctrl + D` Exit the current shell
- `Alt + F` Move cursor forward one word (in OS X, `Esc + F`)
- `Alt + B` Move cursor backward one word (in OS X, `Esc + B`)

Mastering these and other keyboard shortcuts will save you a lot of time. You may want to print this list and keep it next to your keyboard—in a while you will have them all memorized and will start using them automatically.

Basic UNIX commands

Here we introduce some of the most basic (and most useful) UNIX commands. We write the commands in fixed-width font and specific, user-provided input is capitalized in square brackets. Again, the brackets and special formatting are not required to execute a command in your terminal.

Many commands require some arguments (e.g., copy which file to where), and all can be modified using the several options available. Typically, options are either written as a dash followed by a single letter (older style, e.g., `-f`) or two dashes followed by words (newer style, e.g., `--full-name`). A command, its options and arguments are separated by a space.

How to get help in UNIX

UNIX ships with hundreds of commands. As such, it is impossible to remember them all, let alone all their possible options. Fortunately, each command is described in detail in its manual page, which can be accessed directly from the shell by typing `man [COMMAND OF YOUR CHOICE]` (not available in Git Bash). Use arrows to scroll up and down and hit q to close the manual page.

Checking the exact behavior of a command is especially important, given that the shell will execute any command you type without asking whether you know what you're doing (such that it will promptly remove all of your files, if that's the command you typed). You may be used to more forgiving (and slightly patronizing) operating systems in which a pop-up window will warn you whenever something you're doing is considered dangerous. In UNIX, it is always better to consult the manual rather than improvising.

If you want to interrupt the execution of a command, press `Ctrl + C` to halt any command that is currently running in your shell.

Navigating the directory system

You can navigate the hierarchical UNIX directory system using these commands:

- `pwd` print the path of the current working directory.
- `ls` list the files and sub-directories in the current directory. `ls -a` list all (including hidden) files. `ls -l` return the long list with detailed information. `ls -lh` provide file sizes with units (B, M, K, etc.).)
- `cd [NAMEOFTDIR]` change directory. `cd ..` move one directory up; `cd /` move to the root directory; `cd ~` move to your home directory; `cd -` go back to the directory you visited previously (like "Back" in a browser).

Handling directories and files

Create and delete files or directories using the following commands:

- `cp [FROM] [TO]` copy a file. The first argument is the file to copy. The second argument is where to copy it (either a directory or a file name).
- `mv [FROM] [TO]` move or rename a file. Move a file by specifying two arguments: the file, and the destination directory. Rename a file by specifying the old and the new file name in the same directory.
- `touch [FILENAME]` Update the date of last access to the file. Interestingly, if the file does not exist, this command will create an empty file.
- `rm [TODEMOVE]` remove a file. `rm -r` deletes the contents of a directory recursively (i.e., including all files and sub-directories in it; use with caution!). Similarly, `rm -f` removes the file and suppresses any prompt asking whether you are sure you want to remove the file.
- `mkdir [DIRECTORY]` make a directory. To create nested directories, use the option `-p` (e.g., `mkdir -p d1/d2/d3`).
- `rmdir [DIRECTORY]` remove an empty directory.

Printing and modulating files

UNIX was especially designed to handle text files, which is apparent when considering the multitude of commands dealing with text. Here are a few popular ones:

- `less [FILENAME]` progressively print a file on the screen (press `q` to exit). Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, in UNIX, `less` is `more`.

- `cat [FILENAME]` concatenate and print files.
- `wc [FILENAME]` line, word, and byte (character) count of a file.
- `sort [FILENAME]` sort the lines of a file and print the result to the screen.
- `uniq [FILENAME]` show only unique lines of a file. The file needs to be sorted first for this to work properly.
- `file [FILENAME]` determine the type of a file.
- `head [FILENAME]` print the head (i.e., first few lines of a file).
- `tail [FILENAME]` print the tail (i.e., last few lines of a file).
- `diff [FILE1] [FILE2]` show the differences between two files.

Exercise To familiarize yourself with these commands, try the following:

- Go to the data directory for this tutorial.
- How many lines are in file `Marra2014_data.fasta`?
- Go back to the code directory.
- Create the empty file `toremove.txt`.
- List the content of the directory.
- Remove the file `toremove.txt`.

Miscellaneous commands

- `echo "[A STRING]"` print the string `[A STRING]`.
- `time` time the execution of a command.
- `wget [URL]` download the webpage at `[URL]`. (Available in Ubuntu; for OS X look at `curl`, or install `wget`).
- `history` list the last commands you executed.

Advanced UNIX commands

Redirection and pipes

So far, we have printed the output of each command (e.g., `ls`) directly to the screen. However, it is easy to direct the output to a file (*redirect*) or use it as the input of another command (*pipe*). Stringing commands together in pipes is the real power of UNIX—the ability to perform complex processing of large amounts of data in a single line of commands. First, we show how to redirect the output of a command into a file:

```
$ [COMMAND] > [FILENAME]
```

Note that if the file `[FILENAME]` exists, it will be overwritten. If instead we want to append to an existing file, we can use the `>>` symbol as in the following line:

```
$ [COMMAND] >> [FILENAME]
```

When the command is very long and complex, we might want to redirect the content of a file as input to a command, “reversing” the flow:

```
$ [COMMAND] < [FILENAME]
```

To run a few examples, let's start by moving to our code directory:

```
$ cd ~/BSD-QBio2/tutorials/advanced_computing2/code
```

The command `echo` can be used to print a string on the screen. Instead of printing to the screen, we redirect the output to a file, effectively creating a file containing the string we want to print:

```
$ echo "My first line" > test.txt
```

We can see the result of our operation by printing the file to the screen using the command `cat`:

```
$ cat test.txt
```

To append a second line to the file, we use `>>`:

```
$ echo "My second line" >> test.txt
$ cat test.txt
```

We can redirect the output of any command to a file. For example, it is quite common to have to determine how many files are in a directory. The files could have been created by an instrument or provided by a collaborator. Before analyzing the data, we want to get a sense of how many files will we need to process. If there are thousands of files, it is quite time consuming to count them by hand or even open a file browser that can do the counting for us. It is much simpler and faster to just type a command or two. To try this, let's create a file listing all the files contained in `data/Saavedra2013`:

```
$ ls ../data/Saavedra2013 >> filelist.txt
$ cat filelist.txt
```

Now we want to count how many lines are in the file. We can do so by calling the command `wc -l` (count only the lines):

```
$ wc -l filelist.txt
$ rm filelist.txt
```

However, we can skip the creation of the file by creating a short pipeline. The pipe symbol `|` tells the shell to take the output on the left of the pipe and use it as the input of the command on the right of the pipe. To take the output of the command `ls` and use it as the input of the command `wc` we can write:

```
$ ls ../data/Saavedra2013 | wc -l
```

We have created our first, simple pipeline. In the following sections, we are going to build increasingly long and complex pipelines. The idea is always to start with a command and progressively add one piece after another to the pipeline, each time checking that the result is the desired one.

Selecting columns using *cut*

When dealing with tabular data, you will often encounter the Comma Separated Values (CSV) Standard File Format. The CSV format is platform and software independent, making it the standard output format of many experimental devices. The versatility of the file format should also make it your preferred choice when manually entering and storing data.

The main UNIX command you want to master for comma-, space-, tab-, or character-delimited text files is *cut*. To showcase its features, we work with data on generation time of mammals published by Pacifici *et al.*. First, let's make sure we are in the right directory (`advanced_computing2/data`). Then, we can print the header (the first line, specifying the content of each column) of the CSV file using the command `head`, which prints the first few lines of a file on the screen, with the option `-n 1`, specifying that we want to output only the first line:

```
$ head -n 1 Pacifici2013_data.csv  
TaxID;Order;Family;Genus;Scientific_name;...
```

We now pipe the header to `cut`, specify the character to be used as delimiter (`-d ','`), and use the `head` command to extract the name of the first column (`-f 1`), or the names of the first four columns (`-f 1-4`):

```
$ head -n 1 Pacifici2013_data.csv | cut -d ',' -f 1  
TaxID  
  
$ head -n 1 Pacifici2013_data.csv | cut -d ',' -f 1-4  
TaxID;Order;Family;Genus
```

Remember to use the Tab key to auto-complete file names and the arrow keys to access your command history.

In the next example, we work with the file content. We specify a delimiter, extract specific columns, and pipe the result to the `head` command—to display only the first few elements:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | head -n 5  
Order  
Rodentia  
Rodentia  
Rodentia  
Macroscelidea  
  
$ cut -d ';' -f 2,8 Pacifici2013_data.csv | head -n 3  
Order;Max_longevity_d  
Rodentia;292  
Rodentia;456.25
```

Now, we specify the delimiter, extract the second column, skip the first line (the header) using the `tail -n +2` command (i.e., return the whole file starting from the second line), and finally display the first five entries:

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | tail -n +2 | head -n 5
Rodentia
Rodentia
Rodentia
Macroscelidea
Rodentia
```

We pipe the result of the previous command to the `sort` command (which sorts the lines), and then again to `uniq`, (which takes only the elements that are not repeated). Effectively, we have created a pipeline to extract the names of all the Orders in the database, from Afrosoricida to Tubulidentata (a remarkable Order, which today contains only the aardvark).

```
$ cut -d ';' -f 2 Pacifici2013_data.csv | tail -n +2 | sort | uniq
Afrosoricida
Carnivora
Cetartiodactyla
...
```

This type of manipulation of character-delimited files is very fast and effective. It is an excellent idea to master the `cut` command in order to start exploring large data sets without the need to open files in specialized programs (if you don't want to modify the content of a file, you should not open it in an editor!).

Exercise:

- If we order all species names (fifth column) of `Pacifici2013_data.csv` in alphabetical order, which is the first species? Which the last?
- How many families are represented in the database?

Substituting characters using `tr`

We often want to substitute or remove a specific character in a text file (e.g., to convert a comma-separated file into a tab-separated file). Such a one-by-one substitution can be accomplished with the command `tr`. Let's look at some examples in which we use a pipe to pass a string to `tr`, which then processes the text input according to the search term and specific options.

Substitute all characters a with b:

```
$ echo 'aaaabbbb' | tr 'a' 'b'
bbbbbbb
```

Substitute every number in the range 1 through 5 with 0:

```
$ echo '123456789' | tr 1-5 0
000006789
```

Substitute lower-case letters with upper-case (note the one-to-one mapping):

```
$ echo 'ACtGGcAaTT' | tr actg ACTG  
ACTGGCAATT
```

We achieve the same result using bracket expressions that provide a predefined set of characters. Here, we use the set of all lower-case letters [:lower:] and translate into upper-case letters [:upper:]:

```
$ echo 'ACtGGcAaTT' | tr [:lower:] [:upper:]  
ACTGGCAATT
```

We can also indicate ranges of characters to substitute:

```
$ echo 'aabbcdddee' | tr a-c 1-3  
112233ddee
```

Delete all occurrences of a:

```
$ echo 'aaaaabbbb' | tr -d a  
bbbb
```

“Squeeze” all consecutive occurrences of a:

```
$ echo 'aaaaabbbb' | tr -s a  
abbbb
```

Note that the command `tr` reads standard input, and does not operate on files directly. However, we can use pipes in conjunction with `cat`, `head`, `cut`, etc. to create input for `tr`:

```
$ tr ' ' '\t' < [INPUTFILE] > [OUTPUTFILE]
```

In this example we input [INPUTFILE] to the `tr` command to replace all spaces with tabs. Note the use of quotes to specify the space character. The tab is indicated by `\t` and is called a “meta-character”. We use the backslash to signal that the following character should not be interpreted literally, but rather is a special code referring to a character that is difficult to represent otherwise.

Now we can apply the command `tr` and the commands we have showcased earlier to create a new file containing a subset of the data contained in `Pacifici2013_data.csv`, which we are going to use in the next section.

First, we change directory to the code:

```
$ cd ../code/
```

Now, we want to create a version of `Pacifici2013_data.csv` containing only the `Order`, `Family`, `Genus`, `Scientific_name`, and `AdultBodyMass_g` (columns 2-6). Moreover, we want to remove the header, sort the lines according to body mass (with larger critters first), and have the values separated by spaces. This sounds like an awful lot of work, but we’re going to see how this can be accomplished piping a few commands together.

First, let’s remove the header:

```
$ tail -n +2 ../data/Pacifici2013_data.csv
```

Then, take only the columns 2-6:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6
```

Now, substitute the current delimiter (;) with a space:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' ''
```

To sort the lines according to body size, we need to exploit a few of the options for the command `sort`. First, we want to sort numbers (option `-n`); second, we want larger values first (option `-r`, reverse order); finally, we want to sort the data according to the sixth column (option `-k 6`):

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' '' | sort -n -r -k 6
```

That's it. We have created our first complex pipeline. To complete the task, we redirect the output of our pipeline to a new file called `BodyM.csv`.

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ';' -f 2-6 | tr -s ';' '' | sort -r -n -k 6 > BodyM.csv
```

You might object that the same operations could have been accomplished with a few clicks by opening the file in a spreadsheet editor. However, suppose you have to repeat this task many times, e.g., to reformat every file that is produced by a laboratory device. Then it is convenient to automate this task such that it can be run with a single command.

Similarly, suppose you need to download a large CSV file from a server, but many of the columns are not needed. With `cut`, you can extract only the relevant columns, reducing download time and storage.

Selecting lines using grep

`grep` is a powerful command that finds all the lines of a file that match a given pattern. You can return or count all occurrences of the pattern in a large text file without ever opening it. `grep` is based on the concept of regular expressions, which we will cover just below (but using R, in which the syntax is slightly different).

We will test the basic features of `grep` using the file we just created. The file contains data on thousands of species:

```
$ wc -l BodyM.csv  
5426 BodyM.csv
```

Let's see how many wombats (family Vombatidae) are contained in the data. First we display the lines that contain the term "Vombatidae":

```
$ grep Vombatidae BodyM.csv
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus krefftii 31849.99
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus latifrons 26163.8
Diprotodontia Vombatidae Vombatus Vombatus ursinus 26000
```

Now we add the option -c to count the lines:

```
$ grep -c Vombatidae BodyM.csv
3
```

Next, we have a look at the genus Bos in the data file:

```
$ grep Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
Cetartiodactyla Bovidae Boselaphus Boselaphus tragocamelus 182253
```

Besides all the members of the Bos genus, we also match one member of the genus Boselaphus. To exclude it, we can use the option -w, which prompts grep to match only full words:

```
$ grep -w Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
```

Using the option -i we can make the search case-insensitive (it will match both upper- and lower-case instances):

```
$ grep -i Bos BodyM.csv
Proboscidea Elephantidae Loxodonta Loxodonta africana 3824540
Proboscidea Elephantidae Elephas Elephas maximus 3269794
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
...
```

Sometimes, we want to know which lines precede or follow the one we want to match. For example, suppose we want to know which mammals have body weight most similar to the gorilla (*Gorilla gorilla*). The species are already ordered by size, thus we can simply print the two lines before the match using the option -B 2 and the two lines after the match using -A 2:

```
$ grep -B 2 -A 2 "Gorilla gorilla" BodyM.csv
Cetartiodactyla Bovidae Ovis Ovis ammon 113998.7
Cetartiodactyla Delphinidae Lissodelphis Lissodelphis borealis 113000
```

```
Primates Hominidae Gorilla Gorilla gorilla 112589
Cetartiodactyla Cervidae Blastocerus Blastocerus dichotomus 112518.5
Cetartiodactyla Iniidae Lipotes Lipotes vexillifer 112138.3
```

Use option `-n` to show the line number of the match. For example, the gorilla is the 164th largest mammal in the database:

```
$ grep -n "Gorilla gorilla" BodyM.csv
164:Primates Hominidae Gorilla Gorilla gorilla 112589
```

To print all the lines that do not match a given pattern, use the option `-v`. For example, to get the other species of the genus *Gorilla* with the exception of *Gorilla gorilla*, we can use:

```
$ grep Gorilla BodyM.csv | grep -v gorilla
Primates Hominidae Gorilla Gorilla beringei 149325.2
```

To match one of several strings, use `grep "[STRING1]||[STRING2]"`

```
$ grep -w "Gorilla\|Pan" BodyM.csv
Primates Hominidae Gorilla Gorilla beringei 149325.2
Primates Hominidae Gorilla Gorilla gorilla 112589
Primates Hominidae Pan Pan troglodytes 45000
Primates Hominidae Pan Pan paniscus 35119.95
```

You can use grep on multiple files at a time! Simply, list all the files to use instead of just one file.

Interfacing R and the UNIX shell

Note: this will work if you're using Mac OSX or UNIX; in Windows options are much more limited.

Calling R from the command line

In R, you typically work in “interactive” mode — you type a command, it gets executed, you type another command, and so on. Often, we want to be able to re-run a script on different data sets or with different parameters. For that purpose you can store all the commands in a text file (typically, with extension `.R`), and then re-run the analysis by typing in the UNIX command line

```
$ Rscript my_script_file.R
```

To properly automate our analysis and figure generation, however, we can additionally pass command-line arguments to R. This allows us for instance to perform the analysis using a specific input file, or save the figure using a specific file name.

Rscript accepts command-line arguments, that need to be parsed within R. The code at the beginning of the following script shows how this is accomplished:

```

# Get all the command-line arguments
args <- commandArgs(TRUE)
# Assign each argument to a variable,
# making sure to convert it to the right
# type of variable (string by default)

# check the number of arguments
num.args <- length(args)
print(paste("Number of command-line arguments:", num.args))
# print all the arguments
if (num.args > 0) {
  for (i in 1:num.args) {
    print(paste(i, "->", args[i]))
  }
}

# We can initially set to default values
# (but pay attention to the order,
# the optional arguments should be at the end)
input.file <- "test.txt"
number.replicates <- 10
starting.point <- 3.14

if (num.args >= 1) {
  input.file <- args[1]
}
if (num.args >= 2) {
  number.replicates <- as.integer(args[2])
}
if (num.args >= 3) {
  starting.point <- as.double(args[3])
}

print(c(input.file, number.replicates, starting.point))

# Save this script as my_script.R
# Run the script in bash with different arguments
# Rscript my_script.R abc.txt 5 100.0
# Rscript my_script.R abc.txt 5
# Rscript my_script.R abc.txt
# Rscript my_script.R

```

Calling the command line from R

You can call the operating system from within R (assuming you're in /advanced_computing_2/code):

```
system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno")
```

You can also capture the output from the shell commands and save it into R. Everything is treated as text (convert to numeric if necessary):

```
numlines <- system("wc -l < ../../basic_computing_1/data/H938_Euro_chr6.geno",
  intern = TRUE)
```

```
## Warning in system("wc -l < ../../basic_computing_1/data/
## H938_Euro_chr6.geno", : running command 'wc -l < ../../basic_computing_1/
## data/H938_Euro_chr6.geno' had status 2
```

```
numlines
```

```
## character(0)
## attr(,"status")
## [1] 2
```

You can also use a combination of shell commands and `read.table` to capture more complex output:

```
mydf <- system("grep rs125283 ../../basic_computing_1/data/H938_Euro_chr6.geno",
  intern = TRUE)
mydf <- read.table(file = textConnection(mydf))
mydf
```

Regular expressions in R

Sometimes data is hidden in free text. Think of citations in a manuscript, mentions of DNA motifs in tables, etc. You could copy and paste data from these unstructured texts yourself, but if you have much text, the task is very boring and error-prone. What you need is a way to describe a text pattern to a computer, and then have it extract the data automatically. Regular Expressions do exactly that.

Because you want to describe a text pattern using text, a level of abstraction is inevitable. What you want to do is to construct a pattern using **literal** characters and **metacharacters**.

For all our examples, we will use the package `stringr`, which makes the regular expression syntax consistent (there are many *dialects*), and provides a set of easy-to-use functions:

```
library(stringr)
```

All the functions have a common structure. For example, `str_extract` extracts text matching a pattern: `str_extract(text, pattern)`. The simplest possible expression is one in which the pattern is described literally (i.e., we want to find exactly the text we're typing):

```

str_extract("a string of text", "t")

## [1] "t"

str_extract_all("a string of text", "t")

## [[1]]
## [1] "t" "t" "t"

```

Of course, you need to be able to describe much more general patterns. Use the following metacharacters:

- \d Match a digit character (0-9)
- \D Match any character that is not a digit
- \n Match a newline
- \s Match a space
- \t Match a Tab
- \b Match a “word boundary”
- \w Match a “word” character (alphanumeric)
- . Match any character

Some examples (note that to escape characters, you want to use two backslashes — you need to escape the backslash itself!):

```

# find the first digit
str_extract("123.25 grams", "\d")

## [1] "1"

# find word separator + word character + word separator
str_extract("Albert Einstein was a genius", "\b\w\b")

## [1] "a"

# find all digits
str_extract_all("my cell is 773 345 6789", "\d")

## [[1]]
## [1] "7" "7" "3" "3" "4" "5" "6" "7" "8" "9"

# extract all characters
str_extract_all("for example, this and that", ".")

```

```

## [[1]]
## [1] "f" "o" "r" " " "e" "x" "a" "m" "p" "l" "e" " "," " " "t" "h" "i" "s"
## [18] " " "a" "n" "d" " " "t" "h" "a" "t"

```

Of course, you don't want to type `\w` fifteen times, in case you are looking for a string that is 15 characters long! Rather, you can use quantifiers:

- `*` Match zero or more times. Match as many times as possible.
- `*?` Match zero or more times. Match as few times as possible.
- `+` Match one or more times. Match as many times as possible.
- `+?` Match one or more times. Match as few times as possible.
- `?` Match zero or one times. In case both zero and one time match, prefer one.
- `??` Match zero or one times, prefer zero.
- `{n}` Match exactly `n` times.
- `{n,}` Match at least `n` times. Match as many times as possible.
- `{n,m}` Match between `n` and `m` times.

Exercise

What does this do? Try to guess, and then type the command into R

```
str_extract_all("12.06+3.21i", "\\d+\\.\\?\\d+")
my_str <- "most beautiful and most wonderful have been, and are being, evolved."
str_extract(my_str, "\\b\\w{6,10}\\b")
str_extract(my_str, "\\b\\w+\\b")
str_extract(my_str, "w\\w*")
str_extract(my_str, "b\\w*")
str_extract(my_str, "b\\w+?")
str_extract(my_str, "\\s\\wn\\w+")
```

What if you want to match the characters `?, +, *, .?` You will need to escape them: for example, `\. matches the “dot” character.`

You can specify anchors to signal that the match has to be in certain special positions in the text:

- `^` Match at the beginning of a line.
- `$` Match at the end of a line.

```
str_extract("Ah, ba ba ba Barbara Ann", "\\w{2,}$")
```

```
## [1] "Ann"
```

```
str_extract("Ah, ba ba ba Barbara Ann", "^\\w{2,}")
```

```
## [1] "Ah"
```

To match one of several characters, list them between brackets:

```
str_extract("01234567890", "[3120]+")
```

```
## [1] "0123"
```

```

str_extract("01234567890", "[3-5]+")

## [1] "345"

str_extract("supercalifragilisticexpialidocious", "[a-i]{3,}")

## [1] "agi"

```

To match either of two patterns, use alternations:

```

str_extract_all("The quick brown fox jumps over the lazy dog", "fox|dog")

```

```

## [[1]]
## [1] "fox" "dog"

```

If you need more complex alternations, use parentheses to separate the patterns.

Parentheses can also be used to define **groups**, which are used when you want to capture unknown text that is however flanked by known patterns. For example, suppose you want to save the user name of a UofC email:

```

str_match_all("sallesina@uchicago.edu mjsmith@uchicago.edu",
             "\\b([a-zA-Z0-9]*?)@uchicago.edu")

```

```

## [[1]]
##      [,1]          [,2]
## [1,] "sallesina@uchicago.edu" "sallesina"
## [2,] "mjsmith@uchicago.edu"     "mjsmith"

```

Note that you have to use `str_match` or `str_match_all` to obtain information on the groups.

Useful functions

Many functions have a similar `str_***_all` version, returning all matches.

- `str_detect(strings, pattern)` do the strings contain the pattern? Returns a logical vector.
- `str_locate(strings, pattern)` find the character position of the pattern
- `str_extract(strings, pattern)` extracts the first match
- `str_match(strings, pattern)` like extract but capture groups defined by parentheses
- `str_replace(strings, pattern, newstring)` replaces the first matched pattern with `newstring`

Exercises in groups

Extract primers and polymorphic sites

In the file `data/Ptak_etal_2004.txt` you find a text version of the supplementary materials of Ptak *et al.* (PLoS Biology 2004, [doi:10.1371/journal.pbio.0020155](https://doi.org/10.1371/journal.pbio.0020155)).

- Take a look at the file. You see that it first lists the primers used for the study (e.g., TAP2-17-3' -> CTTGGATATAACACCAAACGCA), and then the polymorphic sites for 24 chimpanzees (e.g., .
- Read the text as a single string, intervalled by new lines

```
my_txt <- paste(readLines("../data/Ptak_etal_2004.txt"), collapse="\n")
```

- Use regular expressions to produce a data frame containing the primers used in the study:

```
head(primers)
```

| | ID | Sequence |
|---|-----------|-------------------------|
| 1 | TAP2-1-5' | GAGAACACTTGAACCTGGGAG |
| 2 | TAP2-2-5' | TTGTCCACAGTGTACCACATGA |
| 3 | TAP2-3-5' | TATTTCCTCCTGGGGTTTCCTT |
| 4 | TAP2-4-5' | CATGATGTGTATGCTGAATTG |
| 5 | TAP2-5-5' | ATAGAACAAAGAACCAAAGCCCA |
| 6 | TAP2-6-5' | GGACAACAGATAAAGTTGCCCT |

- Write another regular expression to extract the polymorphic region for each chimp. Use `str_replace_all` to remove extra spaces and newlines. The results should look like:

| | Chimp | Sequence |
|---|-------|---|
| 1 | 311 | ATACCCTGGAGGCAAGAACATCTCCGATAGACGCCAGTCCCTAGTTGT... |
| 2 | 312 | GCACCCTGGAGGCAAGAGTCTTCCGATAGACGCCAGTCCCTAGTTGC... |
| 3 | 313 | GCACCCTGGAGGCAAGAGTCTTCCGATAGACGCCAGTCCCAGTTGT... |
| 4 | 314 | GCACCCTGGAGGCAAGAGTCTTCCGATAGACCCCAGTCCCCAGTTGC... |
| 5 | 317 | GCACCCTGGAGGTGGGGCCCCCAGGAGGCCAGCCCCCGGTCTG... |
| 6 | 320 | GCACCCTGGAGATAAGGGCCCCCAGGAGGCCAGCCCCCGGTCTG... |

A map of Science

Where does science come from? This question has fascinated researchers for decades and even led to the birth of the field of “Science of Science”, where researchers use the same tools they invented to investigate nature to gain insights on the development of science itself. In this exercise, you will build a map of *Science*, showing where articles published in *Science* magazine have originated. You will find two files in the directory `data/MapOfScience`. The first, `pubmed_results.txt`, is the output of a query to PubMed listing all the papers published in *Science* in 2015. You will extract the US ZIP codes from this file, and then use the file `zipcodes_coordinates.txt` to extract the geographic coordinates for each ZIP code.

- Read the file `pubmed_results.txt`, and extract all the US ZIP codes.
- Count the number of occurrences of each ZIP code using `dplyr`.
- Join the table you've created with the data in `zipcodes_coordinates.txt`
- Plot the results using `ggplot2` (either use points with different colors/alphas, or render the density in two dimensions)

Defensive Programming in R

Sarah Cobey

Defensive Programming

- **Goal:** Convince new and existing programmers of the importance of defensive programming practices, introduce general programming principles, and provide specific tips for programming and debugging in R.
- **Audience:** Scientific researchers who use R and believe the accuracy of their code and/or efficiency of their programming could be improved.
- **Installation:** For people who have completed the other tutorials, there is nothing new to install. For others starting fresh, install R and RStudio. To install R, follow instructions at cran.rstudio.com. Then install Rstudio following the instructions at goo.gl/a42jYE. Download the knitr package.

Motivation

Defensive programming is the practice of anticipating errors in your code and handling them efficiently.

If you’re new to programming, defensive programming might seem tedious at first. But if you’ve been programming long, you’ve probably experienced firsthand the stress from

- inexplicable, strange behavior by the code
- code that seems to work under some conditions but not others
- incorrect results or bugs that take days or weeks to fix
- a program that seems to produce the correct results but then, months or years later, gives you an answer that you know must be wrong... thereby putting all previous results in doubt
- the nagging feeling that maybe there’s still a bug somewhere
- not getting others’ code to run or run correctly, even though you’re following their instructions

Defensive programming is thus also a set of practices for preserving sanity and conducting research efficiently. It is an art, in that the best methods vary from person to person and from project to project. As you will see, which techniques you use depend on the kind of mistakes you make, who else will use your code, and the project’s requirements for accuracy and robustness. But that flexibility does not imply defensive programming is “optional”: steady scientific progress depends on it. In general, we need scientific code to be perfectly accurate (or at least have well understood inaccuracies), but compared to other programmers, we are less concerned with security and ensuring that completely naive users can run our programs under diverse circumstances (although standards here are changing).

In the first part of this tutorial, we will review key principles of defensive programming for scientific researchers. These principles hold for all languages, not just R. In the second part, we will consider R-specific debugging practices in more depth.

Part 1: Principles

Part 1 focuses on defense. You saw a few of these principles in Basic Computing 2, but they are important enough to be repeated here.

1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.

2. Write clearly and cautiously.
3. Develop one function at a time, and test it before writing another.
4. Document often.
5. Refactor often.
6. When you run your code, save each “experiment” in a separate directory with a complete copy of the code repository and all parameters.
7. Don’t be *too* defensive.

In part 2, we will focus on what to do when tests (from Principle 3) indicate something is wrong.

Principle 1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.

Many of us have had the experience of writing a long paper only to realize, several pages in, that we in fact need to say something slightly different. Restructuring a paper at this stage can be a real chore. Outlining your code as you would outline a paper avoids this problem. In fact, outlining your code can be even more helpful because it helps you think not just generally about how your algorithm will flow and where you want to end up, but also about what building blocks (functions and containers) you’ll use to get there. Thinking about the “big picture” design will prevent you from coding yourself into a tight spot—when you realize hundreds of lines in that you should’ve been tracking past states of a variable, for instance, but your current containers are only storing the current state. Drafting also makes the actual writing much more easy and fun.

For complex programs, your draft may start as a diagram or flowchart showing how major functions and variables relate to one another or brief notes about each of step of the algorithm. These outlines are known as pseudocode, and there are myriad customs for pseudocode and programmers loyal to particular styles. For simple scripts, it is often sufficient to write pseudocode as placeholder comments. For instance, if we are simulating a population in which individuals can be born or die (and nothing else happens), we could write:

```
# initialize population size (N), birth rate (b), death rate (d),
# total simulation time (t_max), and time (t=0) while N > 0 and t < t_max
# ...generate random numbers to determine whether next event is
# birth or death, and time to event (dt) ...update N
# (increment if birth, decrement if death) and update time t to t+dt
```

Here, **b** and **d** are per capita rates. This is an example of the Gillespie algorithm. It was initially developed as an exact stochastic approach for simulating chemical reaction kinetics, and it is widely used in biology, chemistry, and physics.

Can you see some limitations of the pseudocode so far? First, it lacks obvious modularity, though this is partly due to its vagueness. The first step under the `while` loop could become its own function that is defined separately. Second, it is missing a critical feature, in that it’s not obvious what is being output: do we want the population size at the end of the simulation, or the population size at each event? If the latter, we may need to initialize a container, such as a dataframe, in which we store the value of **N** and **t** at every event. After further thought, we might decide such a container would be too big—perhaps we only need to know the value of **N** at 1/1000th the typical rate of births, and so we might introduce an additional loop to store **N** only when the new time (**t+dt**) exceeds the most recent prescribed observation/sampling time. This sampling time would need to be stored in an extra variable, and we could also make the sampling procedure into its own function. And maybe we want a function to plot **N** over time at the end.

The next stage of drafting is to consider how the code might go wrong, and what it would take to convince ourselves that it is accurate. We’ll spend more time on this later, but now is the time to think of every possible sanity check for the code. Sometimes this can lead to changes in code design.

Exercise

What sanity checks and tests would you include for the code above?

Some examples:

- Initial values of b , d , and t_{\max} should be non-negative and not change
- The population size N should probably start as a positive whole number and never fall below zero
- If the birth and death rates are zero, N should not change
- If the birth rate equals the death rate, on average, N should not change when it is large
- The ratio of births to deaths should equal the ratio of the birth rate to the death rate, on average
- The population should, on average, increase exponentially at rate $b-d$ (or decrease exponentially if $d>b$)

Some of these criteria arise from common sense or assumptions we want to build into the model (for instance, that the birth and death rates aren't negative), and others we know from the mathematics of the system. For instance, the population cannot change if there are no births and deaths, and it must increase on average if the birth rate exceeds the death rate. It helps that the Gillespie algorithm represents kinetics that can be written as simple differential equations. However, because the simulations are stochastic, we need to look at many realizations (randomizations, trajectories) to ensure there aren't consistent biases. We must use statistics to confirm that the distribution of N in 10,000 simulations after 100 time units is not significantly different from what we would predict mathematically.

The bottom line is that we have identified some tests for the (1) inputs, (2) intermediate variable states (like N), (3) final outputs to test that the program is running correctly. Note that one of our tests, the fraction of birth to death events, is not something we were originally tracking, and thus we might decide now to create a separate function and variables to handle these quantities. We will talk in more detail about how to implement these tests in Principle 3. Generally, tests of specific functions are known as **unit tests**, and tests of aggregate behavior (like the trajectories of 10,000 simulations) are known as **system tests**. As a general principle, we need to have both, and we need as much as possible to compare their outputs to analytic expectations. It is also very useful to identify what you want to see right away. For instance, you may want to write a function to plot the population size over time *before* you code anything else because having immediate visual feedback can be extremely helpful.

"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do." -Donald Knuth

This is also a good time to draft very high-level documentation for your code, for instance in a `readme.MD` file. What are the inputs, what does the code do, and what does it return?

Exercise

Assume there are two discrete populations. Each has nonoverlapping generations and the same generation time. Their per capita birth rates are b_1 and b_2 . Some of the newborns migrate between populations.

- Write pseudocode to calculate the distribution of population frequencies after 100 generations.
- Is your code optimally modular?
- What are the inputs and outputs of the program? Of each function?
- How could you test the inputs, functions, and overall program?
- Discuss your approach with your neighbor.

Principle 2. Write clearly and cautiously.

You're already on your way to writing clearly and cautiously if you outline your program before you start writing it. Here we'll discuss some practices to follow as you write.

A general rule is that it is more important for scientific code to be readable and correct than it is for it to be fast. Do not obsess too much about the efficiency of the code when writing.

"Premature optimization is the root of all evil." -Donald Knuth

Develop useful conventions for your variable and function names. There are many conventions, some followed more religiously than others. It's most important to be consistent within your own code.

- Don't make yourself and others guess at meaning by making names too short. It's generally better to write out `maxSubstitutionRatePerSitePerYear` or `max.sub.rate.per.site.yr` than `maxSR`.
- However, very common variables should have short names.
- When helpful, incorporate identifiers like `df` (data frame), `ctr` (counter), or `idx` (index) into variable names to remind you of their purpose or structure.
- Customarily, variables that start with capital letters indicate global scope in R, and function names also start with capital letters. People argue about conventions, and they vary from language to language. Here's Google's style guide.

Do not use magic numbers. "Magic numbers" refer to numbers hard-coded in your functions and main program. Any number that could possibly vary should be in a separate section for parameters. The following code is not robust:

```
while ((age >= 5) && (inSchool == TRUE)) {  
    yearsInSchool = yearsInSchool + 1  
}
```

We may decide the age cutoff of 5 is inappropriate, but even if we never do, being unable to change the cutoff limits our ability to test the code. Better:

```
while ((age >= AgeStartSchool) && (inSchool == TRUE)) {  
    yearsInSchool = yearsInSchool + 1  
}
```

Most of the time, the only numbers that should be hard-coded are 0, 1, and pi.

Use labels for column and row names, and load functions by argument.

Don't force (or trust) yourself to remember that the first column of your data frame contains the time, the second column contains the counts, and so on. When reviewing code later, it's harder to interpret `cellCounts[,1]` than `cellCounts$time`.

In the same vein, if you have a function taking multiple inputs, it is safest to pass them in with named arguments. For instance, the function

```
BirthdayGiftSuggestion <- function(age, budget) {  
    # ...  
}
```

could be called with

```
BirthdayGiftSuggestion(age = 30, budget = 20)  
# or  
BirthdayGiftSuggestion(30, 20)
```

but the former is obviously safer.

Avoid repetitive code. If you ever find yourself copying and pasting a section of code, perhaps modifying it slightly each time, stop. It's almost certainly worth writing a function instead. The code will be easier to read, and if you find an error, it will be easier to debug.

Principle 3. Develop one function at a time, and test it before writing another.

The first part of this principle is easy for scientists to understand. When building code, we want to change one thing at a time. It makes it easier to understand what's going on. Thus, we start by writing just a single function. It might not do exactly what we want it to do in the final program (e.g., it might contain mostly

placeholders for functions it calls that we haven't written yet), but we want to be intimately familiar with how our code works in every stage of development.

The second part of this principle underscores one of the most important rules in defensive programming: **do not believe anything works until you have tested it thoroughly, and then keep your guard up.** *Expect* your code to contain bugs, and leave yourself time to play with the code (e.g., by trying to "break" it) until you can convince yourself they are gone. This involves an extra layer of defensive programming beyond the straightforward good practices discussed in Principle 2. Testing the code as you build it makes it much faster to find problems.

Unit tests. Unit tests are tests on small pieces of code, often functions. An intuitive and informal method of unit testing is to include `print()` statements in your code.

Here's a function to calculate the Simpson Index, a useful diversity index. It gives the probability that two randomly drawn individuals belong to the same species or type:

```
SimpsonIndex <- function(C) {  
  print(paste(c("Passed species counts:", C), collapse=" "))  
  fractions <- C / sum(C)  
  print(paste(c("Fractions:", round(fractions, 3)), collapse=" "))  
  S <- sum(fractions ^ 2)  
  print(paste("About to return S =",S))  
  return(S)  
}  
  
# Simulate some data  
numSpecies <- 10  
maxCount <- 10 ^ 3  
fakeCounts <- floor(runif(numSpecies, min = 1, max = maxCount))  
  
# Call function with simulated data  
S <- SimpsonIndex(fakeCounts)  
  
## [1] "Passed species counts: 423 116 758 530 82 389 597 832 927 943"  
## [1] "Fractions: 0.076 0.021 0.135 0.095 0.015 0.07 0.107 0.149 0.166 0.168"  
## [1] "About to return S = 0.127786909760388"
```

It's very useful to print values to screen when you are writing a function for the first time and testing that one function. When you've drafted your function, I recommend walking through the function with `print()` and comparing the computed values to calculations you perform by hand or some other way. It can also be useful to do this at a very high level (more on that later).

The problem with relying on `print()` is that it rapidly provides too much information for you to process, and hence errors can slip through.

Assertions

A more reliable way to catch errors is to use assertions. Assertions are automated tests embedded in the code. The built-in function for assertions in R is `stopifnot()`. It's very simple to use.

Let's remove the `print` statements and add a check to our input data:

```
SimpsonIndex <- function(C) {  
  stopifnot(C > 0)  
  fractions <- C / sum(C)  
  S <- sum(fractions ^ 2)  
  return(S)  
}
```

If each element of our abundances vector `C` is positive, `stopifnot()` will be TRUE, and the program will continue. If any element does not satisfy the criterion, then FALSE will be returned, and execution will terminate. Explore for yourself:

```
# Simulate two sets of data
numSpecies <- 10
maxCount <- 10 ^ 3
goodCounts <- floor(runif(numSpecies, min = 1, max = maxCount))
badCounts <- floor(runif(numSpecies, min = -maxCount, max = maxCount))

# Call function with each data set
S <- SimpsonIndex(goodCounts)
S <- SimpsonIndex(badCounts)
```

This gives a very literal error message, which is often enough when we are still developing the code. But what if the error might arise in the future, e.g., with future inputs? We can use the built-in function `stop()` to include a more informative message:

```
SimpsonIndex <- function(C) {
  if(any(C < 0)) stop("Species counts should be positive.")
  fractions <- C / sum(C)
  S <- sum(fractions ^ 2)
  return(S)
}
```

Now try it with `badCounts` again.

What about warnings? For instance, our calculation of the Simpson Index is an approximation: the index formally assumes we draw without replacement, but we compute $S = \sum p^2$, where p is the fraction of each species. It should be $S = \sum \frac{n(n-1)}{N(N-1)}$, where n is the abundance of each species n and N the total abundance. This simplification becomes important at small sample sizes. We could add a warning to alert users to this issue:

```
SimpsonIndex <- function(C) {
  if(any(C < 0)) stop("Species counts should be positive.")
  if((mean(C) < 20) || (min(C) < 5)) {
    warning("Small sample size. Result will be biased. Consider corrected index.")
  }
  fractions <- C / sum(C)
  S <- sum(fractions ^ 2)
  return(S)
}

smallCounts <- runif(10)
S <- SimpsonIndex(smallCounts)

## Warning in SimpsonIndex(smallCounts): Small sample size. Result will be
## biased. Consider corrected index.
```

The main advantage of `warning()` over `print()` is that the message is red and will not be confused with expected results, and warnings can be controlled (see `?warning`).

You could make a case that `warning()` should be `stop()`. In general, with defensive programming, you want to halt execution quickly to identify bugs and to limit misuse of the code.

Exercise

- What other input checks would make sense with `SimpsonIndex()`?

- You can see how the code would be more readable and organized if most of that function were dedicated to actually calculating the Simpson Index. Draft a separate function, `CheckInputs()`, and include all tests you think are reasonable.
- When you and a neighbor are done, propose a bad or dubious input for their function and see if it's caught.

There are many packages that produce more useful assertions and error messages than what is built into R. See, e.g., `assertthat` and `testit`.

Exception handling

Warnings and errors are considered “exceptions.” Sometimes it is useful to have an automated method to handle them. R has two main functions for this: `try()` allows you to continue executing a function after an error, and `tryCatch()` allows you to decide how to handle the exception.

Here’s an example:

```
UsefulFunction <- function(x) {
  value <- exp(x)
  otherStuff <- rnorm(1)
  return(list(value, otherStuff))
}

data <- "2"
results <- UsefulFunction(data)
print(results)
```

Now `results` is quite a disappointment: it could’ve at least returned a random number for you, right? You could instead try

```
UsefulFunction <- function(x){
  value <- NA
  try(value <- exp(x))
  otherStuff <- rnorm(1)
  return(list(value, otherStuff))
}
results <- UsefulFunction(data)
print(results)

## [[1]]
## [1] NA
##
## [[2]]
## [1] 0.3879199
```

Even though the function still can’t exponentiate a string (`exp("2")` still fails), execution doesn’t terminate. If we want to suppress the error message, we can use `try(..., silent=TRUE)`. This obviously carries some risk!

We could make this function even more useful by handling the error responsibly with `tryCatch()`:

```
UsefulFunction <- function(x){
  value <- NA
  tryCatch ({
    message("First attempt at exp()...")
    value <- exp(x)},
    error = function(err){
      message(paste("Darn:", err, " Will convert to numeric."))
      value <- exp(as.numeric(x))})
```

```

        }
    )
otherStuff <- rnorm(1)
return(list(value, otherStuff))
}
results <- UsefulFunction(data)
print(results)

```

It is also possible to assign additional blocks for warnings (not just errors). The `<<-` is a way to assign to the value in the environment one level up (outside the `error=` block).

Exercise

The new package `ggjoy` works with package `ggplot2` to show multiple distributions in a superimposed but interpretable way. Let's say we want to run the following code:

```

library(ggplot2)
library(ggjoy)
ggplot(iris, aes(x = Sepal.Length, y = Species)) + geom_joy()

```

You probably don't have `ggjoy` installed yet, so you'll get an error. Use `tryCatch()` so that the package is installed if you do not have it and then loaded.

Test all the scales!

It's important to consider multiple scales on which to test as you develop. We've focused on unit tests (testing small functions and steps) and testing inputs, but it is easy to have correct subroutines and incorrect results. For instance, we can be excellent at the distinct activities of toasting bread, buttering bread, and eating bread, but we will fail to enjoy buttered toast for breakfast if we don't pay attention to the order.

With scientific programming, it is critical to simplify code to the point where results can be compared to analytic expectations. You saw this in Principle 1. It is important to add functions to check not only inputs and intermediate results but also larger results. For instance, when we set the birth rate equal to the death rate, does the code reliably produce a stable population? We can write functions to test for precisely such requirements. These are **system tests**. When you change something in your code, always rerun your system tests to make sure you've not messed something up. Often it's helpful to save multiple parameter sets or data files precisely for these tests.

It's hard to overstate the importance of taking a step back from the nitty-gritty of programming and asking, Are these results reasonable? Does the output make sense with different sets of extreme values? Schedule time to do this, and update your system tests when necessary.

Principle 4. Document often.

It is helpful to keep a running list of known “issues” with your code, which would include the functions left to implement, the tests left to run, any strange bugs/behavior, and features that might be nice to add later. Sites like GitHub and Bitbucket allow you to associate issues with your repositories and are thus very helpful for collaborative projects, but use whatever works for you. Having a formal list, however, is much safer than sprinkling to-do comments in your code (e.g., `# CHECK THIS!!!`). It's easy to miss comments.

Research code will always need a `readme` describing the software's purpose and implementation. It's easiest to develop it early and update as you go.

Principle 5. Refactor often.

To refactor code is to revise it to make it clearer, safer, or more efficient. Because it involves no changes in the scientific outputs (results) of the program, it might feel pointless, but it's usually not. Refactor when

you realize that your variable and function names no longer reflect their true content or purpose (rename things quickly with **Ctrl + Alt + Shift + M**), when certain functions are obsolete or should be split into two, when another data structure would work dramatically better, etc. Any code that you'll be working with for more than a week, or that others might ever use, should probably be refactored a few times. Debugging will be easier, and the code will smell better.

Important tip, repeated: Run unit and system tests after refactoring to make sure you haven't messed anything up. This happens more than you might think.

Principle 6. When you run your code, save each “experiment” in a separate directory with a complete copy of the code repository and all parameters.

When you're done developing the code and are using it for research, keep results organized by creating a separate directory for each execution of the code that includes not only the results but also the precise code used to generate the results. This way, you won't accidentally associate one set of parameters with results that were in fact generated by another set of parameters. Here's a sample workflow, assuming your repository is located remotely on GitHub, and you're in a UNIX terminal:

```
$ mkdir 2017-09-15_rho=0.5  
$ cd 2017-09-15_rho=0.5  
$ git clone git@github.com:MyName/my-repo
```

If we want, we can edit and execute our code from within R or RStudio, but we can also keep going with the command line. Here we are using a built-in UNIX text editor known as emacs. If you are a glutton for punishment, you could instead use vi(m).

```
$ cd my-repo  
$ emacs parameters.json // (edit parameters, with rho=0.5)  
$ Rscript mycode.R
```

Keeping your experiments separate is going to save your sanity for larger projects when you repeatedly revise your analyses. It also makes isolating bugs easier.

Principle 7. Don't be *too* defensive.

This is not necessary:

```
myNumbers <- seq(from = 1, to = 500, length.out = 20)  
stopifnot(length(myNumbers) == 20)
```

It's fine to check stuff like this when you're getting the hang of a function, but it doesn't need to be in the code. Code with too many defensive checks becomes a pain to read (and thus debug). Try to find a balance between excess caution and naive optimism. Good luck!

Part 2: Debugging in R

Part 1 introduced principles that should minimize the need for aggressive debugging. You're in fact already debugging if you're regularly using input, unit, and system tests to make sure things are running properly. But what happens when despite your best efforts, you're not getting the right result?

We'll focus on more advanced debugging tools in this part. First, some general guidelines for fixing a bug:

- *Isolate the error and make it reproducible.* Try to strip the error down to its essential parts so you can reliably reproduce the bug. If a function doesn't work, copy the code, and keep removing pieces that are non-essential for reproducing the error. When you post for help on the website stackoverflow, for

instance, people will ask for a MRE or MWE—a minimum reproducible (working) example. You need to have not only the pared code but also the inputs (parameter values and the seeds of any random number generators) that cause the problem.

- *Use assertions and debugging tools to hone in on the problem.* We've already seen how to use `stop()` and `stopifnot()` to identify logic errors in unit tests. We'll cover more advanced debugging here.
- *Change/Test one thing at a time.* This is why we develop only one function at a time.

Tracing calls

If an error appears, a useful technique is to see the call stack, the sequence of functions immediately preceding the error. We can do this in R using `traceback()` or in RStudio.

The following example comes from a nice tutorial by Hadley Wickham:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

You can see right away that running this code will create an error. Try it anyway in RStudio. Click on the “Show Traceback” to the right of the error message. What you’re seeing is the stack, and it’s helpfully numbered. At the bottom we have the most recent (proximate) call that produced the error, the preceding call above it, and so on. (If you’re working from a separate R file that you sourced for this project, you’ll also see the corresponding line numbers next to each item in the stack.) If you’re in R, you can run `traceback()` immediately after the error.

Seeing the stack is useful for checking that the correct functions were indeed called, but it can suggest how to trace the error back in a logical sequence. But we can often debug faster with more information from RStudio’s debugger.

Examining the environment

Let's pretend we have a group of people who need to be assigned random partners. These partnerships are directed (so person A may consider his partner person B, but B's partner is person F), and we'll allow the possibility of people partnering with themselves. Some code for this is in the file `BugFun.R`. (It's not terribly efficient code, but it is useful for this exercise.)

```
source("BugFun.R")
peopleIDs <- seq(1:10)
pairings <- AssignRandomPartners(peopleIDs)
```

Try running this a few times. We have an inconsistent bug.

We can use breakpoints to quickly examine what's happening at different points in the function. With breakpoints, execution stops on that line, and environmental states can be inspected. In RStudio, you can create breakpoints by clicking to the left of the line number in the `.R` file. A red dot appears. You can then examine the contents of different variables in debug mode. To do this, you have to make sure you have the right setting defined: Debug > On Error > Break in Code.

Exercise

Use breakpoints to identify the error(s) in the `AssignRandomPartners()` function. Go to `BugFun.R` and attempt to run the last line. Decide on a place to start examining the code. If you have adjusted your settings, the debug mode should start automatically once you define a breakpoint and try to run the code again. (If a message to source the file appears, follow it.) Your console should now have `Browse[1]>` where it previously had only `>`.

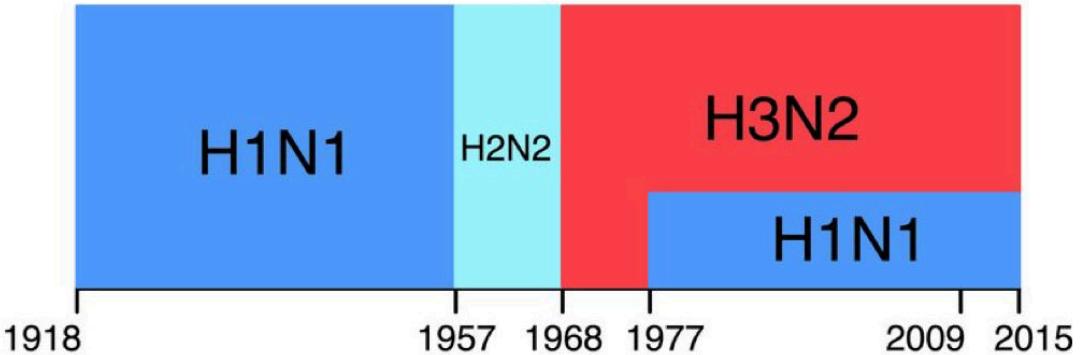


Figure 1: *Endemic influenza subtypes since 1918 (Gostic et al. 2016)*.

The IDE is now giving you lots of information. The green arrow shows you where you are in the code. The line that is about to be executed is in yellow. Anything you execute in the console shows states from your current environment. Test this for yourself by typing a few variables in the console. You can see these values in the Environment pane in the upper right, and you can also see the stack in the middle right.

If you hit enter at the console, it will advance you to the next step of the code. But it is good to explore the Console buttons (especially ‘Next’) to work through the code and watch the Environment (data and values) and Traceback as they change. By calling the function several times, you should be able to convince yourself of the cause of the error.

Much more detail about browsing in debugging mode is available at [here](#).

When you are done, exit the debug mode by hitting the Stop button in the Console.

In R, you can insert the function `browser()` on some line of the code to enter debugging mode. This is also what you have to use if you want to debug directly in R Markdown.

Programming Challenge

Avian influenza cases in humans usually arise from two viral subtypes, H5N1 and H7N9. An interesting observation is that the age distributions for H5N1 and H7N9 cases differ: older people are more likely to get very sick and die from H7N9, and younger people from H5N1. There’s no evidence for age-related differences in exposure. A recent paper showed that the risk of severe infection or death with avian influenza from 1997-2015 could be well explained by a simple model that correlated infection risk with the subtype of seasonal (non-avian) influenza a person was first exposed to in childhood. Different subtypes (H1N1, H2N2, and H3N2) have circulated in different years. Perhaps because H3N2 is more closely related to H7N9 than to H5N1, people with primary H3N2 infections seem protected from severe infections with H7N9. The complement is true for people first infected with H1N1 or H2N2 and later exposed to H5N1.

Of course, we do not know the full infection history of any person who was hospitalized with avian influenza. We only know the person’s age and the year of hospitalization or death. To perform their analysis, the authors needed to calculate the probability that each case had a primary infection with each subtype, i.e., the probability that a person born in a given year was first infected with each subtype. Your challenge is to calculate these probabilities.

The authors had to make some assumptions. First, they assumed that the risk of influenza infection is 28% in each year of life. Second, they assumed that the frequency of each circulating subtype could be inferred from the numbers of isolates sampled (primarily in hospitals) each year. These counts are given in `subtype_counts.csv`. [1]

The challenge: For every year between 1960 and 1996, calculate the probability that a person born in that year had primary infection with H1N1, H2N2, and H3N2. You must program defensively to pull this off.

When you have found the solution, go to stefanoallesina.github.io/BSD-QBio4 and follow the link **Submit solution to defensive programming challenge** to submit your answer (alternatively, you can go directly to goo.gl/forms/NjdZUyAjs9HBWMqL2).

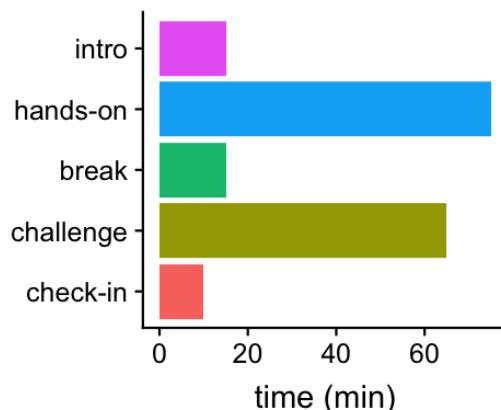
[1] The counts are actually given for each influenza season in the U.S., which is slightly different from a calendar year, but you can ignore this. You'll notice that "1" and "0" are used where we know (or will assume) that only one subtype was circulating. The authors made several other assumptions, but this is good enough for now.

Data visualization tutorial: exploring data and telling stories using ggplot2*

Peter Carbonetto University of Chicago

In this lesson, you will use ggplot2 to create effective visualizations of data. The ggplot2 package is a powerful set of plotting functions that extend the base plotting functions in R. You will learn that creating effective visualizations *hinges on good data preparation*. (In reality, good data preparation in the lab can take days or weeks, but we can still illustrate some useful data preparation practices.) The main difference with Advanced Computing 1 is that we take a more in-depth look at ggplot2 and plotting strategies.

Outline



Motivation

Why plot? One reason: to gain insight into your data. This is “exploratory data visualization.”. Another reason: to tell a story (e.g., for a scientific paper). Both require lots of iteration and refinement to get right. So you need to learn to efficiently create plots, and efficiently improve them.

One solution is the *programmatic approach* to data visualization. This will allow you to:

1. Create an endless variety of plots.
2. Reuse code to quickly create sophisticated plots.

ggplot2 is the preferred approach to creating plots in R.

*This document is included as part of the Data Visualization tutorial packet for the BSD qBio Bootcamp, MBL, 2019. Current version: July 31, 2019; Corresponding author: pcarbo@uchicago.edu. Thanks to John Novembre and Matthew Stephens for their support and guidance.

Setup

If you haven't already done so, install these packages:

```
install.packages("readr")
install.packages("ggplot2")
install.packages("cowplot")
```

Once installed, load the packages into your R environment:

```
library(readr)
library(ggplot2)
library(cowplot)
```

Your first plot

To test that your R environment is properly set up for plotting with ggplot2, plot some fuel economy data from the EPA:

```
epa <- read.csv("epa.csv")
ggplot(mpg, aes(displ, cty, color = class)) + geom_point()
```

Hands-on exercise: “Rising Dough, Rising Neighbourhoods”

The scenario

You have begun a summer internship at the [Mansueto Institute for Urban Innovation](#). The institute has formed a partnership with the City of Chicago to study trends in different neighborhoods across the city.

For your summer project, you will explore ways to study neighbourhoods trends by aggregating publicly available data on pizza restaurants. You will prepare a report summarizing your results. Evocative plots will be critical to a successful report.

Setup

- Make sure you have downloaded the tutorial packet.
- Locate the files for this exercise on your computer (see “Materials” below).
- Make sure your R working directory is the same directory containing the tutorial materials; run `getwd()` to check this.

Materials

- [Food_Inspections.csv.gz](#): CSV file containing food inspection data downloaded from the [Chicago Data Portal](#).

Load the data

Use `read_csv` from the `readr` package instead of the standard function, `read.csv`, because `read_csv` is much faster for large data sets.

```
data.file <- "Food_Inspections.csv.gz"  
dat       <- read_csv(data.file)  
class(dat) <- "data.frame"
```

Run a few basic commands to inspect the data:

Prepare the data for plotting

Creating sophisticated plots using `ggplot2` requires relatively little effort *provided the data are in the right form*. Before creating the plots, you must carefully prepare the data so that it is ready for plotting. *Many analysis mistakes are due to poor data preparation!*

First, remove the columns you don't need (since unnecessary data can be distracting), and change the names of the columns to make the code easier to write. (Note that "DBA" is an abbreviation of "doing business as".)

```
cols      <- c(2:5, 7, 8, 11, 13, 15, 16)  
colnames <- c("dba", "aka", "license", "type", "address", "city",  
            "date", "results", "latitude", "longitude")  
dat       <- dat[cols]  
names(dat) <- colnames
```

Now extract the year from the inspection date. This should create a new column, "year".

```
get.third.entry <- function(x) { return(x[3]) }  
out      <- strsplit(dat$date, "/")  
dat$year <- sapply(out, get.third.entry)  
dat$year <- as.numeric(dat$year)
```

Question: How many restaurant inspections were performed in each year?

Now, write code to filter out rows based on these criteria: you want to keep only data on Chicago restaurants, and remove all rows for which one or more of the license and geographic co-ordinates (latitude and longitude) are not available. Write your R code here to accomplish this:

Recall, the goal is to visualize data on pizza restaurants. So you need to extract the rows of the data frame that correspond to pizza restaurants. Write your R code to do this.

You are nearly done preparing the data. Some restaurants were inspected multiple times, and you don't want to overcount the number of restaurants. The "license" column is useful for uniquely identifying restaurants.

First, write code to sort the rows of the table by year of inspection so that the first rows show the earliest inspections (*hint*: use the `order` function):

Question: What is the difference between "sort" and "order"?

Next, write code to remove rows so that there is only one row per pizza location:

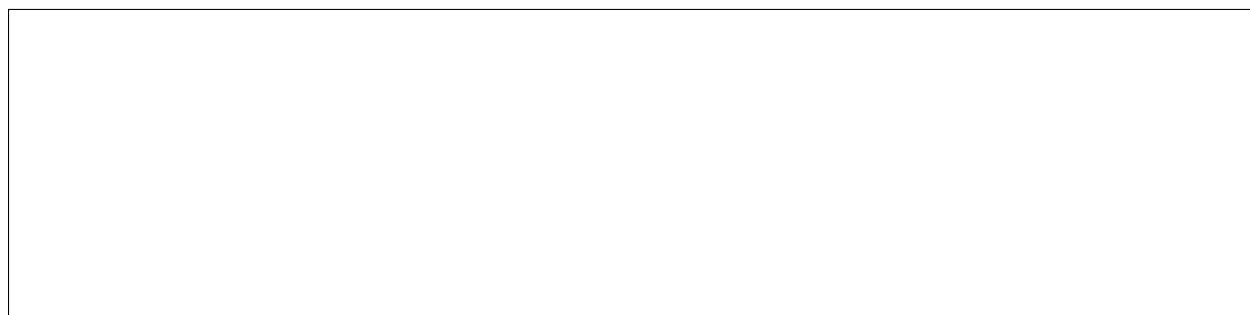
Question: How do you double-check that each license appears only once in the data frame?

Your first plot

Your first plot is a bar chart showing an estimate of the number of new pizza restaurants in Chicago by year. Write code that uses the `table` function to create a table containing number of inspected restaurants by year:



To plot these counts in ggplot2, we need to create a data frame. Write code to convert counts to a data frame with two columns, “year” and “count”:



You now have a data frame with two columns, “year” and “count”. Go ahead and plot it! First, specify the data frame, and the mapping between variables (columns) and plotting elements:

```
a <- aes(x = year, y = count)
p <- ggplot(counts, a)
```

Specify *how* to plot the data points:

```
g <- geom_col()
p <- ggplot_add(g, p)
```

Where is the plot? The print function tells R to draw the plot to the screen:

```
print(p)
```

It is Best Practice to label your axes clearly. Improve the y-axis label:

```
out <- labs(y = "number of locations")
p <- ggplot_add(out, p)
```

Here is equivalent code that is more concise (and more commonly used), but also more difficult to understand because it is less clear what the “+” does:

```
p <- ggplot(counts,aes(x = year, y = count)) +  
  geom_col(width = 0.5) +  
  labs(y = "number of locations")
```

I recommend using the less concise syntax until you are comfortable with ggplot2.

All plots in ggplot2 require these three elements:

1. A data frame.
2. An “aesthetic mapping” that declares how columns are mapped to plot features (axes, shapes, colors, etc.).
3. A “geom”, short for “geometric object,” that specifies the type of plot.

All plots are created by *adding layers*. Layers are added with `ggplot_add` or `+`.

Your second plot

For the second plot, you will create a map of the pizza restaurants. This should be easy *since you have already carefully prepared the data*. Write R code to do this:

Try plotting the restaurants with different shapes—which one works best?

One of the powerful features of ggplot2 is that you can use it to generate a wide variety of plots, and each plot is created in a very similar fashion. For example, modify your code above to draw a contour plot showing the density of pizza restaurants throughout the city:

Based on this plot, which areas of the city have the greatest density of pizza restaurants? Is Hyde Park visible from the contours?

Overlay the contour plot with the points:

Now, try to add color to the points to highlight the “hottest” neighbourhoods:

This is an example where it is important to get the colour scheme right. Write R code that uses function `scale_color_gradient2` to highlight more recent pizza restaurants in warmer (red) colours:

All functions starting with `scale_` modify the behaviour of the geoms.

Now combine the plots. This is easily done with `plot_grid` from the `cowplot` package:

```
p12 <- plot_grid(p,p2,labels = c("A","B"))
```

Finally, save your plot using the `ggsave` function. PNG is a commonly used format to share images.

Follow-up programming challenges

Your project advisor says, “Great work. However, I would like a few changes to better present the results. First, create a line chart to better show the trend in pizza restaurants. For the bar chart, color the bars so that they match the colors in the map. Also, a PNG won’t work for the report—please send a PDF instead.”

These improvements are left as a programming challenge.

1. Your first plot was a bar chart showing an estimate of the number of new pizza restaurants per year in Chicago. Sometimes lines are more effective than bars for showing trends.
 - To show the trend as a line plot, how would you modify the plotting code above to do this? *Hint:* try using `geom_line`.
 - What change do you need to make to the `counts` data frame so that counts can be plotted as lines?
 - Which do you find more effective, the line plot or the bar chart? Identify one benefit of one plot over the other.
 - To submit your response, you will need to upload a file containing your final plot. Use `ggsave` to save your plot as a file.
2. Your project advisor has asked you to colour the bars in the bar chart so that they align with the colours used in the map.
 - What change do you need to make to the `counts` data frame so that the years can be mapped to colors? *Hint:* The code used for creating the line plot may be useful here, too.
 - Submit the *combined* plot (bar chart & map). When submitting your plot, save it as a PDF using `ggsave`.
 - Why did your advisor request a PDF? Why is a PNG file less suitable for a final report or publication?

Main programming challenge: “Mapping the genetic basis of physiological and behavioral traits in outbred mice”

In this programming challenge, you will use simple visualizations to gain insight into biological data.

You have finished your summer internship at the Mansueto Institute. You are embarking on your first graduate research project in a lab studying the genetics of physiological and behavioral traits in mice. The lab has just completed a large study of mice from an outbred mouse population, “CFW” (a contraction of “Carworth Farms White”, the names of the scientists who bred the first CFW mice). The ultimate aim of the study is to identify genetic contributors to variation in behaviour and musculoskeletal traits.

Note: These challenges are roughly ordered in increasing level of complexity. Do not be discouraged if you have difficulty completing every one!

Instructions

- Locate the files for this exercise on your computer (see “Materials” below).
- Make sure your R working directory is the same directory containing the tutorial materials; use `getwd()` to check this.
- Some of the programming challenges require uploading an image file containing a plot. Use `ggsave` to save your plot as a file. Any standard image format (e.g., PDF, PNG) is acceptable.
- No additional R packages are needed beyond what you used in the examples above.

Materials

- **pheno.csv:** CSV file containing physiological and behavioral phenotype data on 1,219 male mice from the CFW outbred mouse stock. Data are from [Parker et al, 2016](#). Use `readpheno.R` to read the phenotype data from the CSV file into a data frame. After filtering out some of the samples, this script should create a new data frame, `pheno`, containing phenotype data on 1,092 samples (rows).
- **hmdp.csv:** CSV file containing bone-mineral density measurements taken in 878 male mice from the Hybrid Mouse Diversity Panel (HMDP). Data are from [Farber et al, 2011](#). To load the data into your R environment, run this code:

```
hmdp <- read.csv("hmdp.csv", stringsAsFactors = FALSE)
```

This will create a data frame, `hmdp`, containing BMD data on 878 mice (rows).

- **gwscan.csv:** CSV file containing results of a “genome-wide scan” for abnormal BMD. Association *p*-values were computed using [GEMMA](#) 0.96. To read the results of the genome-wide scan, run the following code:

```
gwscan <- read.csv("gwscan.csv", stringsAsFactors = FALSE)
gwscan <- transform(gwscan, chr = factor(chr, 1:19))
```

This will create a data frame, `gwscan`. Each row of the data frame is a genetic variant (a single nucleotide polymorphism, or “SNP”). The columns are chromosome (“chr”), base-pair position on the chromosome (“pos”), and the *p*-value for a test of association between variant genotype and trait value (“abnormalBMD”). The value stored in the “abnormalBMD” column is $-\log_{10}(P)$, where P is the *p*-value.

- **geno_rs29477109.csv:** CSV file containing estimated genotypes at one SNP (rs29477109) for 1,038 CFW mice. Use the following code to read the genotype data into your R environment:

```
geno <- read.csv("geno_rs29477109.csv", stringsAsFactors = FALSE)
geno <- transform(geno, id = as.character(id))
```

This will create a new data frame, `geno`, with 1,038 rows (samples). The genotypes are encoded as “dosages”—that is, the expected number of times the alternative allele is observed in the genotype. This will be an integer (0, 1, 2), or a real number between 0 and 2 when there is some uncertainty in the estimate of the genotype. For this SNP, the reference allele is T and the alternative allele is C. Therefore, dosages 0, 1 and 2 correspond to genotypes TT, CT and CC, respectively (genotypes CT and TC are equivalent).

- **wtccc.png:** Example genome-wide scan (“Manhattan plot”) from Fig. 4 of the [WTCCC paper](#). The *p*-values highlighted in green show the regions of the human genome most strongly associated with Crohn’s disease risk.

General tips

- Some “geoms” you may find useful: `geom_point`, `geom_histogram`, `geom_boxplot`.
- In some cases it may be useful to convert a column to a *factor*.
- Coming up with the answers will involve skills other than programming, so collaborate!

Part A: Exploratory analysis of muscle development and conditioned fear data

Your first task is to create plots to explore the data.

1. A basic initial step in an exploratory analysis is to visualize the distribution of the data. It is often convenient if the distribution is normal, or “bell shaped”.
 - Visualize the empirical distribution of tibialis anterior (TA) muscle weight (column “TA”) with a histogram. Units are mg. *Hint:* Try using function `geom_histogram`.
 - Is the distribution of TA weight roughly normal? Are there mice with unusually large or unusually small values (“outliers”)? If so, how many “outliers” are there? (Unusually small or large values can lead to misleading results in some statistical tests.)
2. It is also important to understand relationships among measured quantities. For example, the development of the tibia bone (column “tibia”) could influence TA muscle weight. Create a scatterplot (`geom_point`) to visualize the relationship between TA weight and tibia length. (Tibia length units are mm.) Based on this plot, what can you say about the relationship between TA weight and tibia length? Quantify this relationship by fitting a linear model, before and after removing the outlying TA values. (*Hint:* Use the `lm` and `summary` functions. See also the “`r.squared`” return value in `help(summary.lm)`.)
3. The “AvToneD3” column contains data collected from a behavioral test called the “Conditioned Fear” test.
 - Visualize the empirical distribution of AvToneD3 (“freezing to cue”) with a histogram. Is the distribution of AvToneD3 approximately normal?
 - Freezing to cue is a proportion (a number between 0 and 1). A common way to obtain a more “normal” quantity is to transform it using the “logit” function¹. Visualize the empirical distribution of the logit-transformed phenotype. Is the transformed phenotype more “bell shaped”? After the transformation, do you observe unusually small or unusually large values?
 - A common concern with behavioral tests is that the testing devices can lead to measurement error. It is especially a concern when multiple devices are used, as the devices can give slightly different measurements, even after careful calibration. Create a plot to visualize the relationship between (transformed) freezing to cue and the device used (“FCbox” column). *Hint:* Try a boxplot (`geom_boxplot`). Based on this plot, does the apparatus used affect these behavioral test measurements?

Part B: Exploratory analysis of bone-mineral density data

Now you will examine data on bone-mineral density (BMD) in mice. This is a trait that is important for studying human diseases such as osteoporosis (units are mg/cm^2).

- Plot the distribution of BMD in CFW mice (see column “BMD”). What is most notable about the distribution?
- Compare these data against BMD measurements taken in a “reference” mouse population, the Hybrid Mouse Diversity Panel. To compare, create two histograms, and draw them one on top of the other. What difference do you observe in the BMD distributions? For a correct comparison, you will need to account for: (1) BMD in CFW mice was measured in the femurs

¹R code: `logit <- function(x) log((x + 0.001) / (1 - x + 0.001))`

of male mice only; (2) BMD in HMDP mice was recorded in g/cm². *Hints:* Functions `xlim` and `labs` from the `ggplot2` package, and `plot_grid` from the `cowplot` package, might be useful for creating the plots. The `binwidth` argument in `geom_histogram` may also be useful.

Part C: Mapping the genetic basis of osteopetrotic bones

A binary trait, “abnormal BMD”, was defined that signals whether an individual mouse had “abnormal”, or osteopetrotic, bones. It takes a value of 1 when BMD falls on the “long tail” of the distribution (BMD greater than 90 mg/cm²), otherwise zero.

[GEMMA](#) was used to carry out a “genome-wide association study” (GWAS) for this trait; that is, support for association with abnormal BMD was evaluated at 79,824 genetic variants (single nucleotide polymorphisms, or “SNPs”) on chromosomes 1–19. At each SNP, a *p*-value quantifies the support for an association with abnormal BMD.

1. Your first task is to get an overview of the association results by creating a “Manhattan plot”. Follow as closely as possible the provided prototype, [wtccc.png](#), which shows a genome-wide scan for Crohn’s disease. (Don’t worry about highlighting the strongest *p*-values in green.) *Hints:* Replicating some elements of this plot may be more challenging than others, so start with a simple plot, and try to improve on it. Recall the adage that creating plots requires relatively little effort *provided the data are in the right form*—consider adding appropriate columns to the `gwscan` data frame. Functions from the `ggplot2` package that you may find useful for this exercise include `geom_point`, `scale_color_manual` and `scale_x_continuous`.
 - In your plot, you should observe that the most strongly associated SNPs cluster closely together in small regions of the genome. This is common—it is due to a genetic phenomenon known as linkage disequilibrium (LD). It is a consequence of low recombination rates between markers in small populations. How many SNPs have “strong” statistical support for association with abnormal BMD, specifically with a $-\log_{10} p\text{-value} > 6$? How many distinct regions of the genome are strongly associated with abnormal BMD at this *p*-value threshold?
 - What *p*-value does a $-\log_{10} p\text{-value}$ of 6 correspond to?
 - Using your plot, identify the “distinct region” (this is called a “quantitative trait locus”, or QTL) with the strongest association signal. What is, roughly, the size of the QTL in Megabases (Mb) if we define the QTL by base-pair positions of the SNPs with $-\log_{10} p\text{-value} > 6$? Using the [UCSC Genome Browser](#), get a rough count of the number of genes that are transcribed in this region. Within this QTL, [Parker et al, 2016](#) identified *Col1a1* as a candidate BMD gene. Was this gene one of the genes included in your count? *Hint:* All SNP positions are based on NCBI Mouse Genome Assembly 38 (mm10, December 2011).
2. Your next task is to visualize the relationship between genotype and phenotype. From the genome-wide scan of abnormal BMD, you should find that rs29477109 is the SNP most strongly associated with abnormal BMD. Here you will look closely at the relationship between BMD and the genotype at this SNP. In developing your visualization, consider that:
 - The samples listed in the phenotype and genotype tables are not the same. So you will need to align the two tables to properly show analyze the relationship. *Hint:* Function `match` could be useful for this.
 - The genotypes, stored in file [geno_rs29477109.csv](#), are encoded as “dosages” (numbers

between 0 and 2). You could start with a scatterplot of BMD vs. dosage. But ultimately it is more effective if the genotypes (CC, CT and TT) are plotted instead. *Hints:* In effect, what you need to do is convert from a continuous variable (dosage) to a discrete variable (genotype). One approach is to create a factor column from the “dosage” column. For dosages that are not exactly 0, 1 or 2, you could simply round to the nearest whole number. A boxplot is recommended; see function `geom_boxplot`.

Based on your plot, how would describe (in plain language) the relationship between the genotype and BMD?

Notes

Useful online resources

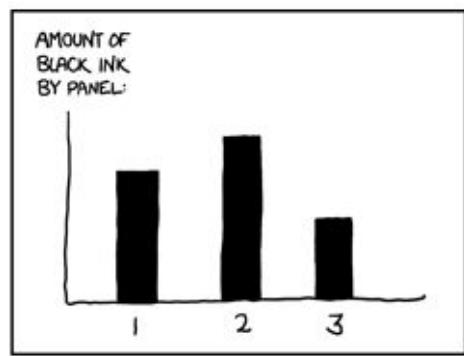
- [ggplot2 reference](#), where you will also find a ggplot2 cheat sheet. (This cheat sheet is also included in the tutorial packet, and you may have seen it in a previous tutorial.)
- [Fundamentals of Data Visualization](#) by Claus Wilke.

License

Except where otherwise noted, all instructional material in this repository is made available under the [Creative Commons Attribution license \(CC BY 4.0\)](#). And, except where otherwise noted, the source code included in this repository are made available under the [OSI](#)-approved [MIT license](#). For more details, see the LICENSE.md file included in the tutorial packet.

Sources

- The CFW phenotype and genotype data were downloaded from the [Data Dryad repository](#).
- The City of Chicago food inspection data were downloaded from the [Chicago Data Portal](#). Specifically, the data were downloaded in CSV format from [here](#) on August 3, 2018.



Reproducibility of data analysis

Stephanie Palmer & Stefano Allesina & Graham Smith

Contents

| | |
|--|-----------|
| Installation notes | 1 |
| git Glossary | 1 |
| Goals | 2 |
| What is reproducibility? | 2 |
| Accessibility | 2 |
| Why use Version Control? | 2 |
| Introduction to git | 3 |
| Remote repositories | 4 |
| Daily Workflow | 4 |
| Clone a Repo | 5 |
| Help using GitKraken and git | 5 |
| Replicability | 5 |
| Understanding the Code | 5 |
| DIY lottery | 7 |
| Simulating noise | 8 |
| Usability | 10 |
| Data Challenge | 10 |
| References and readings | 11 |

Installation notes

For this tutorial, relatively up-to-date versions of R and RStudio are needed. To install R, follow instructions at cran.rstudio.com. Then install Rstudio following the instructions at [goo.gl/a42jYE](https:// goo.gl/a42jYE). Download the ggplot2, cowplot, stats, and RMKdiscrete packages. You will also need [GitKraken](#).

git Glossary

- **branch:** *n.* A lineage of commits. A repository can have multiple branches, and committing changes to one will not affect the others. You can switch between branches (see `checkout`). When you switch branches, all your project's files change to the state they were in at the last commit on that branch. *v.* To create a new branch.
- **checkout:** *v.* Switch branches. Replaces all files with the files from the checked out branch.
- **clone:** *v.* Create a new local copy of a remote repository.
- **commit:** *v.* Take a snapshot of the current progress. *n.* A snapshot of the project at a particular point in time
- **init:** *v.* Create an empty repository.
- **pull:** *v.* Download the latest version of the project from the remote repository.
- **push:** *v.* Upload your latest local commit to the remote repository.
- **repository (repo):** *n.* Entire history of project. Set of all commits on all branches.

- **remote repository**: *n.* Repository hosted on a server (e.g. GitHub)
- **stage**: *v.* Add changes to the set that will be committed. (does not commit!)

Goals

This tutorial will cover methods for making your code reproducible, both by you and by others. Producing results that are reproducible by others is the very essence of science, and writing code that is reproducible by **you** is just the first step. In particular, we will discuss methods to ensure that you can:

- back up versions of your code using `git` and GitKraken
- make your code freely accessible to the wider world via [GitHub](#)
- make your code readable
- reproduce your calculations precisely

Along the way, you will get an introduction to stochastic processes and how they are used to model biological variability. By the end of this tutorial, you should know why it is important to save your seeds and merge your branches. You should also know why reproducible coding practices can help you *now*, even if (indeed, particularly if) you are just learning to code.

What is reproducibility?

Here we'll consider three levels of reproducibility:

0. Accessibility
1. Replicability
2. Reusability
3. Extensibility

These are hierarchically ordered in necessity, difficulty, and goodness.

Reproducible code must be accessible to other researchers. However, accessible code is not necessarily reproducible in any sense. If the code is incomprehensible or doesn't run, then you may as well not have published anything at all.

The first real level of reproducibility is replicability: Does your code run and produce the result you said it would produce? If yes, then at least your result is replicable. If not, then you're just asking your readers to take your results on faith. Indeed, replicability is a bare minimum to qualify as "science."

Most scientists would probably aspire to reusability, the second level of reproducibility: can another scientist take your code and apply it to their own data? In order to apply your code correctly to their own data, another scientist will need to understand your code quite well. Therefore reusability requires more understandable code.

Finally, extensibility requires that your code be reconfigurable to address either different questions or different data. This is fundamentally different from the preceding, and is often beyond the scope of scientific inquiry. If you've written extensible code, you've made a tool, so this is more akin to methods development.

Accessibility

Why use Version Control?

- If you ever collaborate on writing code, then version control is for you.

- If you ever horribly break your project and need to move quickly back in time two days (or two weeks), then version control is for you.
- If you ever need to replicate a figure you made three months ago, version control is for you (I *guarantee* you will need to do this at some point).
- If you ever need to share your code publicly, then version control is for you.
- Most likely, version control is for you.

Version control is useful for small projects, and is essential for large collaborative projects. Without version control, for small projects like manuscripts, often researchers play a game of hot potato. They pass the manuscript back and forth, and the manuscript accumulates an increasingly incomprehensible name (e.g. “Significanceinnovation_draft_2_gs_edits_SEP.docx”, a relatively benign example). You can easily lose track of who has the latest copy, or lose time when one collaborator fails to make promised changes for weeks.

Version control keeps all previously committed versions of the files and directories of your project. This means that it is quite easy to undo short-term changes: Bad day? Just go back to yesterday’s version! You can also access previous stages of the project: “I need to access the manuscript’s version and all the analysis files in exactly the state that they were in when I sent the draft for review three months ago.” Checking out an entire project at a certain point in time is easy with a version control system but much more difficult with Dropbox or Google Drive.

Version control makes it trivial to host your code publicly (e.g. on Github or Bitbucket) and to share a robust link to your code in any publication.

Stefano’s testimonial: “Our laboratory adopted version control for all our projects in 2010, and sometimes we wonder how we managed without it.”

Introduction to git

For this introduction, we will be using `git`, a version control system that is free and is available on all major operating systems. `git` was initially developed by Linus Torvalds (the “Linu” in Linux), exactly for the development of the Linux kernel. It was first released in 2005 and has since become the most widely adopted version control system.

When you start working on a new project, you tell `git` what directory will contain the project. Then `git` takes a snapshot of that directory to create the beginning of your repository. If the project is brand new, this may be a snapshot of an empty directory. After you’ve done some work, you can tell `git` to take a new snapshot, called a **commit**. All snapshots remain available—you can always recover previously committed versions of files.

`git` is especially important for collaborative projects: everybody can simultaneously work on the project, even on the same file. Conflicting changes are reported, and can be managed using side-by-side comparisons. The possibility of **branching** allows you to experiment with changes (e.g. shall we rewrite the introduction of this paper?), and then decide whether to **merge** them into the project. Merging is the chief advantage of `git` over some other version control systems.

Exercise 1: Your First Repo

- 1) Let’s `init` your first repo from scratch! To do so, in GitKraken select `File > Init Repo`. We’ll be creating a “Local Only” repository for this simple example. That just means we’ll not be putting any files on any website (such as GitHub). The files are only stored on your computer. Browse for `Documents` or your equivalent. Note that you cannot put a `git` repository inside another `git` repository.
- 2) Now we have an empty repository. Let’s put something in it. Using a text editor of your choice (Notepad,TextEdit, Sublime, even RStudio), create a file called `origin.txt` with the text “An abstract of an Essay on the Origin of Species...” and save it.

- 3) Now go back to GitKraken. At the top of the center panel, an entry has been added, `// WIP`. This stands for “Work In Progress” and it indicates you’ve made changes to your project that have not been committed to the repository. We have two steps to record the changes to the repository: First, we stage the changes. Second, we commit the staged changes.
- 4) Stage the change.
- 5) Type a commit message (e.g. “Began Essay on the Origin of Species”) and commit the staged change.
- 6) Now let’s get crazy! Create a new file called `end.txt` with the text “Some say the world will end in fire...” AND add onto `origin.txt` so that it says “An abstract of an Essay on the Origin of Species. Darwin’s treatise posits that the process of natural selection passes heritable traits of more advantageous adaptations to subsequent generations. In this essay, we examine...”
- 7) Go back to GitKraken, and `// WIP` should be back. This time when you click on it, you should see two changes are unstaged, one for each file. One is a change, and one is an addition.
- 8) We’ll commit them separately, since they don’t really have anything to do with one another. So first stage the new file `end.txt`, type in a commit message, and commit. You should still see `// WIP` at the top, but underneath should be the commit message you just typed.
- 9) Now click again on `// WIP` and stage the changes to `origin.txt`. Notice when you do so that there are *three* changes to the file: the original first line was deleted, and the new first line was added. That’s just how `git` works. It only does whole-line changes, not within line changes. Of course the second line was simply added.

If you use `git` in the wild, you’ll find that staging and committing is 90% of what you do.

Remote repositories

For the remainder of the tutorial, we’ll be using a more complicated example based in code hosted on [GitHub](#). So far, we have been working with a local repository, meaning the repository is hosted only on your computer. Usually, you will also want to keep a copy of the repository online, called a “remote repository”. With a remote repository, you can collaborate with others, sync your code across multiple machines, and back up your code.

The most popular option for hosting remote repositories is [GitHub](#). As a student you can get private repositories for free! The [Student Developer Pack](#) comes with access to lots of other goodies, but private repositories make it a necessity. They’re useful in the early stages of your project when you’re paranoid about anyone, let alone the entire world, seeing your hasty hacks. But remember to publicize your repository when you do publish!

After the initial setup, you only need to add two new commands to your `git` workflow: `pull` and `push`. When you want to work on a project that is tracking a remote repository, you `pull` the most recent version from the server to sync your local copy of the project to the most recent version. When you are done working, you `push` your commits to the server so that other users can see them.

Daily Workflow

- 1) Pull any new changes from your collaborators
- 2) Work on your project
- 3) When you’ve done something meaningful, stage your changes
- 4) Commit your changes, writing a meaningful commit message
- 5) Repeat steps 2-4

- 6) When you're done for the day, or when you've finished code that you want your collaborators to be able to access, push your changes to the remote repository (e.g. on GitHub).

Clone a Repo

Just as you want to make your code reproducible for other researchers, one day you will want to use another researcher's code. When that time comes, if that researcher has politely hosted their code on GitHub, accessing their code will be as simple as this:

- 1) In GitKraken, `File -> Clone Repo`
- 2) In "Where to clone to" browse to any location on your computer that is NOT within another `git` repo.
For now, I'd recommend your "Documents" folder.
- 3) In your browser, navigate to the repository, click the green button "Clone or download" and copy the URL that appears.
- 4) Paste the URL into the URL field.
- 5) Clone the repo!

Help using GitKraken and git

GitKraken is a graphical user interface (GUI) on top of the older command-line interface (CLI) `git`. If you've never used `git` before, GitKraken has a good tutorial: <https://support.gitkraken.com/start-here/guide>

Replicability

So far, we've covered making your code accessible, whether to yourself in the future (via version control) or to other researchers (via *remote* version control). But accessibility is only the first step. To go further, we need to make sure other researchers can understand our code. You'll learn more about this in defensive programming, but the basics bear repeating.

In order learn how to write understandable code, we're going to try reproducing some figures ourselves.

Understanding the Code

Well-written code is a lot like a well-written essay: it should be understood from the top down.

At the top level, every project should have a file called simply **README**. The README introduces the project as a whole. It should explain the purpose of the project and direct the reader to important files (e.g. the script that runs the analysis). Thus the README is analogous to the essay's introduction.

Then, analogous to the introduction to a section of your essay, each **file** should begin with a few lines of comments explaining what will be found in that file. Additionally, such comments usually include the author's name so future readers know who to blame and who to contact in case of questions.

Similarly, every **block of code** should have a comment describing its purpose. A block of code is simply contiguous lines of code isolated by whitespace. Blocks are like paragraphs, so there isn't any rule as to how to make them. Loosely, if you can write a concise comment describing the action of some lines of code, that would make a good block. The comment describing the block's purpose is analogous to the topic sentence that should appear near the beginning of a paragraph (usually).

Finally, the most specific comments are **in-line comments**. As a beginning programmer, you should probably use in-line comments liberally, to make explicit to yourself what each line does. However, as you gain experience, inline comments should be used sparingly, if at all. There are a few reasons to avoid in-line comments once you gain experience coding:

- 1) If the code in a single line needs a comment to be understandable, then it's too complicated.
- 2) Similarly, if your code is written well, then the comment would be redundant.
- 3) Often, you'll change code but forget to change the comment, leading to misleading in-line comments.

Read the code

While comments are useful and necessary, you should always strive to write code that is understandable by itself. Let's look at an example. You can see a this example drawn out to a painful conclusion in `somocode.R` in the code folder. The below code is *technically* fairly simple, and shouldn't really need any comments to be understandable. But *practically* it's another story. See if you can tease out its purpose, as-is:

```
t0 = 0
a = 35000
r = 0.05
acc = 0
for (i in 1:45) {
  acc = acc * (1 + r) + a
}
print(acc)
```

This code technically “works.” It will run, and it will print a result. But what does the result mean? No idea! Below are some good coding principles that it violates. We'll rewrite the code using these principles and make the code much more understandable.

- **Use good names** Use variable names that are self-explanatory. For example, `a` in the above code would more helpfully be called `annual_income`. Don't worry about using increasingly long (yet descriptive!) names. In RStudio, you can simply type `annual` [TAB] to see a list of all functions you've defined whose name starts with `annual`, so the extra characters don't waste time and can only help you.
- **Do one thing at a time** When you put several things on one line, it can make the line more difficult to parse, much like a run-on sentence. Instead, separate multi-part commands into multiple lines.
- **Don't use magic numbers** This is a special case of “use self-explanatory variable names.” Always name numbers by putting them into variables and using the variable in calculations. For example, the code `area <- 5 * 3` is not as clear as `area <- width * height`. In our code there's only one magic number: 45. We should instead name this `years_to_retirement`.
- **Encapsulate code into functions** Putting code into well named functions is easily the most effective way to write self-documenting, understandable code. You'll understand the code perfectly when I tell you that we should make it a function called `calculate_retirement_savings`.

```
calculate_retirement_savings <- function(annual_income, annual_interest_rate,
                                         years_to_retirement, current_savings=0) {
  for (i in 1:years_to_retirement) {
    current_savings = current_savings * (1 + annual_interest_rate)
    current_savings = current_savings + annual_income
  }
  return(current_savings)
}
calculate_retirement_savings(annual_income=35000, annual_interest_rate=0.05,
                            years_to_retirement=45, current_savings=0)
```

We can understand this code much more readily. The code calculates retirement savings after 45 years, making some assumptions about savings and interest rates along the way.

This code is so well written, you might even notice a problem with the calculation! Most grad students can't save their entire annual income. Good programming like this is defensive because you can more easily notice problems like this. Good programming improves reproducibility because now anyone can understand the purpose, capabilities, and limitations of our code. When you re-run old code, you need to understand what's

happening. If I give you a function called `calculate_retirement_savings`, you have a pretty good idea of the purpose, but that's it. Does it assume a constant interest rate? Constant income? Constant expenses? Are taxes calculated? Does the function take advantage of Roth or Traditional IRAs? Does it know to hide excessive returns in a secret account in the Caymans? These are all important considerations in planning your retirement, and you'll only know what my function does by actually reading the code, so you can only *reuse* it if you can understand it. And if you want to *extend* it, then understanding it is just the first step.

DIY lottery

Now let's try some exercises that combine what we've covered so far. In this section, we'll clone a repository containing some bad code, we'll fix the code, and then we'll try to understand the code well enough to analyse the results.

Follow the instructions in “Clone a Repo” to clone <https://github.com/grahamas/BentCoinLottery>

Before you try reading the code, we should tell you that the code uses one of R's random number generator functions `rnorm`, which will give you uniformly distributed numbers on the interval $[0, 1]$.

Exercise 2: Use meaningful variables

Go through the lottery simulation, making all variable names meaningful, both by changing variable names and by introducing new variables to get rid of magic numbers.

Probability interlude

No matter how well documented code may be, you'll still need some foundational understanding of the concepts being implemented, so that you can then understand the implementation itself. So let's pause and go over probability, which is key to understanding our lottery.

Each flip of a coin with probability, p , of heads is an example of a *Bernoulli trial*, which is the general term for an experiment with only two output states, success or failure. The number of heads in the sequence of independent coin flips generated by our lottery will follow a *binomial distribution*, which extends the Bernoulli trial to many flips.

$$P_n(k) = \binom{n}{k} p^k (1-p)^{n-k},$$

where p is the probability of heads (1's), n is the length of our lottery ticket, and k is the number of heads in the ticket. The prefactor $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is called the binomial coefficient and describes the number of unique ways of placing k identical objects in n bins. Capital “ $P_n(k)$ ” represents the probability distribution of k heads out of n tosses. We're calling this a “bent coin lottery” because the probability of heads is not $1/2$, it's some other number. The coin we are simulating flipping isn't a fair coin, perhaps it's been subtly bent so that it mostly lands tails up.

Knowing this, you can use the function `rbinom` to generate draws from a binomial distribution. In our lottery, this would amount to flipping the coin `n_flips` times.

```
n_flips <- 3
p_heads <- 0.1
rbinom(n_flips, 1, p_heads)

# [1] 0 0 0
```

Exercise 3: Use functions

Rewrite the lottery to use `rbinom`. In case you decide to change the ticket generator in the future, abstract the ticket generation into a function called `generate_ticket`. What arguments should this function accept? What should it return? Be sure to note both answers in a comment at the

beginning of the function. Does using this meaningfully-named function make the code more readable? What about using `rbinom`?

Simulating noise

For the remainder of the exercises, we'll need a bit of extra background. First we'll discuss "seeds," which are essential to making your stochastic code reproducible. For example, seeds will make it possible to draw the exact same "random" lottery tickets as your neighbor. Second we'll go into a bit of the math underlying our lottery tickets (the binomial distribution) so that we can do some analyses of our generated data.

Save your seeds

Compare a draw from your lottery with your neighbor. Do you draw the same sequence of random lottery tickets? Why not? If you wanted to reproduce the *exact* same output from your lottery each time you decide to reset it, you'll need to know a little more about how R's random number generator (RNG) works. Try typing:

```
? RNG
```

That should open documentation in the "Help" pane. You will notice that the function `RNGkind` is the interface for querying the current state of the RNG. Let's find out what the current settings are:

```
RNGkind()
```

```
# [1] "Mersenne-Twister" "Inversion"
```

The first part is the RNG algorithm, the second specifies the algorithm for transforming uniformly distributed random numbers into random samples from the normal, or Gaussian, distribution. The twister algorithm based on Mersenne prime numbers, $M_n = 2^n - 1$, where n is also prime, is a state-of-the-art pseudo-random number generation scheme, developed by Matsumoto and Nishimura in 1997. NB: RNG's should technically be called *pseudo*-random number generators or PRNG's, in part because they all have some period after which they will produce exactly the same sequence. The trick is to find an algorithm with a period so long you'll never notice the "P" in the "PRNG". The Mersenne Twister algorithm has a period of $2^{19937} - 1$ and passes many statistical tests for randomness.

The seed to an RNG is usually a large integer that provides an initialization the RNG algorithm. Scrolling down to "Note" in the "Help" pane, you'll learn that the seed to R's RNG is set by the current time and the process ID. That means that your simulation results will depend on when you start your R session, run your code, and even some local information in your processing environment. Compare the output of `runif` with your neighbor.

```
runif(5)
```

Starting an RNG with the same seed will produce exactly the same sequence of random numbers; an RNG spits out random numbers, but not noisy ones. To reproduce your simulation results precisely when you use an RNG, you'll want control of that seed. R uses `set.seed` which takes a small integer as input and generates, deterministically, all the random seeds necessary for your RNG algorithm.

```
set.seed(19937)
runif(5)
runif(5)
set.seed(19937)
runif(5)
```

Exercise 4: Playing with seeds

- 1) Test whether or not you and your neighbor get precisely the same sequence of numbers when you use the same seed.
- 2) Will this work if you use different RNG algorithms?
- 3) Try changing your RNG algorithm using `RNGkind` and compare your results with your neighbor, when you use the same seed.
- 4) If you wanted to be able to instruct someone to reproduce your exact simulation results using R's RNG, what would you need to tell them?

You can use `RNGkind` to set or query both the RNG and normal algorithms. You can save this information along with the current value of `seed` using something like:

```
seed <- 19937
set.seed(seed)
seed_used <- seed
RNGkind_used <- RNGkind()
save("seed_used", "RNGkind_used", file=RNGinfo_for_mycode)
```

When you are ready to share your code with others, you should also save all the version information for your current R package and libraries to this same file.

Noisy processes

Many of the variables that we observe in biological recordings fluctuate, sometimes because we cannot control all the states of the external and internal experimental system, other times because thermal noise makes the state of the biological system we interrogate inherently variable. Examples of fluctuating quantities in biological systems include: the number of a certain type of molecule in a cell; the number of open channels in a cell; the number of electrical action potentials or “spikes” emitted by a neuron in response to a stimulus; the number of individuals in a population at a particular moment in time; the number of bacterial colonies on a plate. These are all quantities that we can make precise claims about, on average, but cannot specify with certainty for any particular experimental observation.

It is useful to model not only a mean value for a fluctuating variable, but the full shape of its distribution of values. For example, if we observe the firing of neurons in the brain to repeats of the same external stimulus, the precise times of spikes will vary between repeats. By fitting the statistics of this noise to models, we deepen our mechanistic understanding of the neural response. We can test whether or not the “noise” we observe is consistent with a truly random source of output variation, or if it has some structure that tells us about interactions between the biological components and their environment.

Often, noise in biological systems is modeled by what's called a Poisson process, whose values follow a Poisson distribution. The function you wrote to sample the bent coin lottery generated tickets whose statistics follow the binomial distribution. The binomial distribution approaches the familiar Poisson distribution, in the limit of a large number of trials, n , or a small probability of the event, p , per trial:

$$P_n(k)_{n \rightarrow \infty} = \frac{\lambda^k}{k!} e^{-\lambda}$$

where λ is the average rate of occurrence of our event in n trials. We have just written down the Poisson distribution. You will see this used as a model for biological variability again and again, either explicitly or implicitly. It is important to think about whether or not it is a good model for the system under study each time you come across it or are deciding to use it for your own research. Notes on deriving the relationships between some common distributions are provided in the readings folder.

Exercise 5: Replicate a figure

Now using our knowledge of seeds and Poisson distributions, we'll try to replicate a figure created in the analysis of the lottery.

- 1) checkout the branch called “analysis.”
- 2) Find the code necessary to replicate Figure 1 (hint: when you switched branches, the README was updated)
- 3) Replicate Figure 1 (hint at the bottom of “References and Readings”)

Usability

For our final exercise, we’ll try to make a figure of our own. From Exercise 5, you should be on the “analysis” branch.

Exercise 6: Make a new figure

- 1) When is the distribution modeled in Exercise 5 approximately normally distributed?
- 2) Make a figure analogous to Figure 1, but with newly generated data nicely approximated by a normal distribution.
- 3) Make sure you comment your code and include all the information necessary to reproduce your figure.

Data Challenge

For the data challenge, you’ll be analysing real data from research papers found in the `readings` folder (the data is found in the `data` folder).

Arthropod dispersion

In 1941 and 1942, zoologist LaMont Cole, then working at the University of Chicago, set out to survey species diversity and distribution in the woods and pastures of Kendall County, Illinois. He was particularly interested in which species co-occurred in woods versus grazing land and how their numbers varied with changes in humidity and temperature throughout the year. He laid thick oak boards in a variety of locations and counted the number of “cryptozoic” (animals found under stones, rotten logs, tree bark, etc.) individuals found under the boards several times a week, over the course of a year. For his spatial distribution studies, he aimed to determine whether arthropods distributed themselves randomly or if they had a more complex interaction pattern with each other or with their environment.

If the bugs are randomly distributed, then the counts of the bugs under the boards should be Poisson distributed. If there is some pattern, we might try to fit them by a Lagrangian Poisson Distribution or LGP, which can model the attraction or repulsion of individuals to each other. The LGP has a mean rate parameter, λ_1 , just like in the Poisson distribution. The LGP has a second parameter, λ_2 , that describes the deviation from expected Poisson dispersion.

NB: A Poisson distribution has a variance-to-mean ratio equal to 1. Show that the variance of the spider count distribution is approximately equal to its mean.

$\lambda_2 > 0$ implies that organisms are over-dispersed (i.e. they have a variance greater than their mean number) and may be attracted to one another, forming more large-number clusters than expected by chance. $\lambda_2 < 0$ implies that organisms are under-dispersed and are likely repulsed by one another, resulting in a more even distribution across space and, hence, lower count variance. The LGP distribution, in terms of these two parameters, is

$$P(k) = \lambda_1(\lambda_1 + k\lambda_2)^{k-1} \frac{e^{-(\lambda_1+k\lambda_2)}}{k!}.$$

When $\lambda_2 = 0$, we get precisely the Poisson distribution.

The challenge

Check visually how the distributions of the three bug types (for weevils, its their eggs and they lay their eggs inside beans, so it's eggs per bean instead of bugs per board, but you will plot and fit the data in the same way) fit to a Poisson or the LGP distributions.

- 1) Plot the Poisson distribution with the same mean as the spider counts, along with the data
- 2) Plot the Poisson distribution with the same mean as the sowbug counts, along with the data
- 3) Plot the Poisson distribution with the same mean as the weevil egg counts, along with the data
- 4) Add a curve to Plot 1) showing the LGP distribution with the parameter hint below for the spider counts
- 5) Add a curve to Plot 2) showing the LGP distribution with the parameter hint below for the sowbug counts
- 6) Add a curve to Plot 3) showing the LGP distribution with the parameter hint below for the weevil egg counts

Each figure should have datapoints corresponding to empirical data, and lines corresponding to theoretical distribution. Indicate in the README how well the each distribution fits these data sets, as well as the chosen λ_2 .

For LGP, you can use the function `dLGP` from the library `RMKdiscrete`.

For spiders, you can set λ_2 to 0 and λ_1 to the mean.

For sowbugs, you can use the empirically determined $\lambda_2 = 0.53214$ and estimate λ_1 by the formula $\lambda_1 = \text{mean} * (1 - \lambda_2)$.

For weevil eggs, look at the data to see if you can gauge their sociability (i.e. do weevils lay their eggs all together or do they seem to intentionally separate them?) and thereby start guessing appropriate λ_2 .

The requirements

- 1) Create a **private** repository on GitHub.
 - On the GitHub homepage, select “New Repository” and then on the next page select “Private”
- 2) Add all members of your group as collaborators, plus Graham (username: grahamas) and Stephanie (username: sepalmer).
 - On your repository’s GitHub page, select “Settings -> Collaborators” then add by GitHub username
- 3) Every member of your group must make at least one substantive commit.
- 4) Follow the commenting guidelines outlined above.
- 5) Try to write functions, rather than copying and pasting code.

References and readings

Journal articles

All of the data used in this tutorial come from original research papers that are in the `readings` folder. Also in the `readings` folder, you will find an article by Roger Peng arguing for setting standards for reproducible research in computational science. It’s a short article that we hope you will read and adopt as best practices for your own work.

Books and tutorials

There are very many good books and tutorials on `git`. We are particularly fond of *Pro Git*, by Scott Chacon and Ben Straub. You can either buy a physical copy of the book, or read it online for free.

Both [GitHub](#) and [Atlassian](#) (managing Bitbucket) have their own tutorials.

A great way to try out Git in 15 minutes is [here](#).

[Software Carpentry](#) offers intensive on-site workshops and online tutorials.

Hint to Exercise 5

- You'll need to go through the commit history.

Statistics for a data rich world—some explorations*

Stefano Allesina University of Chicago

- **Goal:** More and more often we need to analyze large and complex data sets. However, the statistical methods we've been taught in college have evolved in a data-poor world. Modern biology requires new tools, which can cope with the new questions and methods that arise in a data-rich world. Here we are going to discuss problems that often arise in the analysis of large data sets. We're going to review hypothesis testing (and what happens when we have many hypotheses) and discuss model selection. We're going to see the effects of selective reporting and p-hacking, and how they contribute to the *reproducibility crisis* in the sciences.
 - **Audience:** Biologists with some programming background.
-

Review of hypothesis testing

The basic idea of hypothesis testing is the following: we have devised an hypothesis on our system, which we call H_1 (the “alternative hypothesis”). We have collected our data, and we would like to test whether the data are consistent (or not) with the so-called “null-hypothesis” (H_0), which is associated with a contradiction to the hypothesis we would like to prove.

The simplest example is that of a bent coin: we believe that our coin favors heads (H_1 : the coin is bent). We therefore toss the coin several times and check whether the number of heads we observe is consistent with the null hypothesis of a fair coin (H_0).

In R we can toss many coins in no time at all. Call p the probability of obtaining a head, and initially toss a fair coin ($p = 0.5$) a thousand times:

```
p <- 0.5 # probability of a head (fair coin)
flips <- 1000 # number of times we flip the coin
data <- sample(c("H", "T"),
               flips, prob = c(p, 1 - p),
               replace = TRUE)
heads <- sum(data == "H")
```

If the coin is fair, we expect approximately 500 heads, but of course we might have small variations due to the randomness of the process. We therefore need a way to distinguish between “bad luck” and an incorrect hypothesis.

What is customarily done is to compute the probability of recovering the observed or a more extreme version of the pattern under the null hypothesis: if the probability is very small, we reject the null hypothesis (note that this does not guarantee that the alternative hypothesis is true). We call this probability a *p-value*.

For example, if the coin is fair, the number of heads should follow the binomial distribution. The probability of observing a larger number of heads than what we've got is therefore

*This document is included as part of the Statistics for large data sets packet for the U Chicago BSD qBio5 boot camp, @MBL, 2019. Current version: July 31, 2019; Corresponding author: sallesina@uchicago.edu.

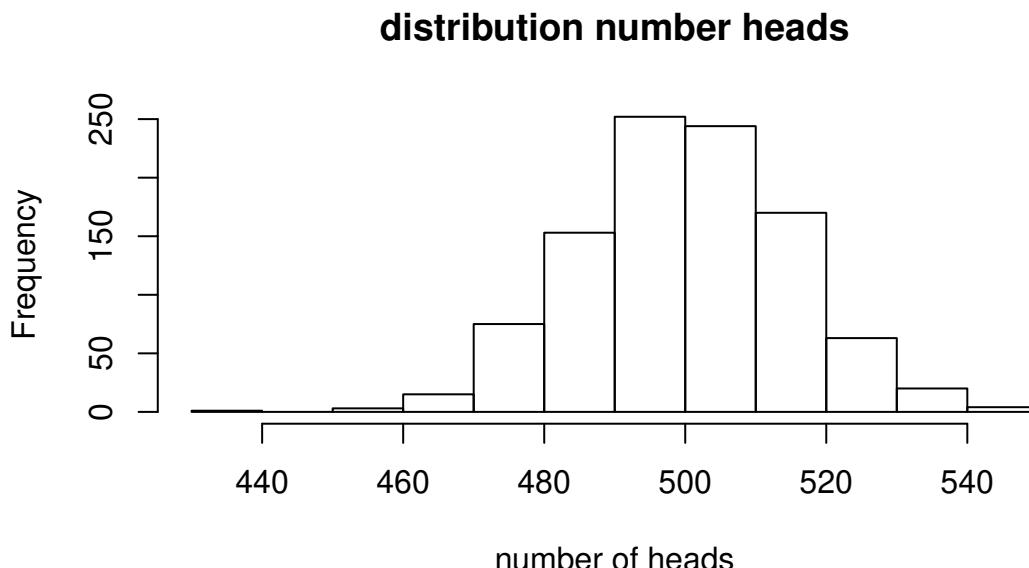
```
pvalue <- 1 - pbinom(heads, flips, 0.5)
```

which also suggests a faster way to flip the coins and count the heads:

```
heads <- rbinom(1, flips, p)
pvalue <- 1 - pbinom(heads, flips, 0.5)
```

What if we repeat the tossing many, many times?

```
# flip 1000 coins 1000 times
# produce histogram of number of heads
heads_distribution <- rbinom(1000, flips, p)
hist(heads_distribution, main = "distribution number heads", xlab = "number of heads")
```



You can see that it is very unlikely to get more than 560 (or less than 440) heads when flipping a fair coin 1000 times. Therefore, if we were to observe say 400 heads (or 800), we would tend to believe that the coin is biased (though of course this could have happened by chance!).

Errors

When testing an hypothesis, we can make two types of errors:

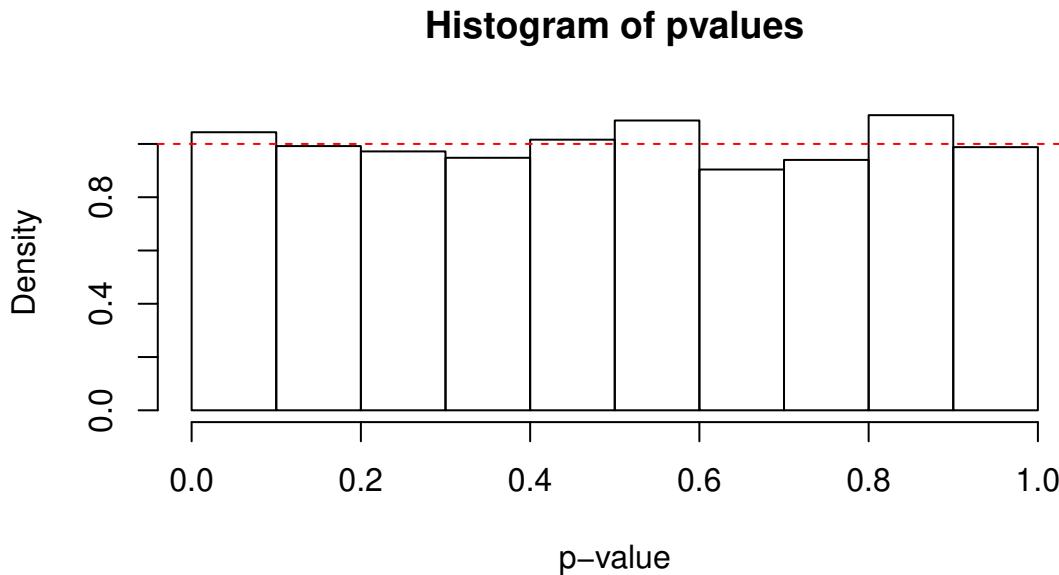
- **Type I error:** reject H_0 when it is in fact true
- **Type II error:** fail to reject H_0 when in fact it is not true

We call α the probability of making an error of type I, and β that of making a type II error. In practice, we tend to choose a quite stringent α (say, 0.01, 0.05), and then try to control for β as best as we can. What is typically reported is the smallest level of significance leading to the rejection of the null hypothesis (p-value). Therefore, the p-value quantifies how strongly the data contradicts the null hypothesis.

The distribution of p-values

Suppose that we are tossing each of several fair coins 1000 times. For each, we compute the corresponding p-value under the hypothesis $p = 0.5$. How are the p-values distributed?

```
ncoins <- 2500  
heads <- rbinom(ncoins, flips, p)  
pvalues <- 1 - pbinom(heads, flips, 0.5)  
hist(pvalues, xlab = "p-value", freq = FALSE)  
abline(h = 1, col = "red", lty = 2)
```

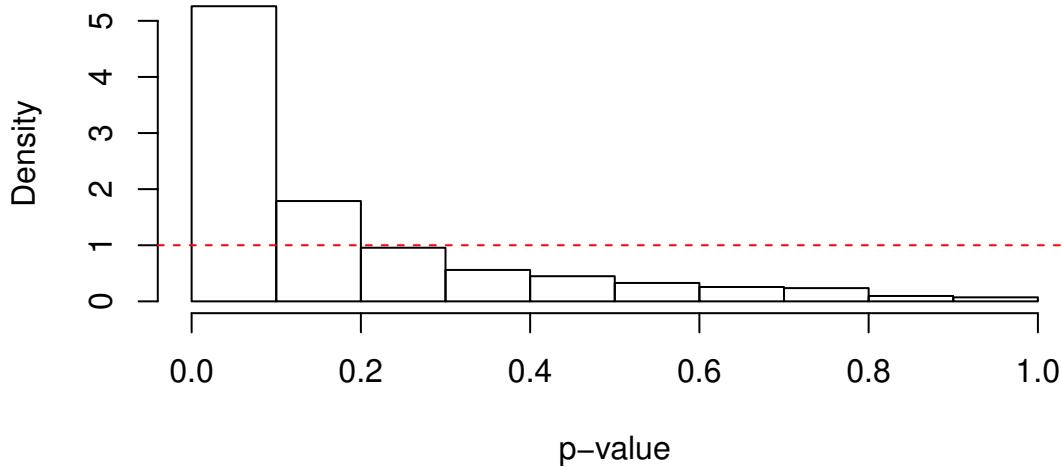


As you can see, if the data were generated under the null hypothesis, the distribution of the p-values would be approximately uniform between 0 and 1. This means that if we set $\alpha = 0.05$, we would reject the null hypothesis 5% of the time (even though in this case we know the hypothesis is correct!).

What is the distribution of the p-values if we are tossing biased coins? We will find an enrichment in small p-values, with stronger effects for larger biases:

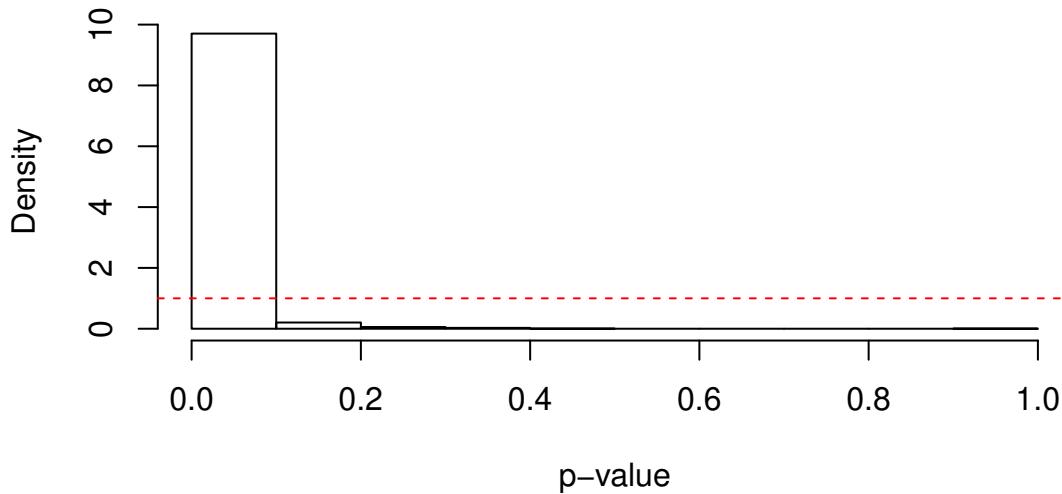
```
p <- 0.52 # the coin is biased  
heads <- rbinom(ncoins, flips, p)  
pvalues <- 1 - pbinom(heads, flips, 0.5)  
hist(pvalues, xlab = "p-value", main = paste0("p = ", p), freq = FALSE)  
abline(h = 1, col = "red", lty = 2)
```

p = 0.52



```
p <- 0.55 # the coin is biased
heads <- rbinom(ncoins, flips, p)
pvalues <- 1 - pbineom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", main = paste0("p = ", p), freq = FALSE)
abline(h = 1, col = "red", lty = 2)
```

p = 0.55



Problem: selective reporting

Articles reporting positive results are easier to publish than those containing negative results. Authors might have little incentive to publish negative results, which could go directly into the file-drawer.

This tendency is evidenced in the distribution of p-values in the literature: in many disciplines, one finds a sharp decrease in the number of tests with p-values just below 0.05 (which is

customarily—and arbitrarily—chosen as a threshold for “significant results”). For example, we find many a sharp decrease in the number of reported p-values of 0.051 compared to 0.049—while we expect the p-value distribution to decrease smoothly.

Selective reporting leads to irreproducible results: we always have a (small) probability of finding a “positive” result by chance alone. For example, suppose we toss a fair coin many times, until we find a “significant” result...

Problem: p-hacking

The problem is well-described by Simonsohn et al. (*J. Experimental Psychology*, 2014): “While collecting and analyzing data, researchers have many decisions to make, including whether to collect more data, which outliers to exclude, which measure(s) to analyze, which covariates to use, and so on. If these decisions are not made in advance but rather are made as the data are being analyzed, then researchers may make them in ways that self-servingly increase their odds of publishing. Thus, rather than placing entire studies in the file-drawer, researchers may file merely the subsets of analyses that produce nonsignificant results. We refer to such behavior as *p-hacking*.”

The same authors showed that with careful p-hacking, almost anything can become significant (read their hilarious article in [Psychological Science](#), where they show that listening to a song can change the listeners’ age!).

Discussion on p-values

Selective reporting and p-hacking are only two of the problems associated with the widespread use and misuse of p-values. The discussion in the scientific community on this issue is extremely topical. I have collected some of the articles on this problem in the [readings](#) folder. Importantly, in 2016 the American Statistical Association released a statement on p-values every scientist should read.

Reproducibility crisis

P-values and hypothesis testing contribute considerably to the so-called *reproducibility crisis* in the sciences. A survey promoted by *Nature* magazine found that “More than 70% of researchers have tried and failed to reproduce another scientist’s experiments, and more than half have failed to reproduce their own experiments.”

This problem is due to a number of factors, and addressing it will likely be one of the main goals of science in the next decade.

Exercise: p-hacking

Go to goo.gl/a3UOEF and try your hand at p-hacking, showing that your favorite party is good (bad) for the economy.

Multiple comparisons

The problem of multiple comparisons arises when we perform multiple statistical tests, each of which can (in principle) produce a significant result.

Suppose we perform our coin tossing exercise, flipping 1000 coins 1000 times each. For each coin, we determine whether our data differs significantly from what expected by contrasting our p-value with a significance level $\alpha = 0.05$.

Even if the coins are all perfectly fair, we would expect to find approximately $0.05 \cdot 1000 = 50$ coins that lead to the rejection of the null hypothesis.

In fact, we can calculate the probability of making at least one error of type I (reject the null when in fact it is true). This probability is called the Family-Wise Error Rate (FWER). It can be computed as 1 minus the probability of making no type I error at all. If we set $\alpha = 0.05$, and assume the tests to be independent, the probability of making no errors in m tests is $1 - (1 - 0.05)^m$. Therefore, if we perform 10 tests, we have about 40% probability of making at least a mistake; if we perform 100 tests, the probability grows to more than 99%. If the tests are not independent, we can still say that in general $FWER \leq m\alpha$.

This means that setting an α per test does not control for FWER.

Moving from tossing coins to biology, consider the following examples:

- **Gene expression** In a typical microarray experiment, we contrast the differential expression of tens of thousands of genes in treatment and control tissues.
- **GWAS** In Genomewide Association Studies we want to find SNPs associated with a given phenotype. It is common to test tens of thousands or even millions of SNPs for significant associations.
- **Identifying binding sites** Identifying candidate binding sites for a transcriptional regulator requires scanning the whole genome, yielding tens of millions of tests.

Organizing the tests in a table

Suppose that we're testing m hypotheses. Of these, an unknown subset m_0 is true, while the remaining $m_1 = m - m_0$ are false. We would like to correctly call the true/false hypotheses (as much as possible). We can summarize the results of our tests in a table, of which the elements are unobservable:

| | not rejected | rejected |
|------------|---------------------|---------------------|
| H is True | U (true negatives) | V (false positives) |
| H is False | T (false negatives) | S (true positives) |

What we would like to know is $m_1 = T + S$ and $m_0 = U + V$. Then V is the number of type I errors (rejected H when in fact it is true), and T is the number of type II errors (failed to reject a false H). However, we can only observe $V + S$ (the number of “discoveries”), and $U + T$ (number of “failures”).

Our Per-Comparison Error Rate is $PCER = E[V]/m$ (where $E[X]$ stands for expectation), our Family-wise error rate is $P(V > 0)$. One quantity of interest is the False Discovery Rate (FDR),

measured as the proportion of true discoveries $FDR = E[V/(V + S)]$ when $V + S > 0$. FDR measures the proportion of falsely rejected hypotheses.

Importantly $PCER \leq FDR \leq FWER$, meaning that when we control for FWER we're automatically controlling for the others, but not viceversa.

Bonferroni correction

One of the simplest and most widespread procedures to control for FWER is Bonferroni's correction. This procedure controls for FWER in the case of independent or dependent tests. It is typically quite conservative, especially when the tests are not independent (in practice, it becomes "too conservative" when the number of tests is moderate to high). Fundamentally, for a desired FWER α we choose as a (PCER) significance threshold α/m , where m is the number of tests we're performing. Equivalently, we can "adjust" the p-values as $q_i = \min(m \cdot p_i, 1)$, and call significant the values $q_i < \alpha$. In R it is easy to perform this correction:

```
original_pvals <- c(0.012, 0.06, 0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "bonferroni")
print(adjusted_pvals)
```

```
## [1] 0.060 0.300 1.000 0.005 1.000
```

With these adjusted p-values, and an $\alpha = 0.05$, we would still single out as significant the fourth test, but not the first. The strength of Bonferroni is its simplicity, and the fact that we can perform the operation in a single step. Moreover, the order of the tests does not matter.

Other procedures

There are several refinements of Bonferroni's correction, some of which use the sequence of ordered p-values. For example, Holm's procedure starts by sorting the p-values in increasing order $p_{(1)} \leq p_{(2)} \leq p_{(3)} \leq \dots \leq p_{(m)}$. The hypothesis $H_{(i)}$ is rejected if $p_{(j)} \leq \alpha/(m - j + 1)$ for all $j = 1, \dots, i$. Equivalently, we can adjust the p-values as $q_{(i)} = \min(1, \max((m - i + 1)p_{(i)}, q_{(i-1)}))$. In this way, we use the most stringent threshold to determine whether the smallest p-value is significant, the next smallest p-value uses a slightly higher threshold and so on. For example, using the same p-values above:

```
original_pvals <- c(0.012, 0.06, 0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "holm")
print(adjusted_pvals)
```

```
## [1] 0.048 0.180 0.770 0.005 0.640
```

We see that we would be calling the first test significant, contrary to what obtained with Bonferroni.

The function `p.adjust` offers several choices for p-value correction. Also, the package `multcomp` provides a quite comprehensive set of functions for multiple hypothesis testing.

Mix of coins

We're going to test these concepts by tossing repeatedly many coins. In particular, we're going to toss 1000 times 50 biased coins ($p = 0.55$) and 950 fair coins ($p = 0.5$). For each coin, we're going to compute a p-value, and count the number of type I, type II, etc. errors when using unadjusted p-values as well as when correcting using the Bonferroni or Holm procedure.

```
toss_coins <- function(p, flips){  
  # toss a coin with probability p of landing on head several times  
  # return a data frame with p, number of heads, pval and  
  # H0 = TRUE if p = 0.5 and FALSE otherwise  
  heads <- rbinom(1, flips, p)  
  pvalue <- 1 - pbinom(heads, flips, 0.5)  
  if (p == 0.5){  
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = TRUE))  
  } else {  
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = FALSE))  
  }  
}  
  
# To ensure everybody gets the same results, we're setting the seed  
set.seed(8)  
data <- data.frame()  
# the biased coins  
for (i in 1:50) data <- rbind(data, toss_coins(0.55, 1000))  
# the fair coins  
for (i in 1:950) data <- rbind(data, toss_coins(0.5, 1000))  
# here's the data structure  
head(data)  
  
##      p   heads      pval     H0  
## 1 0.55    535 1.235282e-02 FALSE  
## 2 0.55    558 1.061983e-04 FALSE  
## 3 0.55    567 9.546428e-06 FALSE  
## 4 0.55    532 1.988964e-02 FALSE  
## 5 0.55    547 1.322765e-03 FALSE  
## 6 0.55    574 1.178147e-06 FALSE
```

Now we write a function that adjusts the p-values and builds the table above

```
get_table <- function(data, adjust, alpha = 0.05){  
  # produce a table counting U, V, T and S  
  # after adjusting p-values for multiple comparisons  
  data$pval.adj <- p.adjust(data$pval, method = adjust)  
  data$reject <- FALSE  
  data$reject[data$pval.adj < alpha] <- TRUE  
  return(table(data[,c("reject", "H0")]))  
}
```

First, let's see what happens if we don't adjust the p-values:

```
no_adjustment <- get_table(data, adjust = "none", 0.05)
print(no_adjustment)
```

```
##          H0
## reject  FALSE TRUE
##   FALSE      2  905
##   TRUE       48   45
```

We correctly declared 48 of the biased coins “significant”, but we also incorrectly called 2 biased coins “not significant” (Type II error). More worryingly, we called 45 fair coins biased when they were not (Type I error). To control for the family-wise error rate, we can correct using Bonferroni:

```
bonferroni <- get_table(data, adjust = "bonferroni", 0.05)
print(bonferroni)
```

```
##          H0
## reject  FALSE TRUE
##   FALSE     40  950
##   TRUE      10    0
```

With this correction, we dramatically reduced the number of Type I errors (from 45 to 0), but at the cost of increasing Type II errors (from 2 to 40). In this way, we would make only 10 discoveries instead of 50.

In this case, Holm's procedure does not help:

```
holm <- get_table(data, adjust = "holm", 0.05)
print(holm)
```

```
##          H0
## reject  FALSE TRUE
##   FALSE     40  950
##   TRUE      10    0
```

More sophisticated methods, for example based on controlling FDR, can reduce the Type II errors, at the cost of a few Type I errors:

```
BH <- get_table(data, adjust = "BH", 0.05)
print(BH)
```

```
##          H0
## reject  FALSE TRUE
##   FALSE     17  946
##   TRUE      33    4
```

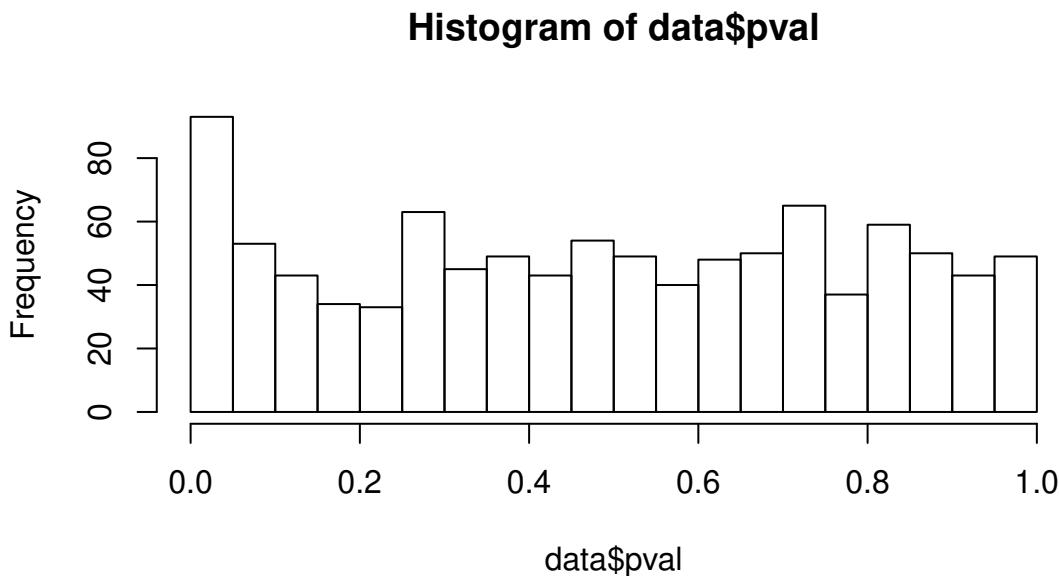
q-values

Inspired by the need for controlling for FDR in genomics, Storey and Tibshirani (PNAS 2003) have proposed the idea of a q-value, measuring the probability that a feature that we deemed significant turns out to be not significant after all.

One uses p-values to control for the false positive rate: when determining significant p-values we control for the rate at which null features in the data are called significant. The False Discovery Rate, on the other hand, measures the rate at which results that are deemed significant are truly null. While setting $PCER = 0.05$ we are stating that about 5% of the truly null features will be called significant, an $FDR = 0.05$ means that among the features that are called significant, about 5% will turn out to be null.

They proposed a method that uses the ensemble of p-values to determine the approximate (local) FDR. The idea is simple. If you plot your histogram of p-values when you have few true effect, and many nulls, you will see something like:

```
hist(data$pval, breaks = 25)
```



where the right side of the histogram is close to a uniform distribution. We could use the high p-values to find how tall the histogram would be if all effects were null, thereby estimating the proportion of truly null features $\pi_0 = m_0/m$.

Storey has built an R-package for this type of analysis:

```
# To install:  
#install.packages("devtools")  
#library("devtools")  
#install_github("jdstorey/qvalue")  
library("qvalue")  
  
## Registered S3 methods overwritten by 'ggplot2':  
##   method           from
```

```

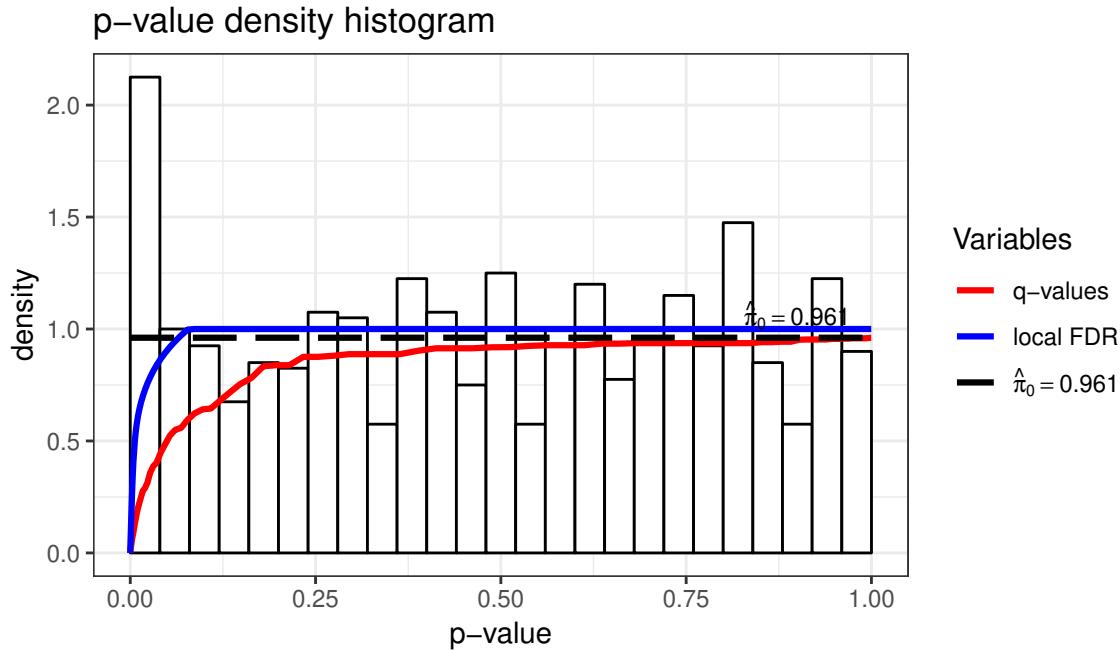
##  [.quosures     rlang
##  c.quosures     rlang
##  print.quosures rlang

qobj <- qvalue(p = data$pval)

```

Here's the estimation of the π_0

```
hist(qobj)
```



which is quite good (in this case we know that $\pi_0 = 0.95$). The small p-values under the dashed line represent our false discoveries. Even better, through randomizations one can associate a q-value to each test, representing the probability of making a mistake when calling a result significant (formally, the q-value is the minimum FDR that can be attained when calling that test significant).

For example:

```
table((qobj$pvalues < 0.05) & (qobj$qvalues < 0.05), data$H0)
```

```

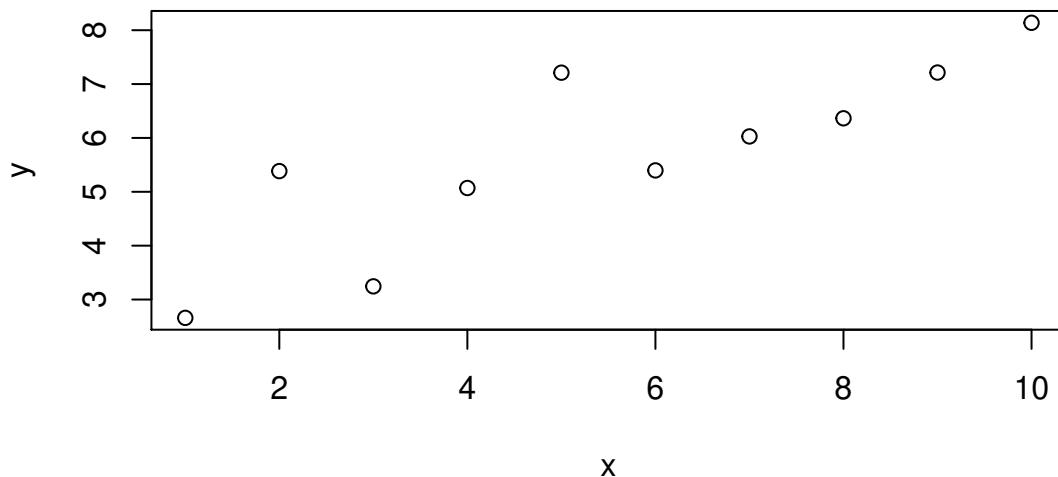
##
##          FALSE  TRUE
##  FALSE      17   946
##  TRUE       33     4

```

Model selection

Often, we need to select a model out of a set of reasonable alternatives. However, we run the risk of overfitting the data (i.e., fitting the noise as well as the pattern). The simplest example is that of a regression:

```
# create fake data
set.seed(5)
x <- 1:10
y <- 3 + 0.5 * x + rnorm(10)
plot(y ~ x)
```



We can fit a linear regression to the data

```
model1 <- lm(y ~ x)
```

Or a more complex polynomial

```
model2 <- lm(y ~ poly(x, 7))
```

Let's see the residuals etc.

```
summary(model1)
```

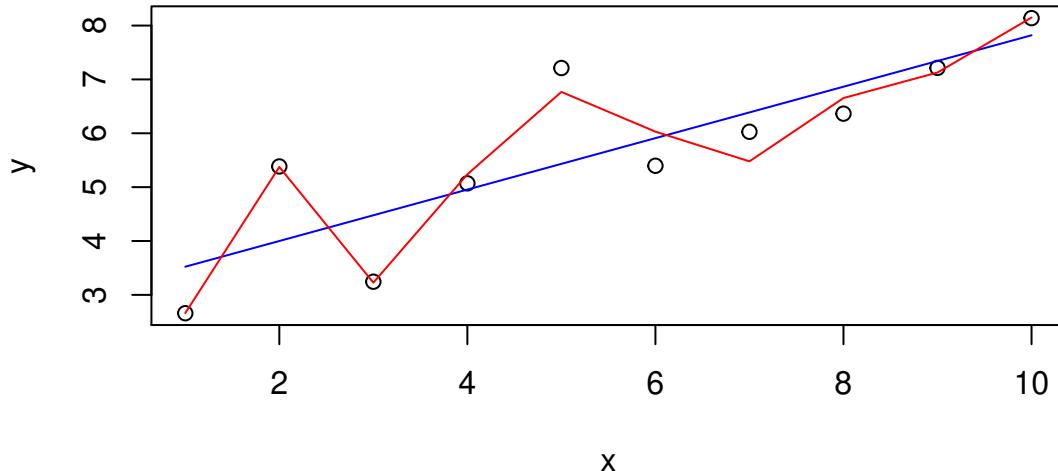
```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min    1Q Median    3Q   Max 
## -1.2330 -0.5096 -0.2437  0.2676  1.7790 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 3.0452     0.6883   4.425  0.00221 ***
## x           0.4774     0.1109   4.304  0.00260 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```

## 
## Residual standard error: 1.007 on 8 degrees of freedom
## Multiple R-squared:  0.6984, Adjusted R-squared:  0.6607
## F-statistic: 18.53 on 1 and 8 DF,  p-value: 0.0026

#summary(model2)
plot(y~x)
points(model1$fitted.values~x, type = "l", col = "blue")
points(model2$fitted.values~x, type = "l", col = "red")

```



Our second model has a much greater R^2 , but also many more parameters. Is the first model more parsimonious?

Model selection tries to address this and similar problems. Most model fitting and model selection procedures are based on likelihoods (e.g., Bayesian models, maximum likelihood, minimum description length). The likelihood $L(\theta|D, M)$ is (proportional to) the probability of observing the data D under the model M and parameters θ . Because likelihood can be very small when you have much data, typically one works with log-likelihoods. For example:

```

logLik(model1)

## 'log Lik.' -13.14838 (df=3)

logLik(model2)

## 'log Lik.' -2.773007 (df=9)

```

Typically, more complex models will yield better (less negative) log-likelihoods. We therefore want to penalize more complex models in some way.

Fox et al. (Research Integrity and Peer Review, 2017) analyzed the invitations to review for several scientific journals, and found that “The proportion of invitations that lead to a submitted review has been decreasing steadily over 13 years (2003–2015) for four of the six journals examined, with a cumulative effect that has been quite substantial”. Their data is stored in `../data/FoxEtAl.csv`.

We're going to build models trying to predict whether a reviewer will agree (or not) to review a manuscript.

```
# read the data
reviews <- read.csv("../data/FoxEtAl.csv", sep = "\t")
# take a peek
head(reviews)

##   Sort Journal msID Year ReviewerID ReviewerInvited ReviewerResponded
## 1    1 Evolution 34152 2007     8426852                 1                  1
## 2    2 Evolution 34152 2007     8425970                 1                  1
## 3    3 Evolution 34152 2007     8425116                 1                  1
## 4    4 Evolution 34152 2007     8426128                 1                  1
## 5    5 Evolution 34152 2007     9327585                 1                  1
## 6    6 Evolution 34152 2007     8423528                 1                  1
##   ReviewerAgreed ReviewerAssigned ReviewSubmitted
## 1            0              0                NA
## 2            0              0                NA
## 3            0              0                NA
## 4            0              0                NA
## 5            1              1                1
## 6            0              0                NA

# set NAs to 0
reviews[is.na(reviews)] <- 0
# how big is the data?
dim(reviews)

## [1] 113876      10

# that's a lot! Let's take 5000 review invitations for our explorations;
# we will fit the whole data set later
set.seed(101)
small <- reviews[order(runif(nrow(reviews))), ] [1:5000,]
```

Logistic regression

We will be playing with logistic regression. Call π_i the probability that a reviewer will agree to review manuscript i . We model $\text{logit}(\pi_i) = \log(\pi_i/(1 - \pi_i))$ as a linear function. This type of regression (along with the probit) is often used to model binary response variables.

Constant rate

As a null model we build a model in which the probability to agree to review does not change in time/for journals:

```

# suppose the rate at which reviewers agree is a constant
mean(small$ReviewerAgreed)

## [1] 0.466

# fit a logistic regression
model_null <- glm(ReviewerAgreed~1, data = small, family = "binomial")
summary(model_null)

##
## Call:
## glm(formula = ReviewerAgreed ~ 1, family = "binomial", data = small)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -1.120  -1.120  -1.120   1.236   1.236
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.13621   0.02835 -4.805 1.55e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 6908.3 on 4999 degrees of freedom
## Residual deviance: 6908.3 on 4999 degrees of freedom
## AIC: 6910.3
##
## Number of Fisher Scoring iterations: 3

# interpretation:
exp(model_null$coefficients[1]) / (1 + exp(model_null$coefficients[1]))

```

```

## (Intercept)
##          0.466

```

Declining trend

We now build a model in which the probability to review declines steadily from year to year:

```

# Take 2003 as baseline
model_year <- glm(ReviewerAgreed~I(Year - 2003), data = small, family = "binomial")
summary(model_year)

```

```

## 
## Call:
## glm(formula = ReviewerAgreed ~ I(Year - 2003), family = "binomial",
##      data = small)
##
## Deviance Residuals:
##    Min      1Q   Median      3Q     Max
## -1.3441 -1.0879 -0.9686  1.2372  1.4017
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.383697  0.065482  5.860 4.64e-09 ***
## I(Year - 2003) -0.074753  0.008491 -8.804 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 6908.3 on 4999 degrees of freedom
## Residual deviance: 6829.7 on 4998 degrees of freedom
## AIC: 6833.7
##
## Number of Fisher Scoring iterations: 4

```

Each year has its own parameter

What if the probability to agree were to vary from year to year, with no clear trend?

```

# Take 2003 as baseline
model_eachyr <- glm(ReviewerAgreed~as.factor(Year), data = small, family = "binomial")
summary(model_eachyr)

```

Journal dependence

Reviewers might be more likely to agree for more prestigious journals:

```

# Take the first journal as baseline
model_journal <- glm(ReviewerAgreed~Journal, data = small, family = "binomial")
summary(model_journal)

```

```

## 
## Call:
## glm(formula = ReviewerAgreed ~ Journal, family = "binomial",
##      data = small)
##
## Deviance Residuals:

```

```

##      Min       1Q   Median       3Q      Max
## -1.208 -1.110 -1.033  1.247  1.329
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.07187  0.06812  1.055  0.29141
## JournalFE   -0.27908  0.09411 -2.965  0.00302 **
## JournalJANIM -0.16544  0.09598 -1.724  0.08476 .
## JournalJAPPL -0.23319  0.09323 -2.501  0.01237 *
## JournalJECOL -0.26183  0.09403 -2.785  0.00536 **
## JournalMEE   -0.42223  0.12868 -3.281  0.00103 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 6908.3 on 4999 degrees of freedom
## Residual deviance: 6892.7 on 4994 degrees of freedom
## AIC: 6904.7
##
## Number of Fisher Scoring iterations: 3

```

Model journal and year

Finally, we can build a model combining both features: we fit a parameter for each journal/year combination

```

# Take the first journal as baseline
model_journal_yr <- glm(ReviewerAgreed~Journal:I(Year-2003),
                         data = small, family = "binomial")
#summary(model_journal_yr)

```

For completeness, the most complicated model, in which each journal and year combination has its own parameter.

```

# Take the first journal as baseline
model_journal_eachyr <- glm(ReviewerAgreed~Journal:as.factor(Year),
                             data = small, family = "binomial")
#summary(model_journal_eachyr)

```

Likelihoods

In R, you can extract the log-likelihood from a model object calling the function `logLik`

```
logLik(model_null)
```

```
## 'log Lik.' -3454.167 (df=1)
```

```

logLik(model_year)

## 'log Lik.' -3414.845 (df=2)

logLik(model_eachyr)

## 'log Lik.' -3405.309 (df=13)

logLik(model_journal)

## 'log Lik.' -3446.335 (df=6)

logLik(model_journal_yr)

## 'log Lik.' -3395.365 (df=7)

logLik(model_journal_eachyr)

## 'log Lik.' -3365.024 (df=68)

```

Interpretation: because we're dealing with binary data, the likelihood is the probability of correctly predicting the agree/not agree for all the 5000 invitations considered. Therefore, the probability of guessing a (random) invitation correctly under the first model is:

```

exp(as.numeric(logLik(model_null)) / 5000)

## [1] 0.5011582

while the most complex model yields

exp(as.numeric(logLik(model_journal_eachyr)) / 5000)

## [1] 0.5101733

```

We didn't improve our guessing much by considering many parameters! This could be due to specific data points that are hard to predict, or mean that our explanatory variables are not sufficient to model our response variable.

AIC

One of the simplest methods to select among competing models is the Akaike Information Criterion (AIC). It penalizes models according to the **number of parameters**: $AIC = 2p - 2\log L(\theta|D, M)$, where p is the number of parameters. Note that **smaller** values of AIC stand for "better" models. In R you can compute AIC using:

```
AIC(model_null)
```

```
## [1] 6910.334
```

```
AIC(model_year)
```

```
## [1] 6833.691
```

```
AIC(model_eachyr)
```

```
## [1] 6836.618
```

```
AIC(model_journal)
```

```
## [1] 6904.669
```

```
AIC(model_journal_yr)
```

```
## [1] 6804.73
```

```
AIC(model_journal_eachyr)
```

```
## [1] 6866.049
```

As you can see, AIC would favor the `model_journal_yr` model.

AIC is rooted in information theory and measures (asymptotically) the loss of information when using the model instead of the data. There are several limitations of AIC: a) it only holds asymptotically (i.e., for very large data sets; for smaller data you need to correct it); it penalizes each parameter equally (i.e., parameters that have a large influence on the likelihood have the same weight as parameters that do no influence the likelihood much); it can lead to overfitting, favoring more complex models in simulated data generated by simpler models.

BIC

In a similar vein, BIC (Bayesian Information Criterion) uses a slightly different penalization: $BIC = \log(n)p - 2\log L(\theta|D, M)$, where n is the number of data points. Again, smaller values stand for “better” models:

```
BIC(model_null)
```

```
## [1] 6916.851
```

```

BIC(model_year)

## [1] 6846.725

BIC(model_eachyr)

## [1] 6921.341

BIC(model_journal)

## [1] 6943.772

BIC(model_journal_yr)

## [1] 6850.35

BIC(model_journal_eachyr)

## [1] 7309.218

```

Note that according to BIC, `model_year` is favored.

Cross validation

One very robust method to perform model selection, often used in machine learning, is cross-validation. The idea is simple: split the data in three parts: a small data set for exploring; a large set for fitting; a small set for testing (for example, 5%, 75%, 20%). You can use the first data set to explore freely and get inspired for a good model. The data will be then discarded. You use the largest data set for accurately fitting your model(s). Finally, you validate your model or select over competing models using the last data set.

Because you haven't used the test data for fitting, this should dramatically reduce the risk of overfitting. The downside of this is that we're wasting precious data. There are less expensive methods for cross validation, but if you have much data, or data are cheap, then cross-validation has the virtue of being fairly robust.

Let's try our hand at cross-validation. First, we split the data into three parts:

```

reviews$cv <- sample(1:3, nrow(reviews), prob = c(0.05, 0.75, 0.2), replace = TRUE)
dataexplore <- reviews[reviews$cv == 1,]
datafit <- reviews[reviews$cv == 2,]
datatest <- reviews[reviews$cv == 3,]
# We've already done our exploration.
# Let's fit the data using model_journal
# and model_journal_yr, which seem to be the most promising
cv_model1 <- glm(ReviewerAgreed~I(Year-2003), data = datafit, family = "binomial")
cv_model2 <- glm(ReviewerAgreed~Journal:I(Year-2003), data = datafit, family = "binomial")

```

Now that we've fitted the models, we can use the function predict to find the fitted values for the testdata:

```
mymodel <- cv_model1
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

```
## [1] -15520.91
```

repeat for the other model

```
mymodel <- cv_model2
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

```
## [1] -15473.93
```

Cross validation supports the choice of the more complex model.

Other approaches

Bayesian models are gaining much traction in biology. The advantage of these models is that you can get a posterior distribution for the parameter values, reducing the need for p-values and AIC. The downside is that fitting these models is computationally much more expensive (you have to find a distribution of values instead of a single value).

There are three main ways to perform model selection in Bayesian models:

- **Reversible-jump MCMC** You build a Monte Carlo Markov Chain that is allowed to “jump” between models. You can choose a prior for the probability of being in each of the models; the posterior distribution gives you an estimate of how much the data supports each model. Upside: direct measure. Downside: difficult to implement in practice – you need to avoid being “trapped” in a model.
- **Bayes Factors** Ratio between the probability of two competing models. Can be computed analytically for simple models. Can also be interpreted as the average likelihood when parameters are chosen according to their prior (or posterior) distribution. Upside: straightforward interpretation — it follows from Bayes theorem; Downside: in most cases, one needs to approximate it; can be tricky to compute for complex models.
- **DIC** Similar to AIC and BIC, but using distributions instead of point estimates.

Another alternative paradigm for model selection is Minimum-Description Length. The spirit is that a model is a way to “compress” the data. Then you want to choose the model whose total description length (compressed data + description of the model) is minimized.

A word of caution

The “best” model you’ve selected could still be a terrible model (best among bad ones). Out-of-fit prediction (such as in the cross-validation above) can give you a sense of how well you’re modeling the data.

When in doubt, remember the (in)famous paper in Nature by Tatem et al. 2004, which used some flavor of model selection to claim that, according to their linear regression, in the 2156 olympics the fastest woman would run faster than the fastest man. One of the many memorable letters that ensued reads:

Sir — A. J. Tatem and colleagues calculate that women may out-sprint men by the middle of the twenty-second century (Nature 431,525; 2004). They omit to mention, however, that (according to their analysis) a far more interesting race should occur in about 2636, when times of less than zero seconds will be recorded.

In the intervening 600 years, the authors may wish to address the obvious challenges raised for both time-keeping and the teaching of basic statistics.

Kenneth Rice

Programming Challenge

Presidential tweets and crimes in Chicago

To try your hand at p-hacking and overfitting, and show how these practices can lead to completely inane results, you are going to show the strong correlation between the tweets by President Obama (or then presidential candidate Donald Trump) in 2016 and the number of crimes in Chicago. For example, here’s code showing that the number of tweets by Trump correlates with narcotics violations in Chicago.

```
library(dplyr)
library(ggplot2)
library(readr)
# read tweets
trump <- read_csv("../data/Trump_Tweets_2016.csv")
obama <- read_csv("../data/Obama_Tweets_2016.csv")
# read crimes
crimes <- read_csv("../data/Chicago_Crimes_2016.csv")

# count tweets by day
obama_all <- obama %>% group_by(day, month) %>% tally()
trump_all <- trump %>% group_by(day, month) %>% tally()
```

```

# join the data sets
crimes <- left_join(crimes, obama_all %>% rename(obama = n))
crimes <- left_join(crimes, trump_all %>% rename(trump = n))
# set crimes with 0 occurrences in a day
crimes[is.na(crimes)] <- 0

# take a look at the data
head(crimes)

```

```

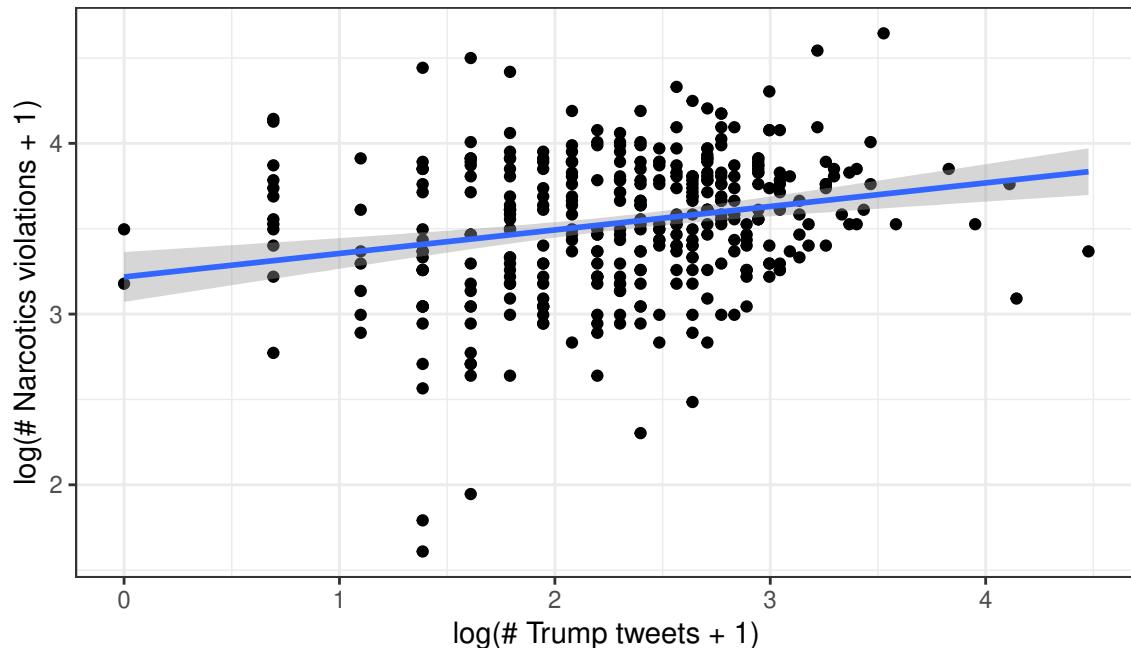
## # A tibble: 6 x 6
##   day month PrimaryType     n obama trump
##   <dbl> <dbl> <chr>     <dbl> <dbl> <dbl>
## 1     1     1 ARSON       1     0    14
## 2     1     2 ARSON       0     0     4
## 3     1     3 ARSON       0     0    12
## 4     1     4 ARSON       2     0    10
## 5     1     5 ARSON       1     0     9
## 6     1     6 ARSON       1     0     3

```

```

# show that narcotics violations correlate with Trump's tweets
narco <- crimes %>% filter(PrimaryType == "NARCOTICS")
pl <- ggplot(narco) + aes(x = log(trump + 1), y = log(n + 1)) +
  theme_bw() + geom_point() + geom_smooth(method = "lm") +
  xlab("log(# Trump tweets + 1)") + ylab("log(# Narcotics violations + 1)")
show(pl)

```



```
# print correlation
print(cor.test(log(narco$trump + 1), log(narco$n + 1)))
```

```
##
## Pearson's product-moment correlation
##
## data: log(narco$trump + 1) and log(narco$n + 1)
## t = 4.4758, df = 364, p-value = 1.019e-05
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.1289053 0.3233371
## sample estimates:
## cor
## 0.2283974
```

You can see that the correlation is respectable (0.23), and that the p-value is minuscule (0.00001)! Try your hand at finding something even more striking. You can:

- transform the data (e.g., use logs, square roots, binning)
- combine multiple crimes
- use a specific date range
- summarize the data by week or by month
- use correlation (Pearson, Spearman, Kendall) or generalized linear models
- etc.

Once you've found something worth reporting in the front page of Chicago's newspapers, post your answer.

Workshops

Workshop A. Murat Eren
Workshop Stephanie Palmer
Workshop Matthias Steinrücken

QBio5 :: Microbiome workshop

A glimpse into the microbial jungle in your mouth

*A. Murat Eren**[†]

Evan Kieft[‡]

Our planet is microbial. The astonishing number of microbial organisms that occupy terrestrial and marine habitats of Earth represent a biomass that exceeds every living organism that can be seen by naked eye, combined. They can survive in a wide range of environmental and chemical gradients, so even the most extreme environments we can find on Earth have some microbial ambassadors, carrying the flag of life to places where you don't want to go. They also are the engine of our planet as they govern large and critical biogeochemical cycles that make Earth a habitable planet for much less talented organisms (such as ourselves) by doing the real tough jobs you don't want to do. Pretty much they are the best.

Our own body is microbial, too. Just to put things into perspective, for every human cell that make up our own body, there is one or more bacterial cells that live on us. Starting from the moment we are born, microbes are with us throughout our journey in life and even a little after that: they help us maintain our health by extracting energy from things we can't digest, by synthesizing vitamins or metabolizing xenobiotics for us, and help us return all the things we borrowed in pristine conditions so other things can be built from us. They are just beautiful like that.

Microbiologists who realized early on that **a complete understanding of life is only possible through a complete understanding of microbes** have been studying the evolution and ecology of microbes everywhere relentlessly for many decades. Of course, the emergence of advanced molecular and computational approaches had a huge influence on this quest, and allowed people like Meren and Evan to be relevant to fundamental questions of microbiology with their computational skills. **As a group (<http://merenlab.org>) we seek answers in very large sequencing datasets about microbial life.** How do they respond to changing environments? How do they evolve? What roles do they play in health and disease? To investigate these questions, we study all sorts of environments: from human gut to the surface ocean, from mosquito ovaries to sewage ecosystems.

Where does the oral cavity fit here? **The oral cavity is also colonized by bacteria** just like every other surface on your body. A complete microbial understanding of this environment has always been essential for medical reasons, but besides its immediate relevance for overall health, we believe **the oral cavity represents a fascinating environment to study the ecology of microbes**. Imagine how every microbial cell can go anywhere in the mouth due to the lack of any physical barriers, and continuous flow of saliva. But, their distribution is far from being random in that jungle of microbial life you all maintain in your mouths. And we are very curious to see whether we can make better sense of how they are distributed to later learn what makes them do that.

The purpose of this workshop is to give you a glimpse of that invisible jungle in your mouth by making sense of high-throughput sequencing data from the oral cavity using R and ggplot.

At the end of this workshop you will have an idea about how the community structures of naturally occurring microbes that live in a given environment can be studied with currently available molecular tools, sequencing technologies, and computational approaches. You will also gain more insights into the power of exploratory data visualization with ggplot, and the power of R in manipulating data.

When you are done here, you will know *whether the microbes live on your tongue are more similar to the ones that live on your cheek, or whether they are more similar to the ones that live on the tongue of the person next to you.*

*Department of Medicine, University of Chicago

[†]Marine Biological Laboratory

[‡]The Biophysical Sciences Program at the University of Chicago.

Setting the stage

The primary raw data we will be playing with throughout this tutorial come from the Human Microbiome Project (HMP, <https://hmpdacc.org/>), a National Institutes of Health (<http://www.nih.gov>) initiative that attempted to make sense of the ‘normal microbiome’ of healthy individuals. The HMP recruited many healthy volunteers, and collected multiple samples from each one of them to study microbes that lived in the healthy human gut, urogenital tract, oral and nasal cavities, as well as skin.

Here we will focus only on the oral cavity to characterize the microbial communities of this particular environment (because the oral cavity is the best), and we will do this in a highly resolved manner using ‘oligotypes’. The dataset we will re-analyze essentially comes from the supplementary tables of our 2014 study (<http://www.pnas.org/content/111/28/E2875.short>), which is available to you in PDF form in the `text` directory. In the same directory you can also find a copy of Carl Zimmer’s take on our study, “The Zoo in the Mouth”.

OK. First things first. We will need the following libraries throughout this tutorial for statistical analyses (`vegan`), re-formatting the input data (`reshape2`), and to visualize it (`ggplot2`):

```
library(vegan)
library(reshape2)
library(ggplot2)
```

If you are missing any of these libraries, you can install them using `install.packages("LIBRARY_NAME_HERE")` notation.

There are two data files for you to read in. The first one is the observation matrix that shows the distribution of each oligotype across each sample:

```
oligotypes <- read.table('~/data/oligotypes.txt',
                         header = TRUE,
                         sep="\t")
```

The second data file contains data about our samples:

```
samples <- read.table('~/data/samples.txt',
                      header = TRUE,
                      sep="\t")
```

Feel free to take a look at its format:

```
head(samples)

##           sample individual environment site
## 1  s_147406386_BM s_147406386 ORAL_CAVITY   BM
## 2  s_147406386_HP s_147406386 ORAL_CAVITY   HP
## 3  s_147406386_KG s_147406386 ORAL_CAVITY   KG
## 4  s_147406386_PT s_147406386 ORAL_CAVITY   PT
## 5  s_147406386_ST s_147406386        GUT   ST
## 6 s_147406386_SUBP s_147406386 ORAL_CAVITY SUBP
```

Samples in our data describes two main environments:

```
levels(samples$environment)

## [1] "GUT"          "ORAL_CAVITY"
```

And more specifically, ten body sites:

```
levels(samples$site)

## [1] "BM"   "HP"   "KG"   "PT"   "ST"   "SUBP" "SUPP" "SV"   "TD"   "TH"
```

While we have 148 individuals:

```
length(levels(samples$individual))

## [1] 148
```

We have a total of 1475 samples:

```
length(levels(samples$sample))

## [1] 1475
```

Fine. We have everything we need.

A visualization-driven exploration of the data

MDS

For this exploration, we're going to use multi-dimensional scaling, a commonly used technique to make sense of large dimensional data sets. In particular, MDS takes as input a “distance” matrix (say the distance between samples, sequences, etc.), and tries to find a projections onto few dimensions (typically 2) that can be used to find “clusters” of similar samples.

To see how this works, let's load some non-biological data.

```
load("../data/travel_times.RData")
```

The matrix `travel_time` contains the number of minutes it would take you to drive between two of the major US cities listed. You can take a look at the data by calling

```
View(travel_times)
```

Now we're going to apply MDS to the data, and find which two cities are “similar” (i.e., close to reach by car).

```
fit_mds <- cmdscale(travel_times, k = 2) # use two dimensions
```

Let's plot the results

```
cities_names <- rownames(travel_times)
plot(fit_mds)
text(fit_mds, labels = cities_names)
```

Not bad! Now, let's try with our microbial community.

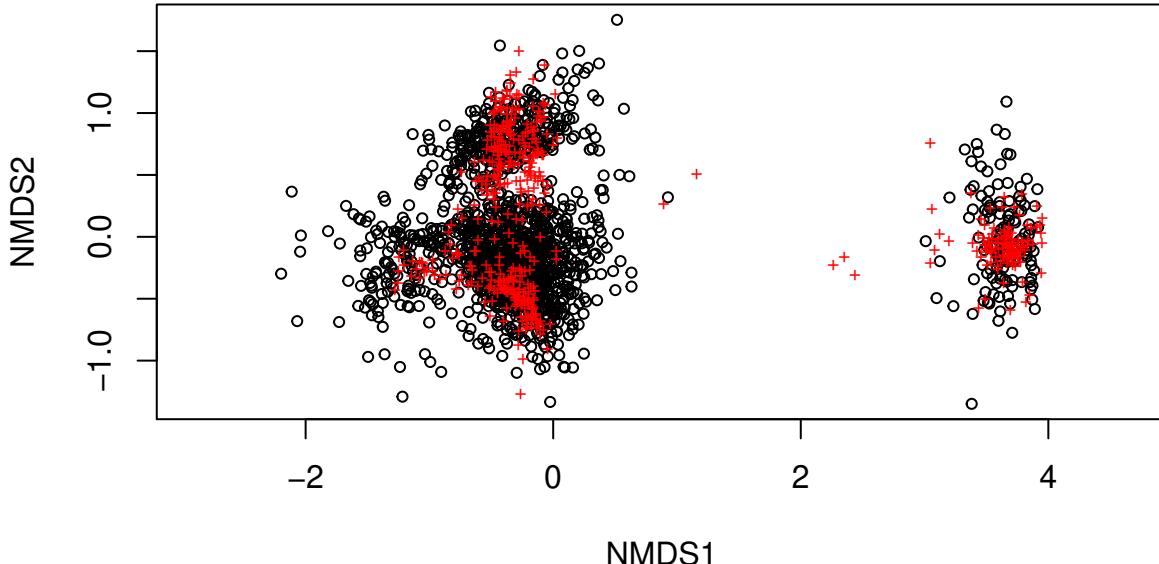
Back to the oral cavity

Here we create an ordination of our data using MDS:

```
# generate the mds object using the Morisita-Horn distance
# (this will take some time)
mds <- metaMDS(oligotypes[,-1], distance='horn')
```

We take a very quick look at the resulting ordination:

```
# show it
plot(mds)
```



We can all agree that this looks quite useless.

Instead of the `plot` function, we could use `ggplot` to have more control over our visualization needs by adding ad hoc information into our plots in an intuitive manner. But `ggplot` will not like the way `mds` object is formatted. But we can turn that object into a data frame rich with information:

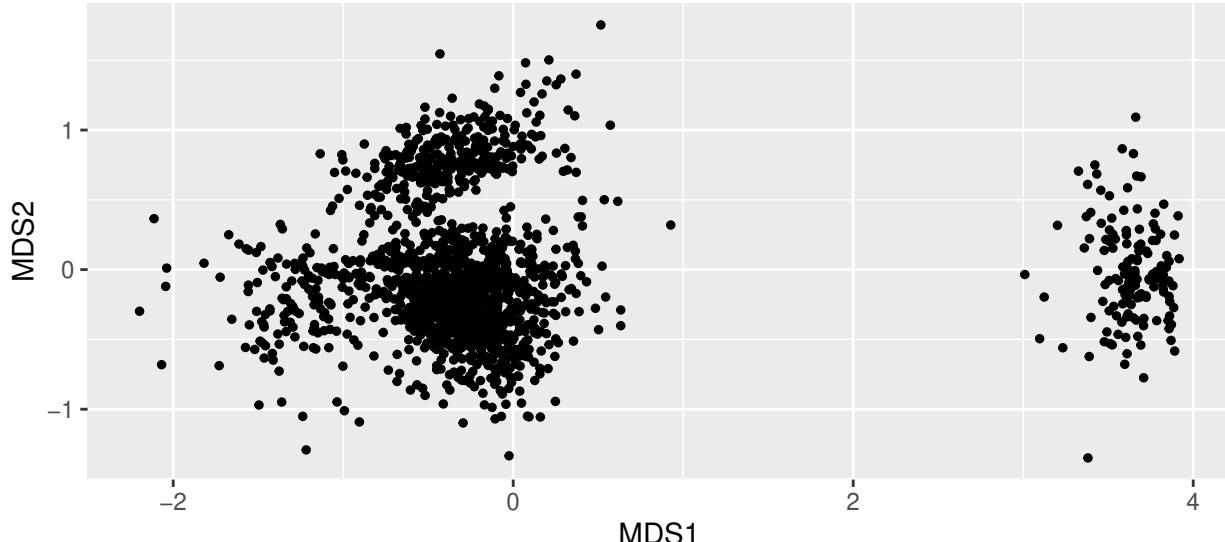
```
# generate a data frame
mds_df <- data.frame(MDS1 = mds$points[,1],
                      MDS2 = mds$points[,2],
                      individual=with(samples, get("individual")),
                      environment=with(samples, get("environment")),
                      site=with(samples, get("site")))

# take a peek
head(mds_df)
```

| | MDS1 | MDS2 | individual | environment | site |
|------|------------|-------------|-------------|-------------|------|
| ## 1 | -0.9001296 | -0.11533750 | s_147406386 | ORAL_CAVITY | BM |
| ## 2 | -0.6191344 | -0.32741437 | s_147406386 | ORAL_CAVITY | HP |
| ## 3 | -1.2018615 | -0.04347894 | s_147406386 | ORAL_CAVITY | KG |
| ## 4 | -0.1139542 | -0.28778080 | s_147406386 | ORAL_CAVITY | PT |
| ## 5 | 3.9113955 | 0.38500523 | s_147406386 | GUT | ST |
| ## 6 | -0.7747015 | 0.61157748 | s_147406386 | ORAL_CAVITY | SUBP |

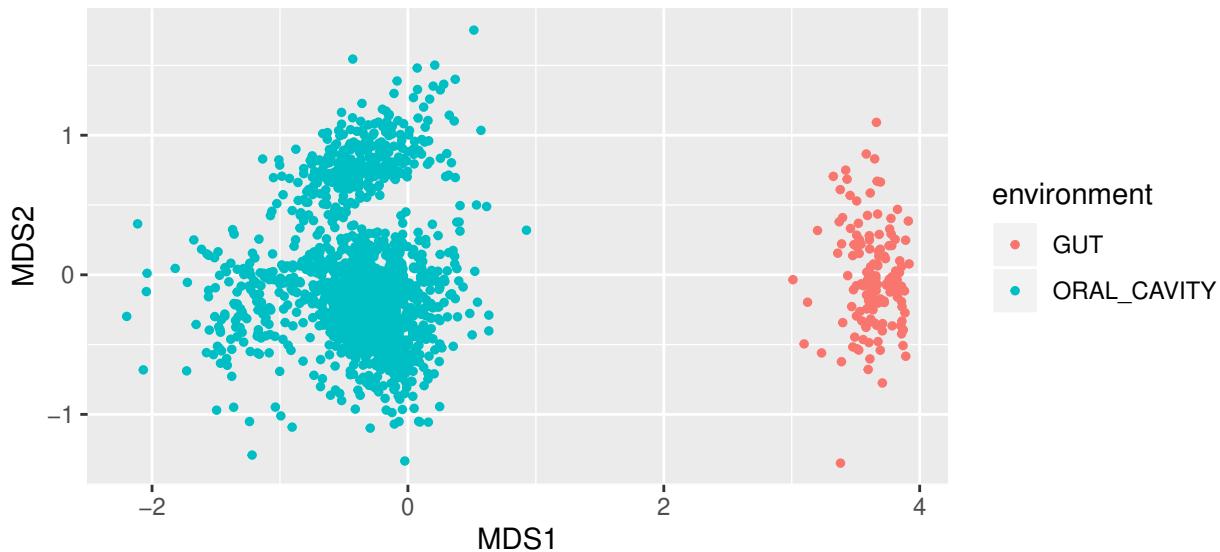
Well, this is more like it. Now we can take a look this with `ggplot`:

```
p <- ggplot(data = mds_df, aes(MDS1, MDS2))
p <- p + geom_point(size = 1)
p
```



This is not much better than the previous plot, but this time we can easily manipulate our visual objects. Let's say we color our points in this display by `environment` to ask this very question: *from the perspective of microbes, do our guts look like our mouths?*:

```
p <- ggplot(data = mds_df, aes(MDS1, MDS2))
p <- p + geom_point(aes(color=environment), size=1)
p
```



This is relieving.

Let's remove all gut samples to focus solely on oral microbes:

```
# take the subset of both data frames:
oral_oligotypes <- oligotypes[!grepl("_ST", oligotypes$sample), ]
oral_samples <- samples[samples$environment == "ORAL_CAVITY", ]

# set the factors straight:
oral_samples$sample <- factor(oral_samples$sample)
oral_samples$site <- factor(oral_samples$site)
```

Since we changed the shape of the data quite a bit, it is better to re-compute the ordination of our samples:

```

# new mds object:
oral_mds <- metaMDS(oral_oligotypes[,-1], distance='horn')

# generating a data frame from it:
oral_mds_df <- data.frame(MDS1 = oral_mds$points[,1],
                           MDS2 = oral_mds$points[,2],
                           individual=with(oral_samples, get("individual")),
                           site=with(oral_samples, get("site")))

# taking a quick look from it because why not:
head(oral_mds_df)

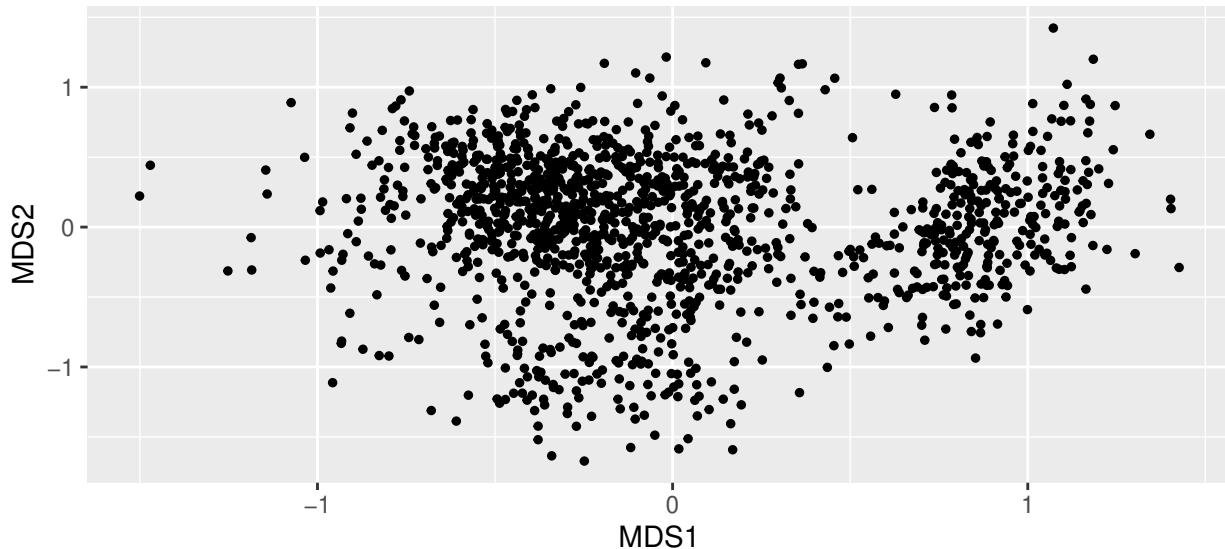
```

Alright! Let's take a quick look:

```

p <- ggplot(data = oral_mds_df, aes(MDS1, MDS2))
p <- p + geom_point(size=1)
p

```



What chaos. What if we color based on individuals:

```

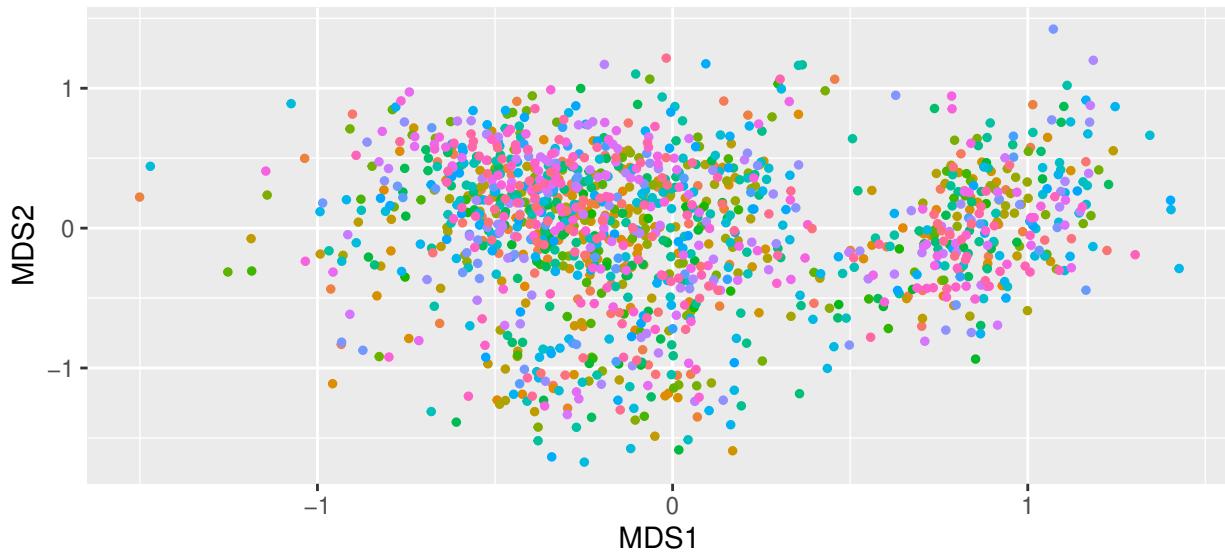
p <- ggplot(data = oral_mds_df, aes(MDS1, MDS2))
p <- p + geom_point(aes(color=individual), size=1)
p

```

| | | | | | | | | | | | |
|---|-------------|---|-------------|---|-------------|---|-------------|---|-------------|---|-------|
| • | s_158964549 | • | s_159591683 | • | s_160380657 | • | s_16098560 | • | s_370425937 | • | s_63 |
| † | s_159005010 | • | s_159611913 | • | s_160400887 | • | s_161007791 | • | s_404239096 | • | s_763 |
| • | s_159146620 | • | s_159632143 | • | s_160421117 | • | s_161028021 | • | s_414519462 | • | s_763 |
| • | s_159207311 | • | s_159672603 | • | s_160441347 | • | s_161068481 | • | s_432193348 | • | s_763 |
| • | s_159227541 | • | s_159713063 | • | s_160461578 | • | s_161230322 | • | s_441369442 | • | s_763 |
| • | s_159247771 | • | s_159814214 | • | s_160481808 | • | s_161270782 | • | s_451588811 | • | s_763 |
| • | s_159268001 | • | s_159915365 | • | s_160502038 | • | s_161311242 | • | s_492786515 | • | s_764 |
| • | s_159288231 | • | s_160016515 | • | s_160542498 | • | s_161331472 | • | s_514014184 | • | s_764 |
| • | s_159308461 | • | s_160036745 | • | s_160582958 | • | s_161351702 | • | s_517810313 | • | s_764 |
| • | s_159328691 | • | s_160056975 | • | s_160603188 | • | s_161412393 | • | s_533247696 | • | s_764 |
| • | s_159369152 | • | s_160097436 | • | s_160643649 | • | s_161473083 | • | s_604812005 | • | s_764 |
| • | s_159389382 | • | s_160137896 | • | s_160663879 | • | s_161554003 | • | s_612472597 | • | s_764 |
| • | s_159429842 | • | s_160158126 | • | s_160684109 | • | s_206906765 | • | s_638754422 | • | s_764 |
| • | s_159450072 | • | s_160178356 | • | s_160704339 | • | s_208027353 | • | s_650853796 | • | s_764 |
| • | s_159470302 | • | s_160218816 | • | s_160765029 | • | s_246515023 | • | s_668248235 | • | s_764 |
| • | s_159490532 | • | s_160239046 | • | s_160845950 | • | s_257905678 | • | s_682449369 | • | s_764 |
| • | s_159510762 | • | s_160259276 | • | s_160906640 | • | s_295137534 | • | s_686765762 | • | s_764 |
| • | s_159551223 | • | s_160319967 | • | s_160947100 | • | s_336497421 | • | s_737052003 | • | s_764 |
| • | s_159571453 | • | s_160340197 | • | s_160967330 | • | s_370027359 | • | s_763456073 | • | s_764 |

Ouch. We don't see anything, because the legend takes the entire space. Let's disable the legend and try again:

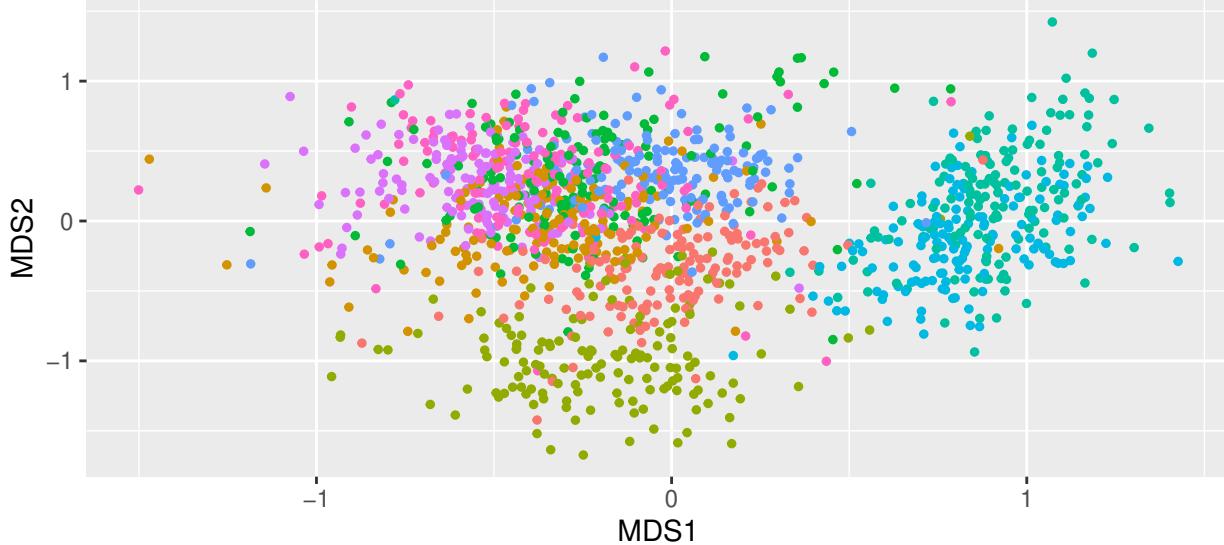
```
p <- ggplot(data = oral_mds_df, aes(MDS1, MDS2))
p <- p + geom_point(aes(color=individual), size=1)
p <- p + theme(legend.position="none")
p
```



Much better. But this doesn't seem to have any structure. Why?

OK. How about we color samples based on oral sites:

```
p <- ggplot(data = oral_mds_df, aes(MDS1, MDS2))
p <- p + geom_point(aes(color=site), size=1)
p <- p + theme(legend.position="none")
p
```



Aha!

What does this tell us?

(It would be great to do an ANOVA here to show oral sites explain a much more significant amount of variance in the dataset before moving on to the next chapter)

Making publication-ready visualizations with R

Let's say we wish to put some circles around our groups to help visualize their distribution and dispersal.

The function below will help us do that by returning all the x and y coordinates to draw a perfect ellipse on an ordination. It is coming from the depths of the library `vegan`, and here we simply are hacking it so we can use it to put ellipses on an ordination drawn by `ggplot`, rather than `vegan`:

```
veganCovEllipse <- function (cov_matrix, center){
  theta <- (0:100) * 2 * pi/100
  circle <- cbind(cos(theta), sin(theta))

  # here we have a perfect circle around the point zero, and the following line will
  # turn it into an ellipse by centering and multiplying that innocent circle with the
  # Choleski-decomposed input covariance matrix, which will represents the variation
  # among the distribution of samples that belong to a single group on the ordination
  # (this part will be much clear when you look at the for loop in the next step where
  # this function is called). if you are not familiar, the notation `%%` is for
  # matrix multiplication. yes, you got it. this entire thing is absolute magic!
  ell <- t(center + t(circle %*% chol(cov_matrix)))

  return(as.data.frame(ell))
}
```

Using the magic up above, we will generate a new data frame, `ellipses_df`, to keep track of ellipses around our data points by going through each group in the for loop below:

```
ellipses_df <- data.frame()

# mighty for loop .. it looks ugly, but is very simple:
for(g in levels(oral_mds_df$site)){
  # get a smaller data frame just for site:
  s_df <- oral_mds_df[oral_mds_df$site==g, ]

  # calculate its center and its covariance matrix:
  center <- c(mean(s_df$MDS1), mean(s_df$MDS2))
  cov_matrix <- cov.wt(cbind(s_df$MDS1, s_df$MDS2))$cov

  # get the ellipse:
  ellipse <- veganCovEllipse(cov_matrix, center)

  # add the new ellipse to the data frame
  ellipses_df <- rbind(ellipses_df, cbind(ellipse, group=g))
}

# let's name the columns in our data frame more appropriately:
names(ellipses_df) <- c('x_coord', 'y_coord', 'group')
```

OK. You must be curious about what comes out of this black magic. Let's take a look at this new data frame:

```
head(ellipses_df)
```

```
##      x_coord    y_coord group
## 1 0.2267117 -0.1990243   BM
## 2 0.2262480 -0.1819480   BM
## 3 0.2248588 -0.1654502   BM
## 4 0.2225495 -0.1495960   BM
## 5 0.2193293 -0.1344480   BM
## 6 0.2152109 -0.1200659   BM
```

Don't let it fool you, this data frame has many entries since it is supposed to draw elliptic objects on our ordination:

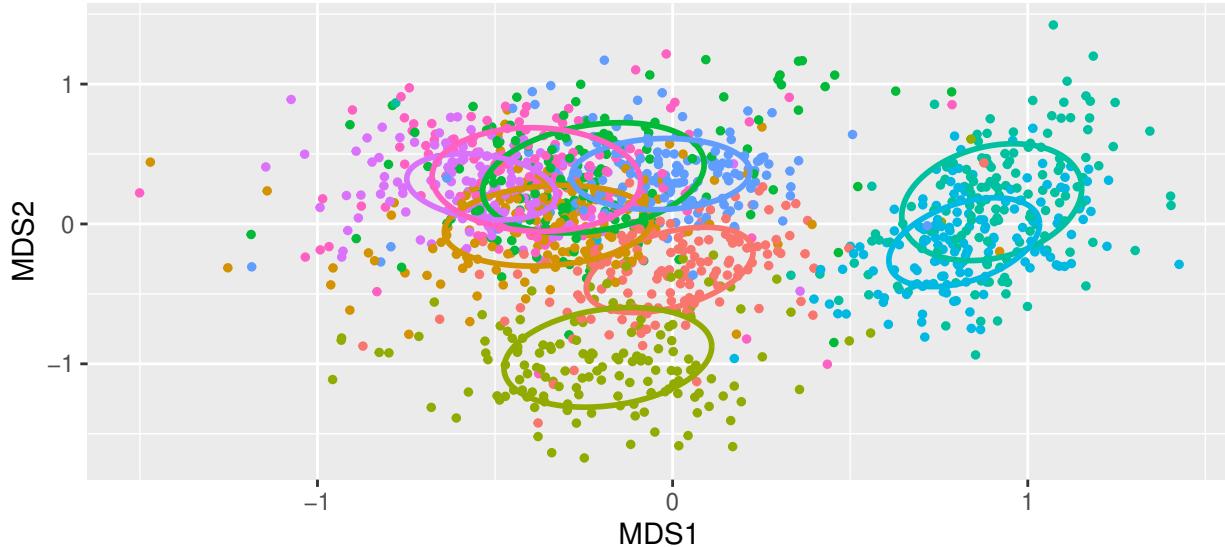
can you predict how many points should it have by looking at the function `veganCovEllipse`?

```
nrow(ellipses_df)
```

```
## [1] 909
```

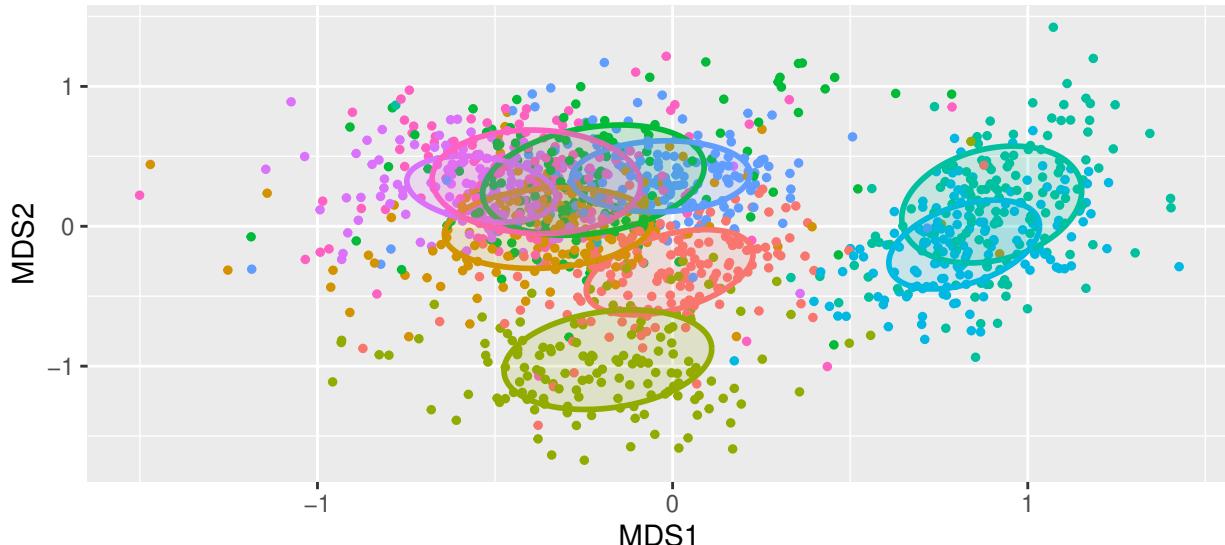
We still have the `ggplot` object in memory, let's add the data frame we just put together:

```
p <- p + geom_path(data=ellipses_df,
                     aes(x=x_coord, y=y_coord, colour=group),
                     size=1,
                     linetype=1)
p
```



There is always room for improvement:

```
p <- p + geom_polygon(data=ellipses_df,
                       aes(x=x_coord, y=y_coord, group=group, fill=group),
                       alpha=0.15)
p
```



It would have been great if we knew exactly what these ellipses represent. Let's add some labels at the center of each. For this, we first need to compute the group means of our samples:

```
oral_mds_group_means = aggregate(oral_mds_df[,1:2],
                                  list(group=with(oral_samples, get('site'))),
                                  mean)
```

Basically this is a new data frame that looks like this:

```
oral_mds_group_means
```

```
##   group      MDS1      MDS2
## 1    BM -0.008270321 -0.32853086
## 2    HP -0.350823982 -0.01267178
```

```

## 3     KG -0.182353841 -0.95353820
## 4     PT -0.222971056  0.32986660
## 5   SUBP  0.898656623  0.15570038
## 6   SUPP  0.821035565 -0.13045405
## 7     SV -0.033550862  0.35681564
## 8     TD -0.537879644  0.26518787
## 9     TH -0.383842482  0.31762440

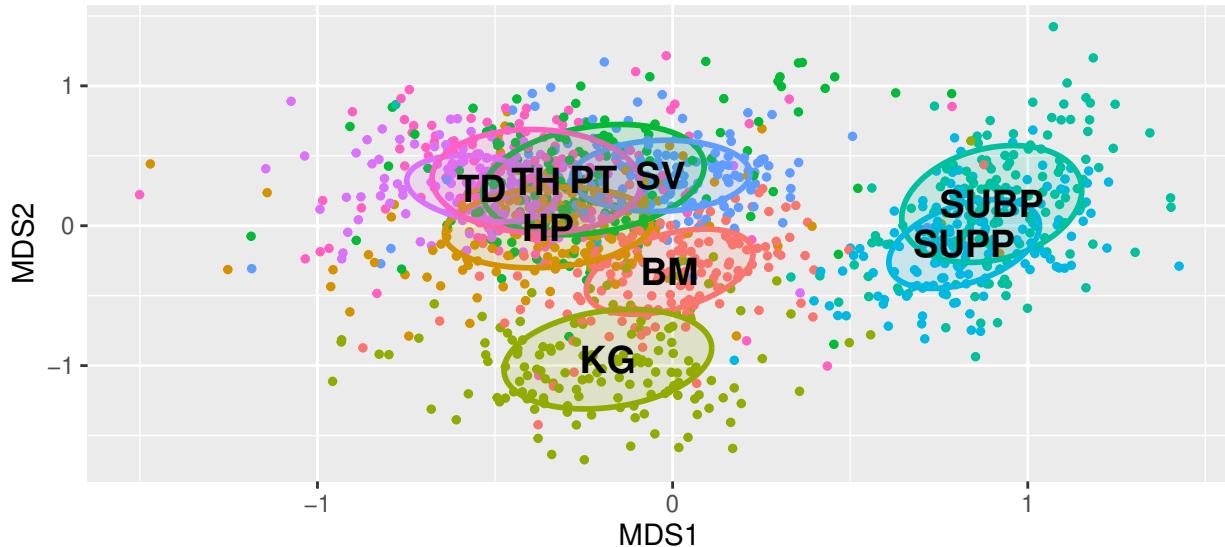
```

And we can extend the ggplot object with one more layer:

```

p <- p + annotate("text",
                   x=oral_mds_group_means$MDS1,
                   y=oral_mds_group_means$MDS2,
                   label=oral_mds_group_means$group,
                   size=5,
                   fontface = 2)
p

```



Neural computation workshop

Stephanie Palmer & Jeff Johnston

Contents

| | |
|--|-----------|
| Installation notes | 1 |
| Welcome | 1 |
| Neural encoding | 1 |
| The task | 2 |
| The brain regions of interest | 4 |
| Spiking: the substrate of neural communication | 4 |
| The spiking of an MT neuron | 5 |
| Tuning curves | 9 |
| The spiking of an LIP neuron | 10 |
| Comparing MT to LIP | 11 |
| Quantifying category tuning | 12 |
| Bonus: an important control | 13 |

Installation notes

For this tutorial, relatively up-to-date versions of R and RStudio are needed. To install R, follow instructions at cran.rstudio.com. Then install RStudio following the instructions at goo.gl/a42jYE. Download the `ggplot2`, `stats`, `plyr`, and `rmatio` packages.

Welcome

Understanding the detailed workings of the brain is one of the grand challenges in modern biology. While we have learned quite a bit about the brain in the 100 years or so since Golgi and Ramon y Cajal first revealed its neural-cellular microstructure, we still have yet to fully “crack the code” this marvelous piece of tissue uses to generate complex actions, feelings, memories, and thoughts. This workshop will demonstrate a simple computation performed by neurons in two different regions in the brain of the Rhesus macaque. It will illustrate how information is transformed when moving from sensory representations – in this case, motion direction – to more abstract, cognitive representations – in this case, categorization of motion direction into two clusters: up and to the right, and down and to the left.

Neural encoding

To solve complex tasks, the brain must often interpret continuous sensory stimuli according to some non-physical or abstract discrete category membership. Thus, both quantities (a sensory and a categorical representation) are likely to be represented in the brain. Here, we will investigate one such transform and evaluate data from both before and after it has begun. As noted, we’ll be exploring this question in the

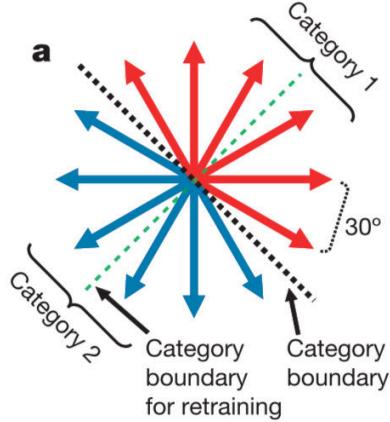


Figure 1: A schematic of the category boundaries (both original and retrained) used in the delayed match-to-category (DMC) task, see below. Taken from Freedman and Assad (2006).

context of the transformation of a visual motion signal. We'll explore how different brain areas encode something like a stimulus variable (the direction of motion of the visual object) versus something closer to a stimulus motion category (like "up" or "down").

The task

To investigate how visual information is transformed from a (roughly) continuous representation to a discrete, categorical representation, we need to ask the subject to perform a task that relies on that transformation.

Here, we'll use data originally published in Freedman and Assad (2006) *Nature*, in which they train two Rhesus macaques to categorize a visual motion stimulus according to an arbitrary category boundary, as here:

The motion stimulus is a patch of dots that move across the screen. They are placed at random positions within a small region in visual space, but all move in the same direction. The region in space where the dots appear on the screen is fixed, but the direction of motion of the dots varies from trial-to-trial. The subject is asked, during a particular trial, to view the motion of an initial dot patch, then report whether or not it matches a second moving dot patch.

Specifically, the subjects are performing a delayed match-to-category (DMC) task with the following structure:

1. They fixate in the center of a computer screen.
2. A motion stimulus in one of the 12 directions, the sample, is played.
3. There is a delay period of 1 second.
4. A second motion stimulus, the test, is played. If its category is the same as the first motion stimulus, the subject must release a lever – otherwise, they continue to hold the lever. If the lever release is correct, they receive a juice reward.
5. If the subject was supposed to continue holding the lever, there is a second delay followed by a third stimulus which always matches the category of the first stimulus – the subject must then release the lever to receive juice.

We will focus primarily on step 2 of the task. Even though this task seems fairly complicated, the subjects can perform well at it with sufficient training – and their errors follow the pattern one might expect: They make more errors when performing categorizations that are close to the category boundary, but perform well-above chance at all distances far from the boundary.

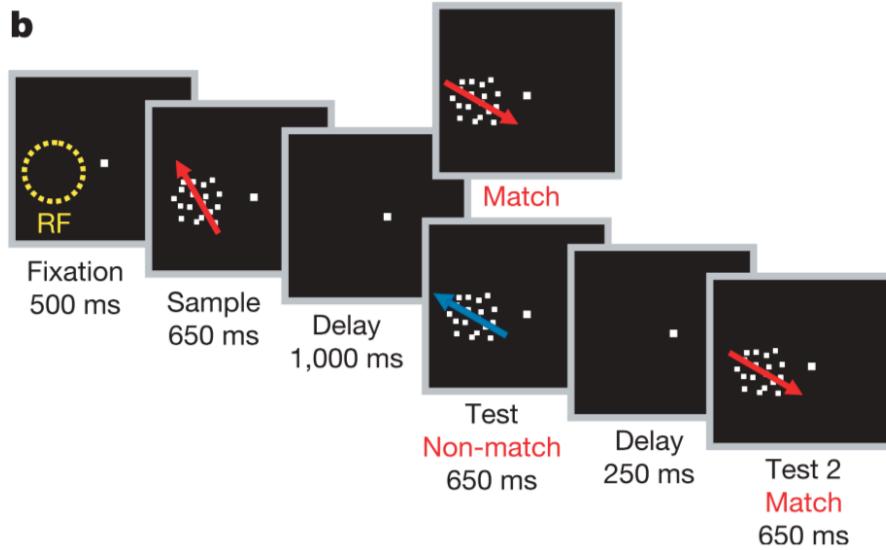


Figure 2: A schematic of the task described below. The stimuli are coherent patches of dot motion and RF indicates the receptive field of a recorded neuron. Taken from Freedman and Assad (2006).

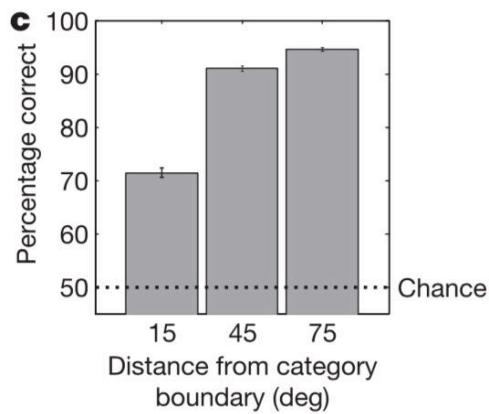


Figure 3: The average performance of the two subjects on the DMC. Taken from Freedman and Assad (2006).

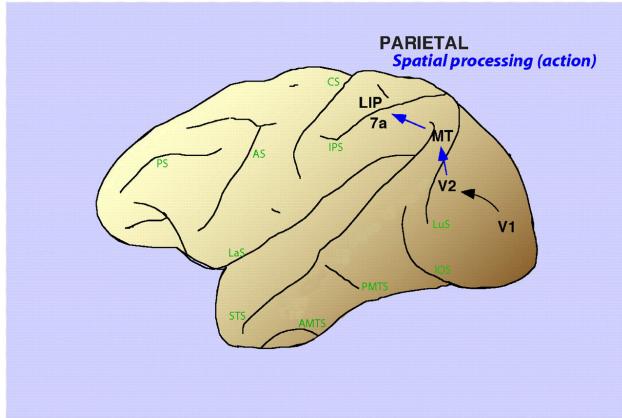


Figure 4: A schematic of the canonical dorsal visual stream overlaid on the Rhesus macaque brain. Adapted from Sereno and Lehky (2007).

We have in hand our task that requires the process that we are interested in, the categorization of motion direction of the dot patch. But, now, how do we study this in the brain? Or, perhaps more immediately, *where* do we study it in the brain?

The brain regions of interest

The middle temporal area (MT) is believed to be the first higher brain area with robust representations of visual motion. MT neurons display “tuning” to motion direction, meaning that they have a preferred direction that they respond to most vigorously, and lesser responses to other directions. By monitoring the responses of many MT neurons, one can read out the direction of the visual stimulus. MT is also part of the canonical dorsal visual stream, which is thought to primarily underly the allocation of visual attention and spatial processing. MT has been shown to be necessary to perform a similar categorization task, however, MT is thought to provide a more sensory (or literal) representation of motion direction, rather than representing more abstract category membership, so it can't be the end of the road for this neural computation.

The next brain area after MT in the dorsal stream is posterior parietal cortex (PPC) a complex of regions including the lateral intraparietal area (LIP). LIP is known to display a range of representations of task variables including those aligned with a subject's upcoming behavioral choice and with memories of past visual stimuli.

How does representation of visual motion while a subject is performing our categorization task differ between MT and LIP? What are some hypotheses that one could test?

Spiking: the substrate of neural communication

Neurons in the brain communicate through the firing of action potentials, a short (~1ms long) spike in electrical depolarization that travels down the axon and releases a chemical signal at the synapse that opens ion channels in the downstream cell.

Understanding how the brain functions involves catching this electrical activity in action and recording from neurons while the subject receives sensory inputs, makes decisions, and reports those choices behaviorally.

One of the most common neural recording techniques is called extracellular electrophysiology, which can be performed in awake and behaving subjects. With this technique, an experimentalist is able to record single spikes from single neurons in an anatomically targeted region of the brain. While techniques exist now for

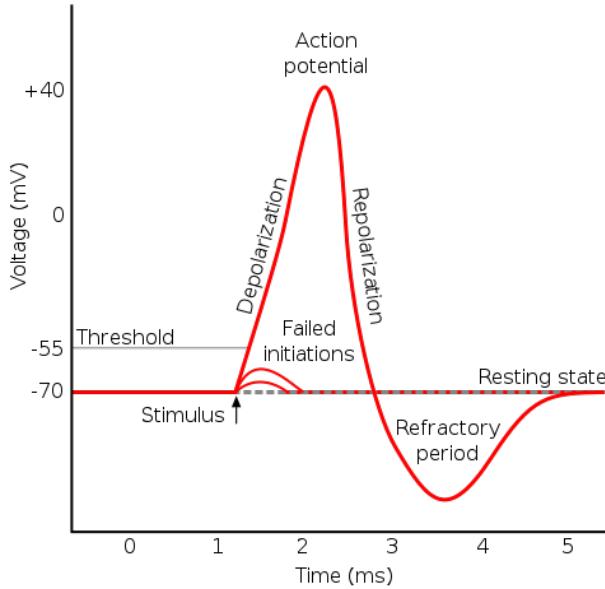


Figure 5: A schematic of an action potential or 'spike' from a neuron.

recording modestly sized groups of neurons (e.g. up to 100 or 1000), the dataset we are using today is from 2006, and all of the recordings are from single neurons, recorded independently of one another.

This technique has high temporal resolution (i.e., we are recording single spikes) and high spatial resolution (i.e., we are recording from single cells), and current engineering efforts focus on attaining this kind of space-time resolution while recording from tens of thousands of cells, or perhaps the entire brain of a smaller organism. Here, we have the benefit of decades of neuroanatomy that let's us pinpoint regions in the brain that display evidence of a concrete neural computation in the response of single cells.

We will now examine spikes from a many neurons in both MT and LIP. We want to understand how their representations of visual motion direction and (potentially) visual motion category differ. Let's get started!

The spiking of an MT neuron

First, we need to load our dataset, which has been saved in the MatLab data format (for which we can use the library "rmatio" to import into R). Let's start by loading the data for one particular cell:

```
library(rmatio)

mt_neuron_folder <- '../data/MT/'
mt_neuron_path <- '../data/MT/dseao01.mat'

mt_neuron <- read.mat(mt_neuron_path)
```

Now, let's examine what's happening in this data structure:

```
str(mt_neuron)

# List of 3
# $ trial_raster : num [1:246, 1:2700] 0 0 0 0 0 0 0 0 0 ...
# $ samp_direction_this_trial: num [1:246] 150 180 240 90 90 300 240 210 60 30 ...
# $ test_direction_this_trial: num [1:246] 210 330 240 0 210 180 300 90 30 150 ...
```

We can see that the data is in a named list format and the list has three elements:

- **trial_raster**: An array of dimensions 246 and 2700. This is the spike “raster” for our MT neuron. There are 246 trials and each trial is segmented into 2700 bins, each 1 ms in size. The value of each bin is 0 if no spike was recorded from this neuron at that time point, and 1 if a spike was recorded. The visual motion stimulus appears 500 ms into the trial, at element 500.

- **samp_direction_this_trial**: This is the direction in degrees of the first visual motion stimulus. Since MT neurons are thought to be tuned to visual motion, we might expect them to vary their responses with changes in direction.

- **test_direction_this_trial**: We will largely ignore this here, but it is the direction of the second visual motion stimulus.

We can begin to visualize the response of this neuron by creating a “raster plot” where the times of spikes in each trial are marked (e.g. by a dash or circle), and trials are aligned and stacked along the y-axis.

Exercise 1: Make a raster plot for the MT neuron we loaded earlier.

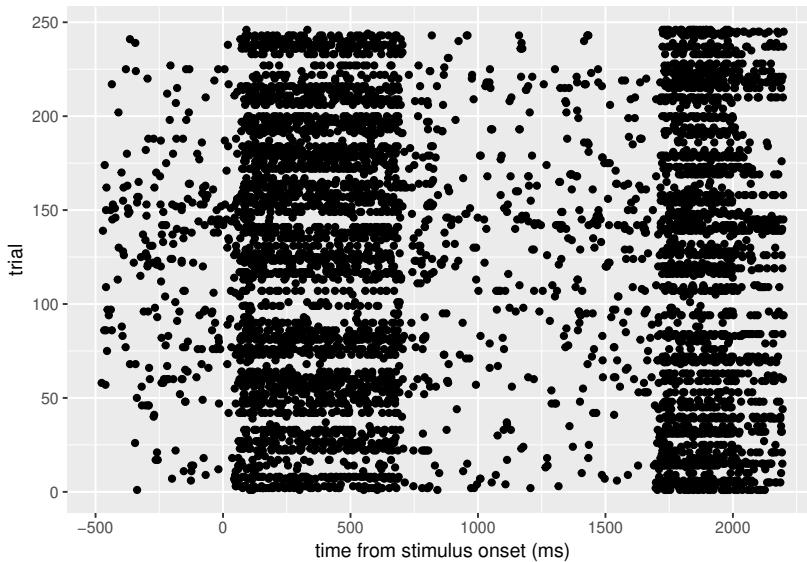
```
library(ggplot2)

# the stimulus starts 500ms after the beginning of each trial
stim_onset = 500

mt_raster <- mt_neuron[['trial_raster']]

# Find all the (time bin, trial) pairs in which there were spikes.
# We'll use the `which` function to find the values where the raster
# contains spikes (a value of '1' in the raster matrix), and we'll use
# `arr.ind` (which means 'array indices') to report back where in the
# matrix those values of '1' were found.

spkpairs = which(mt_raster == 1, arr.ind=TRUE)
qplot(spkpairs[, 2] - stim_onset, spkpairs[, 1],
      xlab='time from stimulus onset (ms)',
      ylab='trial')
```



This gives us the overall structure of the trial. We get a response to the first or “sample” stimulus, relatively less firing during the delay period, followed by a strong response to the second, or “test” stimulus.

In each trial, a different motion direction was shown, and these were pseudo-randomly interleaved (i.e. not in

any particular order in direction space). Let's pull out the trials for one particular direction of motion during the sample period and take a look at that activity. First we can find out which directions were played to the neuron in the experiment (in degrees).

```
dirs <- mt_neuron[['samp_direction_this_trial']]

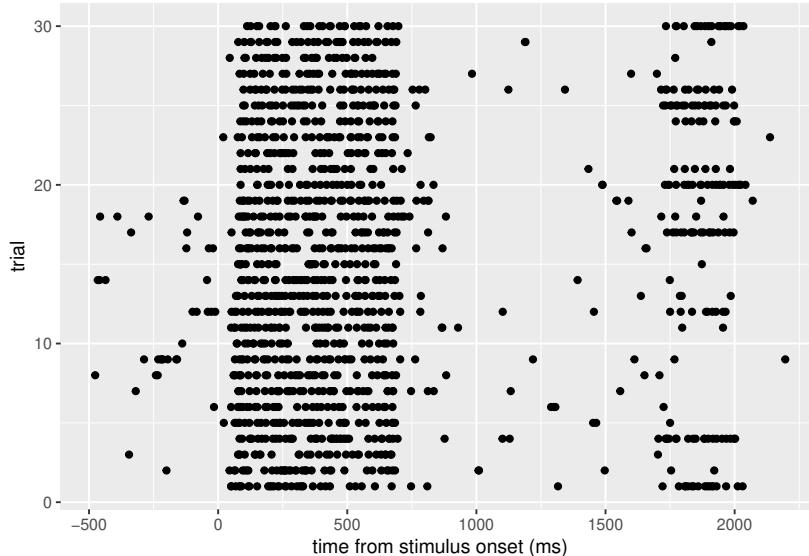
unique_dirs <- unique(dirs)
print(unique_dirs)

# [1] 150 180 240 90 300 210 60 30 270 330 120 0
```

Next, let's choose the 240 degree direction and plot just those trials. You can change the direction you choose to plot by modifying "choose_dir" in the code below.

```
choose_dir <- 240

dir_raster <- mt_raster[dirs == choose_dir,]
dir_spks <- which(dir_raster == 1, arr.ind=TRUE)
qplot(dir_spks[, 2] - stim_onset, dir_spks[, 1],
      xlab='time from stimulus onset (ms)',
      ylab='trial')
```



We would expect, based on what we learned about area MT, that the neuron will be tuned to its particular favorite direction... so, let's create a spike raster sorted by motion direction!

Exercise 2: Color the raster plot by the motion direction of the first or "sample" period direction. You'll want to grab the sample directions for each trial from the data structure `mt_neuron` and its component `samp_direction_this_trial`. You can try coloring each trial by its direction first, then sort those directions into clusters so you can plot them in contiguous rows.

There are many ways to create this plot, but here is some code that succinctly creates a nice colored raster plot, using the `factor` function in R:

```
sort_dirs <- sort(dirs, index.return=TRUE)
order_inds <- sort_dirs$ix
s_dirs <- sort_dirs$x

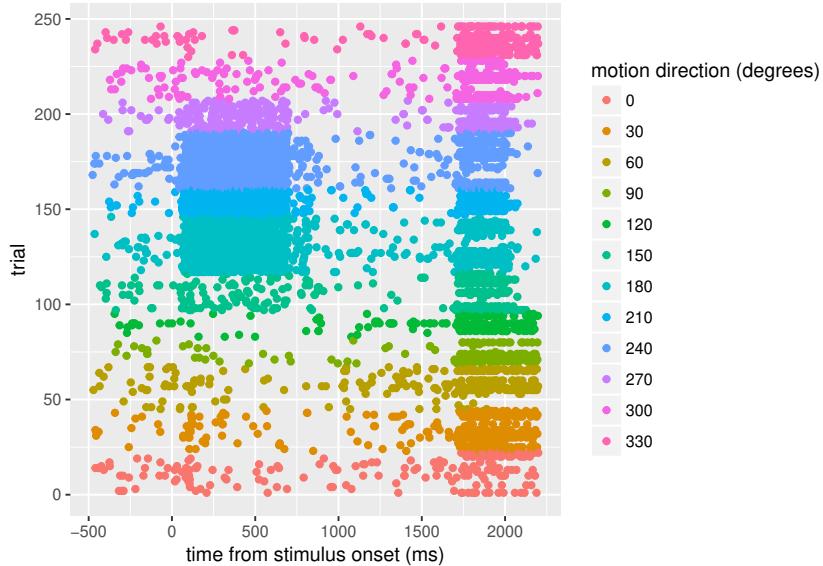
ordered_raster <- mt_raster[order_inds, ]
sorted_spkpairs <- which(ordered_raster == 1, arr.ind=TRUE)
```

```

dfact <- factor(s_dirs[sorted_spkpairs[, 1]], levels=unique(s_dirs))

qplot(sorted_spkpairs[ , 2] - stim_onset, sorted_spkpairs[ , 1],
      color=dfact, xlab='time from stimulus onset (ms)',
      ylab='trial') +
  labs(color='motion direction (degrees)')

```



This neuron appears to be strongly tuned to motion around 210 and 240 degrees. Let's summarize this by taking some averages.

Exercise 3: Use the R code in this workshop's directory to plot the average number of spikes (across trials) for a particular motion direction, as a function of the time within the trial. Spike times are measured with millisecond precision, so you may wish to create a smoothed average of these spike times across trials, to get a clearer picture of the shape of the neuron's response in time. Neuroscientists call this the “peri-stimulus time histogram” or PSTH.

Let's start to pull together some of the useful, and reusable, code elements we've developed. We're going to define a function that creates a PSTH plot for a given neural dataset, with a particular choice of smoothing window. First, we'll create a function that reads in the data, then we'll create another function that takes those data and plots them. We'll use a boxcar or moving average smoothing window, and we'll send the size of the smoothing window to the PSTH plotting function. We'll spell out the `read_neuron` function here. You can check out the contents of `plot_PSTH_neuron` in the `code` folder.

```

read_neuron <- function(filename) {
  neuron <- read.mat(filename)
  name <- tail(strsplit(filename, '/')[[1]], n=1)
  xs = 1:dim(neuron[['trial_raster']])[2]
  dirs <- neuron[['samp_direction_this_trial']]
  sort_dirs <- sort(dirs, index.return=TRUE)
  order_inds <- sort_dirs$ix
  dirs <- sort_dirs$x
  udirs <- sort(unique(dirs))
  raw_spks <- neuron[['trial_raster']]
  spks <- ts(t(raw_spks[order_inds, ]), xs[1], xs[length(xs)],
             frequency=1)
  neur <- list(name=name, dirs=dirs, udirs=udirs, spks=spks)
}

```

```
neur  
}
```

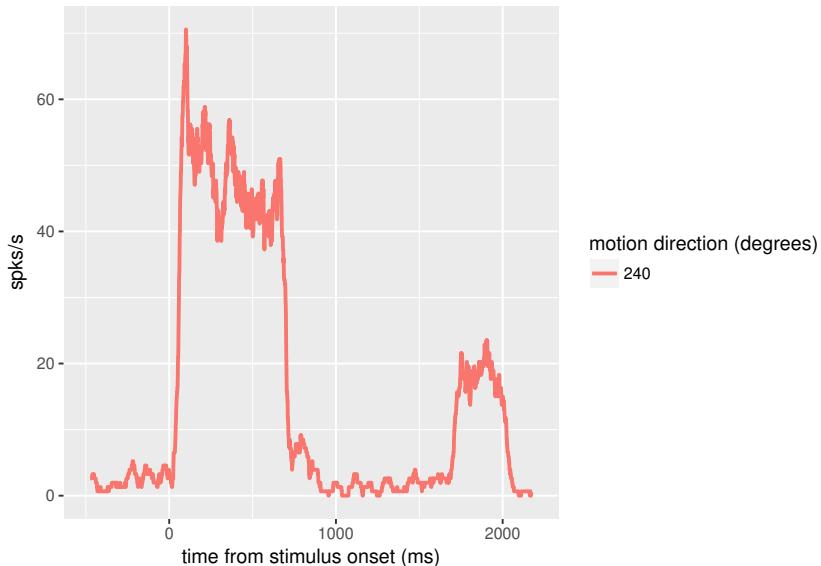
Next, let's let R know we'd like to load the function `plot_PSTH_neuron` so that we can use it here in our console. While we're at it, let's source the `read_neuron` function, which is also saved in the code directory for this workshop. There are actually a whole bunch of functions we wrote for this workshop, so let's run `source_neuro_workshop_functions.R` to do everything at once!

```
source('source_neuro_workshop_functions.R')
```

...and now let's test the plotting function by setting a filter size of 51ms and sending it the MT neuron we've been using so far in this workshop, for the 240 degree motion direction.

```
filter_size <- 51  
plot_PSTH_neuron(mt_neuron_path,filter_size,240)
```

```
# Warning: Removed 50 rows containing missing values (geom_path).
```



This function is set up so that if you do not specify which motion direction or directions you'd like to plot, it will plot ALL directions. Make a plot of the average number of spikes emitted over time in the experiment, for all motion directions.

Exercise 4: Explore different sizes of smoothing windows. How much smoothing is appropriate? What do we lose by using more smoothing? What do we gain?

Tuning curves

You will notice that the neuron's response is *approximately* constant in each trial part: the “sample” period, the “delay” period, and the “test” period. To further summarize the neuron's response, we can plot the average number of spikes in a large window within one of these trial periods, as a function of the direction of the stimulus during that period. Now that you're familiar with the data structure, we've created some useful functions that will help you pull out and plot average spike counts from particular parts of the trial.

The function you will find most useful is `get_direction_tuning`. This takes as arguments: the path to the data file we'd like to open, the time in the trial to begin counting spikes, the time in the trial to end counting spikes, and the direction to put at the center of the tuning plot.

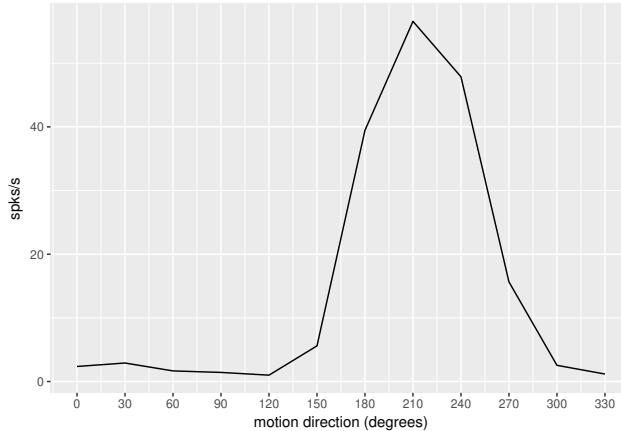
An example call to `get_direction_tuning` is:

```

count_start <- 100
count_end <- 600

get_direction_tuning(mt_neuron_path, count_start + stim_onset,
                     count_end + stim_onset)

```



Exercise 5: Use `get_direction_tuning` to plot the average number of spikes our original MT neuron emits between 100ms and 600ms, as a function of the direction of motion of the stimulus. Cool! This neuron has strongly peaked responsiveness for a particular motion direction. Experiment with different spike counting windows (i.e. change `count_start` and `count_end`), and see how this changes the tuning curve. Refer back to the PSTH plot to examine different regions of interest.

Category 1 runs from 135 to 315 degrees, so we can shift our data (slightly) to place 135 degrees in the center of the x-axis. The `get_direction_tuning` function circularly shifts the data so that the center of the direction axis is between two stimulus directions. You can set this to be the category boundary, so you can see how ‘categorical’ an individual neuron’s response looks. A perfect category neuron would respond equally and strongly to all directions within a category and not at all to directions in the other category. A perfectly direction tuned neuron would respond to only one motion direction and no others. Let’s take a look at data from the other brain area we’re interested in, LIP!

The spiking of an LIP neuron

LIP stands for the lateral interparietal area of the brain and it codes for a variety of task variables, particularly those relating to visual motion and eye movements. Area LIP sits downstream of area MT in the visual motion pathway, and upstream of the motor areas that control eye movements. In this part of the workshop, we’ll explore how LIP neurons respond during the delayed match-to-same task, and compare those responses to the MT neuron we started with.

```

category_bound <- 135
lip_neuron_folder <- '../data/LIP/'
lip_neuron_name <- 'dhbbg05.mat'
lip_neuron_path <- paste(lip_neuron_folder, lip_neuron_name,
                         sep=' ')
plot_PSTH_neuron(lip_neuron_path, filter_size)

```

Exercise 6: Plot the tuning curve for this LIP neuron during the sample period, then for the delay period (from, say, 750ms to 1500ms).

```

count_start <- 100
count_end <- 600
category_bound <- 135
get_direction_tuning(lip_neuron_path, count_start + stim_onset,
                     count_end + stim_onset, category_bound)

```

This response during the sample period looks a bit more categorical than the MT neuron we started with, but perhaps is tuned to motion at 300 degrees. When we have a look at the delay period, the time in between the presentation of the first sample motion direction and the second, “test”, motion direction, however, we see something more interesting.

```

count_start <- 750
count_end <- 1500
get_direction_tuning(lip_neuron_path, count_start + stim_onset,
                     count_end + stim_onset, category_bound)

```

Now we see something striking in LIP! The response of our MT neuron was mostly zero during this same delay period. Our LIP neuron is computing something different about the stimulus, something that more closely reflects the abstract category boundary we, as experimental designers, cooked up. The brain of the subject has formed a representation of this arbitrary boundary and we can “see” it. Next up, we’ll make this more quantitative.

Comparing MT to LIP

MT neurons don’t have much activity during this delay period, so to compare the two neural responses, we’ll start with an analysis window that’s the same for both. We’ll use the sample period:

```

count_start <- 100
count_end <- 600

```

Exercise 7: Plot the tuning curves during the sample period defined above for a handful of MT neurons. You may decide to choose them at random. Make plots for at least 5 neurons.

```

mt_neuron_names <- list.files(mt_neuron_folder)

## take a random subset from MT too!
plots <- 5
mt_samp <- sample(mt_neuron_names, plots)
for (n in mt_samp) {
  neur_path <- paste(mt_neuron_folder, n, sep=' ')
  f <- get_direction_tuning(neur_path, count_start + stim_onset,
                           count_end + stim_onset, category_bound)
  print(f)
}

```

Exercise 8: Plot the tuning curves during the sample period defined above for a handful of LIP neurons. How do the tuning curves compare? Can you uncover any systematic differences between the populations? What do you notice if you try plotting the LIP tuning curves different trial periods, like the delay period?

```

lip_neuron_names <- list.files(lip_neuron_folder)

## take a random subset of 5 of these data files
plots <- 5
lip_samp <- sample(lip_neuron_names, plots)

```

```

for (n in lip_samp) {
  neur_path <- paste(lip_neuron_folder, n, sep=' ')
  f <- get_direction_tuning(neur_path, count_start + stim_onset,
                            count_end + stim_onset, category_bound)
  print(f)
}

```

Quantifying category tuning

Okay, it's difficult to draw quantitative conclusions just by squinting at all of these plots. Some plots from MT and LIP look similar – others look different! We need to find a way to summarize the category and direction tuning of each.

In the paper, the authors develop a measure of within-category difference (WCD; how much heterogeneity in response does a neuron have for members of the same category?) as well as a between-category difference (BCD; how much heterogeneity is there across categories?). Using both of these, to construct a single index, written as:

$$CI = \frac{BCD - WCD}{WCD + BCD}$$

which ranges from $[-1, 1]$ with positive values indicating a neuron is more modulated by changes in motion direction across the category boundary than by changes in motion direction within a category – thus, more categorically tuned.

Looking at this index across our MT and LIP populations will tell us whether they tend to represent the categories particularly or direction uniformly. We've written a function that will compute the category index for you. You can take a look under the hood if you're interested in how to implement this in R. For now, let's just use the function to have a look at the results. First, we need to `source` all the functions we'll call from it, then we'll run the function itself.

```

get_category_index(lip_neuron_path, category_bound,
                   count_start + stim_onset, count_end + stim_onset)

# [1] 0.2278473

```

That's the category tuning index for one LIP neuron. Now, let's compute the distribution of category indices for our MT and LIP populations. We've created some functions to help you compute these numbers across all of the data in the data folder. The `get_ci_dist` function evaluates the category tuning index for all neurons in the specified folder. The syntax is illustrated here.

```

lip_ci <- get_ci_dist(lip_neuron_folder, category_bound,
                       count_start + stim_onset, count_end + stim_onset)

mt_ci <- get_ci_dist(mt_neuron_folder, category_bound,
                      count_start + stim_onset, count_end + stim_onset)

```

The vectors `lip_ci` and `mt_ci` now contain all of the category indices for the 67 MT neurons in our data folder and the 92 LIP neurons. Now, let's plot the results.

Exercise 9: Make plots of the distribution of category tuning indices for the MT neurons and for the LIP neurons. How do these compare? Is either population more category tuned than the other?

We plotted these distributions using a few R tricks you might find useful.

```
library(plyr)
```

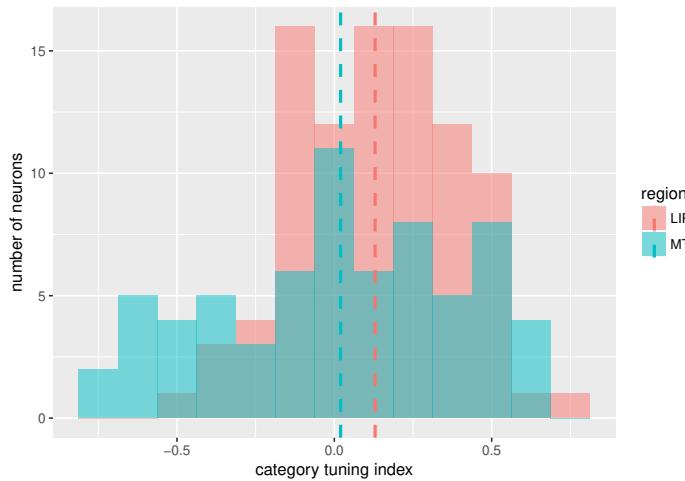
```

d <- data.frame(region=factor(c(rep("LIP", length(lip_ci)),
                                rep("MT", length(mt_ci)))),
                 ci=c(lip_ci, mt_ci))

mean_dat <- ddply(d, "region", summarise, ci.mean=mean(ci))

ggplot(d, aes(x=ci, fill=region)) +
  geom_histogram(binwidth=.125, alpha=.5, position="identity") +
  geom_vline(data=mean_dat, aes(xintercept=ci.mean, colour=region),
             linetype="dashed", size=1) +
  xlab('category tuning index') +
  ylab('number of neurons')

```



So, it looks like LIP does have a significant bias toward category tuning. But let's test that:

```

str(c('LIP, p-value ', t.test(lip_ci)$p.value))
str(c('MT, p-value ', t.test(mt_ci)$p.value))

```

```

# chr [1:2] "LIP, p-value " "2.45817971663735e-06"
# chr [1:2] "MT, p-value " "0.677607532282032"

```

Now let's publish this in Nature! ...at least, that's what happened back in 2006, modulo some other controls and elaborations. Congrats! You've just worked through a major result in neuroscience that displays how a neural computation evolves from a stimulus centered representation in one brain area (area MT) to a more abstract categorical representation in the next brain area (area LIP).

Bonus: an important control

But, wait, what if our category boundary just happened to reflect some innate bias in LIP direction representations? That would be weird, but it's not impossible. Let's retrain the subjects on a new category boundary and then check that (a) we no longer have significant category tuning for the old categories and (b) we do have significant category tuning for the new boundary.

```

lip2_neuron_folder <- '../data/LIP2/'
category_bound2 <- ensure_angle(category_bound + 90)

lip2_ci_perp <- get_ci_dist(lip2_neuron_folder, category_bound,
                             count_start + stim_onset,

```

```

        count_end + stim_onset)

lip2_ci_cong <- get_ci_dist(lip2_neuron_folder, category_bound2,
                           count_start + stim_onset,
                           count_end + stim_onset)

```

These distributions are, by definition, negatives of each other – but we can plot them anyway just to see:

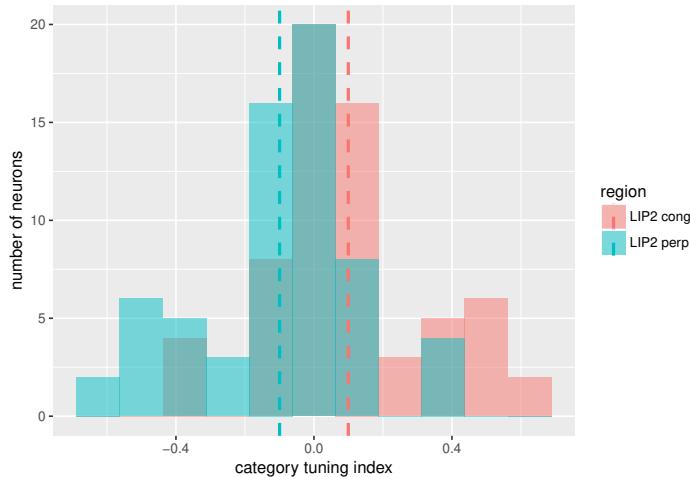
```

d <- data.frame(region=factor(c(rep("LIP2 perp", length(lip2_ci_perp)),
                                rep("LIP2 cong", length(lip2_ci_cong))),
                                ci=c(lip2_ci_perp, lip2_ci_cong)))

mean_dat <- ddply(d, "region", summarise, ci.mean=mean(ci))

ggplot(d, aes(x=ci, fill=region)) +
  geom_histogram(binwidth=.125, alpha=.5, position="identity") +
  geom_vline(data=mean_dat, aes(xintercept=ci.mean, colour=region),
             linetype="dashed", size=1) +
  xlab('category tuning index') +
  ylab('number of neurons')

```



```

str(c('LIP original bound, p-value ', t.test(lip2_ci_perp)$p.value))
str(c('LIP new bound, p-value ', t.test(lip2_ci_cong)$p.value))

```

```

# chr [1:2] "LIP original bound, p-value " "0.00151062222812236"
# chr [1:2] "LIP new bound, p-value " "0.00151062222812236"

```

Significant category tuning for the new boundary, and (of course) significant tuning against the old boundary! LIP really does appear to be learning and robustly representing these arbitrary motion direction boundaries.

This is one step in the processing of a “raw” sensory representation into the kind of categorical representation that we all use for making decisions!

Population genetics workshop

Instructor: **Matthias Steinrücken**

Course Assistant: **Zach Miller**

Welcome

This exercise is going to expose you to several basic ideas in probability and statistics as well as show you the utility of using R for basic statistical analyses. We'll do so in the context of a basic population genetic analysis.

The scenario

As a biologist, you will learn what are the major patterns that are expected when the data you work with is clean. Using that expertise will save you from the mistake of misinterpreting error-prone data. In population genetics, there are a number of patterns that we expect to see immediately in our datasets. In this exercise you will explore one of those major patterns. Rather than give it away — let's begin some analysis and see what we find. In the narrative that follows, we'll refine our thinking as we go.

Introductory terminology

- Single-nucleotide polymorphism (SNP): A nucleotide basepair that is *polymorphic* (i.e. it has multiple types or *alleles* in the population)
- Allele: A particular variant form of DNA (e.g. A particular SNP may have the “A-T” allele in one DNA copy and “C-G” in another; We typically define a reference strand of the DNA to read off of, and then denote the alleles according to the reference strand base - so for example, these might be called simply the “A” and “C” alleles. In many cases we don’t care about the precise base, so we might call these simply the A_1 and A_2 alleles, or the A or a alleles, or the 0 and 1 alleles.)
- Minor allele: The allele that is more rare in a population
- Major allele: The allele that is more common in a population
- Genotype: The set of alleles carried by an individual (E.g. AA, AC, CC; or AA, Aa, aa; or 0, 1, 2)
- Genotyping array: A technology based on hybridization with probes and florescence that allows genotype calls to be made at 100s of thousands of SNPs per individual at an affordable cost.

The data-set and basic pre-processing

We will look at Illumina 650Y genotyping array data from the CEPH-Human Genome Diversity Panel. This sample is a global-scale sampling of human diversity with 52 populations in total.

The data were first described in Li et al (Science, 2008) and the raw files are available from the following link: <http://hgsc.org/hgdp/files.html>. These data have been used in numerous subsequent publications (e.g Pickrell et al, Genome Research, 2009) and are an important reference set. A few technical details are that the genotypes were filtered with a GenCall score cutoff of 0.25 (a quality score generated by the basic genotype calling software). Individuals with a genotype call rate <98.5% were removed, with the logic being that if a sample has many missing genotypes it may due to poor quality of the source DNA, and so none of the genotypes should be trusted. Beyond this, to prepare the data for the workshop, we have filtered down the individuals to a set of 938 unrelated individuals. (For those who are interested, the data are available as plink-formatted files `H938.bed`, `H938.fam`, `H938.bim` from this link: <http://bit.ly/1aluTln>). We have also extracted the basic counts of three possible genotypes.

The files with these genotype frequencies are your starting points.

Note about logistics

We will use some functions from the `dplyr` and `ggplot2` and `reshape2` libraries so first let's load them:

```
library(dplyr)
library(ggplot2)
library(reshape2)
```

If you get an error message saying there is no package titled `dplyr`, `ggplot2`, or `reshape2` you may need to first run `install.packages("dplyr")`, `install.packages("ggplot2")`, or `install.packages("reshape2")` to install the appropriate package.

We will not be outputting files - but you may want to set your working directory to the `sandbox` sub-directory in case you want to output some files.

The `MBL_WorkshopJN.Rmd` file has the R code that you can run. Code is provided for most steps, but some you will need to devise for yourselves to answer the questions that are part of the workshop narrative.

Initial view of the data

Read in the data table:

```
g <- read.table("../data/H938_chr15.geno", header=TRUE)
```

It will be read in as a dataframe in R.

Then use the “head” command to see the beginning of the dataframe:

```
head(g)
```

You should see that there are columns each with distinct names.

CHR SNP A1 A2 nA1A1 nA1A2 nA2A2

- CHR: The chromosome number. In this case all SNPs are from chromosome 15.
- SNP: The rsid of a SNP is a unique identifier for a SNP and you can use the rsid to look up information about a SNP using online resource such as dbSNP or SNPedia.
- A1: The minor allele at the SNP.
- A2: The major allele.
- nA1A1 : The number of A1/A1 homozygotes.
- nA1A2 : The number of A1/A2 heterozygotes.
- nA2A2 : The number of A2/A2 homozygotes.

Calculate the number of observations at each locus

Next compute the total number of observations by summing each of the three possible genotypes. Here we use the `mutate` function from the `dplyr` library to do the addition and add a new column to the dataframe in one nice step. (Note: You could also use the `colSums` function from the base R library).

```
g <- mutate(g, nObs = nA1A1 + nA1A2 + nA2A2)
```

Run `head(g)` and confirm your dataframe `g` has a new column called `nObs`.

Now use the `summary` function to print a simple summary of the distribution:

```
summary(g$nObs)
```

The `ggplot2` library has the ability to make “quick plots” with the command `qplot`. If we pass it a single column it will make a histogram of the data for that column. Let’s try it:

```
qplot(nObs, data = g)
```

Our data are from 938 individuals. When the counts are less than this total, it's because some individuals had array data that was difficult to call a genotype for and so no genotype was reported.

Question: Do most of the SNPs have complete data?

Question: What is the lowest count observed? Is this number in rough agreement with what we know about how the genome-wide missingness rate filter was set to >98.5% of all SNPs?

Calculating genotype and allele frequencies

Let's move on to calculating genotype and allele frequencies. For allele A_1 we will denote its frequency among all the samples as p_1 , and likewise for A_2 we will use p_2 .

```
# Compute genotype frequencies
g <- mutate(g, p11 = nA1A1/nObs , p12 = nA1A2/nObs, p22 = nA2A2/nObs )
# Compute allele frequencies from genotype frequencies
g <- mutate(g, p1 = p11 + 0.5*p12, p2 = p22 + 0.5*p12)
```

Question: With a partner or group member, discuss whether the equations in the code for p_1 and p_2 are correct and if so, why?

Run `head(g)` again and confirm that `g` now has the extra columns for the genotype and allele frequencies.

And let's plot the frequency of the major allele (A2) vs the frequency of the minor allele (A1). The `ggplot2` library has the ability to make "quick plots" with the command `qplot`. Let's try it here:

```
qplot(p1, p2, data=g)
```

Notice that $p_2 > p_1$ (be careful to inspect the axes labels here) This makes sense because A_1 is supposed to be the minor (less frequent) allele. Note also that there is a linear relationship between p_2 and p_1

Question: What is the equation describing this relationship?

The relationship exists because there are only two alleles - and so their proportions must sum to 1. The linear relationship you found exists because of this constraint. It also provides a nice check on our work (if p_1 and p_2 didn't sum to 1 it would suggest something is wrong with our code!).

Plotting genotype on allele frequencies

Let's look at an initial plot of genotype vs allele frequencies. We could use the base plotting functions, but the following uses the `ggplot2` commands. These are a little trickier, but end up being very compact (we need fewer lines of code overall to achieve our desired plot). To use `ggplot2` commands effectively our data need to be what statisticians call "tidy" (in this case, that means with one row per pair of points we will plot).

To do this, we first subset the data on the columns we'd like (using the `select` command and listing the set of columns we want), then we pass this (using the `%>%` operator) to the `melt` command which will reformat the data for us, and output it as `gTidy`:

```
gTidy <- select(g, c(p1,p11,p12,p22)) %>% melt(id='p1',value.name="Genotype.Proportion")
head(gTidy)
ggplot(gTidy) + geom_point(aes(x = p1,
                                y = Genotype.Proportion,
                                color = variable,
                                shape = variable))
```

Now let's look at the graph that we produced. There is some scatter in the relationship between genotype proportion and allele frequency for any given genotype, but at the same time there is a very regular underlying relationship between these variables.

Question: What are approximate relationships between p_{11} vs p_1 , p_{12} vs p_1 , and p_{22} vs p_1 ? (Hint: These look like parabolas, which suggests are some very simple quadratic functions of p_1).

You might start to recognize that these are the classic relationships that are taught in introductory biology courses. If you recall, under assumptions that there is no mutation, no natural selection, infinite population size, no population substructure and no migration, then the genotype frequencies will take on a simple relationship with the allele frequencies. That is: $p_{11} = p_1^2$, $p_{12} = 2p_1(1 - p_1)$ and $p_{22} = (1 - p_1)^2$. In your basic texts, they typically use p and q for the frequencies of allele 1 and 2, and present these *Hardy-Weinberg proportions* as: p^2 , $2pq$, and q^2 .

Another way to think of the Hardy-Weinberg proportions is in the following way. If the state of an allele (A_1 vs A_2) is *independent* within a genotype, then the probability of a particular genotype state (such as A_1A_1) will be determined by taking the product of the alleles within it (so $p_{11} = p_1p_1$ or p_1^2).

Let's add to the plot lines that represent Hardy-Weinberg proportions:

```
ggplot(gTidy) +
  geom_point(aes(x=p1,y=Genotype.Proportion,color=variable,shape=variable)) +
  stat_function(fun=function(p) p^2, geom="line", colour="red",size=2.5) +
  stat_function(fun=function(p) 2*p*(1-p), geom="line", colour="green",size=2.5) +
  stat_function(fun=function(p) (1-p)^2, geom="line", colour="blue",size=2.5)
```

On average, the data follow the classic theoretical expectations fairly well. It is pretty remarkable that such a simple theory has some bearing on reality!

By eye, we can see that the fit isn't perfect though. There is a systematic deficiency of heterozygotes and excess of homozygotes. Why?

Let's look at this more closely and more formally...

Testing Hardy Weinberg

Pearson's χ^2 -test is a basic statistical test that can be used to see if count data o_i conform to a particular expectation. It is based on the X^2 -test statistic:

$$X^2 = \sum_i \frac{(o_i - e_i)^2}{e_i}$$

which follows a χ^2 distribution under the null hypothesis that the data are generated from a multinomial distribution with the expected counts given by e_i .

Here we compute the test statistic and obtain its associated p-value (using the `pchisq` function). We keep in mind that there is 1 degree of freedom (because we have 3 observations per SNP, but then they have to sum to a single total sample size, and we have to use the data once to get the estimated allele frequency, which reduces us down to 1 degree of freedom).

```
g <- mutate(g, X2 = (nA1A1-nObs*p1^2)^2 / (nObs*p1^2) +
  (nA1A2-nObs*2*p1*p2)^2 / (nObs*2*p1*p2) +
  (nA2A2-nObs*p2^2)^2 / (nObs*p2^2))
g <- mutate(g,pval = 1-pchisq(X2,1))
```

The problem of multiple testing

Let's look at the top few p-values:

```
head(g$pval)
```

How should we interpret these? A p-value gives us the frequency at which the observed departure from expectations (or a more extreme departure) would occur if the null hypothesis (the SNP follows HW proportions) is true. As an agreed upon standard (of the frequentist paradigm for statistical hypothesis testing), if the observation is relatively rare under the null (e.g. p-value < 5%), we reject the null hypothesis, and we would infer that the given SNP departs from Hardy-Weinberg expectations. This is problematic here though. The problem is that we are testing many, many SNPs (Use `dim(g)` to remind yourself how many rows/SNPs are in the dataset). Even if the null is universally true, 5% of our SNPs would be expected to be rejected using the standard frequentist paradigm. This is called the multiple testing problem. As an example, if we have 20,000 SNPs, that all obey the null hypothesis, we would on average naively reject the null for ~1000 SNPs based on the p-values < 0.05.

We clearly need some methods to deal with the “multiple testing problem”. Two frameworks are the Bonferroni approach and false-discovery-rate (FDR) approaches. We will not say more about these here. Instead, we will do two simple checks to see though if our data are globally consistent with the null.

First, let’s see how many tests have p-values less than 0.05. Is it much larger than the number we’d expect on average given the total number of SNPs and a 5% rate of rejection under the null?

```
sum(g$pval < 0.05, na.rm = TRUE)
```

Wow - we see many more. This is our first sign that although by eye these data show qualitative similarities to HW, statistically they are not fitting Hardy-Weinberg well enough.

Let’s look at this another way. A classic result from Fisher is that under the null hypothesis the p-values of a well-designed test should be distributed uniformly between 0 and 1. What do we see here?

```
qplot(pval, data = g)
```

The data show an enrichment for small p-values relative to a uniform distribution. Notice how the whole distribution is shifted towards small values - The data appear to systematically depart from Hardy-Weinberg.

Plotting expected vs observed heterozygosity

To understand this more clearly, let’s make a quick plot of the expected vs observed heterozygosity (the proportion of heterozygotes):

```
qplot(2*p1*(1-p1), p12, data = g) + geom_abline(intercept = 0,
                                                    slope=1,
                                                    color="red",
                                                    size=1.5)
```

Most of the points fall below the $y=x$ line. That is, we see a systematic deficiency of heterozygotes (and this implies a concordant excess of homozygotes). This general pattern is contributing to the departure from HW seen in the X^2 statistics.

Discussion: Population subdivision and departures from Hardy-Weinberg expectations

We might wonder why the departure from Hardy-Weinberg proportional is directional, in that, on average, we are seeing a deficiency of heterozygotes (and excess of homozygotes). One enlightening way to understand this is by thinking about what Sewall Wright (a former eminent University of Chicago professor) called “the correlation of uniting gametes”. To produce an A_1A_1 individual we need an A_1 -bearing sperm and an A_1 -bearing egg to unite. If these events were independent of each other, we would expect A_1A_1 individuals at the rate predicted by multiplying probabilities, that is, p_1^2 (an idea we introduced above). However, what if uniting gametes are positively correlated, in that an A_1 -bearing sperm is more likely to join with an

A_1 -bearing egg? In this case we will have more A_1A_1 individuals than predicted by p_1^2 , and conversely fewer A_1A_2 individuals than predicted by $2p_1p_2$. If our population is structured somehow such that A_1 sperm are more likely to meet with A_1 eggs, then we will have such a positive correlation of uniting gametes, and the resulting excess of homozygotes and deficiency of heterozygotes.

Given the HGDP data is from 52 sub-populations from around the globe, and alleles have some probability of clustering within populations, a good working hypothesis for the deficiency of heterozygotes in this dataset is the presence of some population structure.

While statistically significant, the population structure appears to be subtle in absolute terms — based on our plots, we have seen the genotype proportions are not wildly off from HW proportions.

Question: As an exercise, compute the average deficiency of heterozygotes relative to the expected proportion. This is the average of

$$\frac{2p_1(1-p_1) - p_{12}}{2p_1(1-p_1)}$$

What is this number for this data-set? A common “rule-of-thumb” for this deficiency in a global sample of humans is approximately 10%. Do you find this to be true from the data?

A ~10% difference between expected and observed seems pretty remarkable given these samples are taken from across the globe. It is a reminder that human populations are not very deeply structured. Most of the alleles in the sample are globally widespread and not sufficiently geographically clustered to generate correlations among the uniting alleles. This is because all humans populations derived from an ancestral population in Africa around 100-150 thousand years ago, which is relatively small amount of time for variation across populations to accumulate.

Finding specific loci that show large departure from Hardy-Weinberg proportions

Now, let's ask if we can find any loci that are wild departures from HW proportions. These might be loci that have erroneous genotypes, or loci that cluster geographically in dramatic ways (such that they have few heterozygotes relative to expectations).

To find these loci, we'll compute the same relative deficiency you computed above, but let's look at it per SNP. This number is referred to as F by Sewall Wright and has connections directly to correlation coefficients (advanced exercise: Try to work this out!). If we assume there is no inbreeding within populations, this number is an estimator of F_{ST} (a quantity that appears often in population genetics).

Let's add this value to our dataframe and plot how it's value changes across the chromosome from one end to another:

```
g <- mutate(g, F = (2*p1*(1-p1)-p12) / (2*p1*(1-p1)))
plot(g$F, xlab = "SNP number")
```

There are a few interesting SNPs that show either a very high or low F value.

Now, here's a trick. When a high or low F value is due to genotyping error, it likely only effects a single SNP. However, when there is some population genetic force acting on a region of the genome, it likely effects multiple SNPs in the region. So let's try to take a local average in a sliding window of SNPs across the genome, computing an average F over every 5 consecutive SNPs (in real data analysis we might use 100kb or 0.1cM windows).

The `stats::filter` command below calls the `filter` function from the `stats` library. The code instructs the function to take 5 values centered on a focal SNP, weighting them each by 1/5 and then taking the sum. In this way it produces a local average in a sliding window of 5 SNPs. Let's define the `movingavg` function and then make a plot of its values:

```

movingavg <- function(x, n=5){stats::filter(x, rep(1/n,n), sides = 2)}
avgF <- movingavg(g$F)
plot(avgF, xlab="SNP number")

```

Wow — there appears to be one large spike where the average F is approximately 60% in the dataset!

Let's extract the SNP id for the largest value, and look at the dataframe:

```

outlier=which (movingavg(g$F) == max(movingavg(g$F),na.rm=TRUE))
g[outlier,]

```

Question: Which SNP is returned? By inserting the rs id into the UCSC genome browser (<https://genome.ucsc.edu/>), and following the links, find out what gene this SNP resides near. The gene names should start with “SLC..” What gene is it?

Question: Carry out a literature search on this gene using the term “positive selection” and see what you find. It’s thought the high F value observed here is because natural selection led to a geographic clustering of alleles in this gene region. Discuss with your partners why this might or might not make sense.

Discussion: The outlier region

The region you’ve found is one of the most differentiated between human populations that is known. Notice in your literature search, how it is known to affect skin pigmentation and is thought to contribute to differences in skin pigmentation that are seen between human populations. Finding strong population structure for alleles that affect external morphological phenotypes is not uncommon when looking at other chromosomes. Some of the most differentiated genes that exist in humans are those that involve morphological phenotypes - such as skin pigmentation, hair color/thickness, and eye color (the genes OCA2/HERC2, SCL45A2, KITLG, EDAR all come to mind). Many of these are thought to have arisen due to direct or indirect effects of adaptation to local selective pressures (e.g. adaptation to varying levels of UV exposure, local pathogens, local diets, local mating preferences), though in most cases we still do not yet have a fully convincing understanding of their evolutionary histories. Regardless of the reasons, it is notable that many of the features that humans see externally in each other (i.e. the morphological differences) are controlled by genes that are outliers in the genome. At most variant SNPs, the patterns of variation are much closer to those of a single random mating population than they are at variant sites like EDAR. Put another way, a genomic perspective shows us many of the differences people see in each other are in a sense, just skin-deep.

Wrap-up

Modern population genetics has a lot of additional tools on its workbench, but here using relatively simple and classical ideas combined with genomic-scale data, we have been able to observe and interpret some major features of human genetic diversity. We have also revisited some basic concepts of probability and statistics such as independence vs correlation, the χ^2 test, and the problems of multiple testing. One remarkable thing we saw is that a very simple mathematical model based on assuming independence of alleles and genotypes can predict genotype proportions within ~10% of the true values. This gives us a hint of how simple mathematical models may be useful even in the face of biological complexity. Finally, we have gained more familiarity with R. We didn’t discuss how genotyping errors might create Hardy-Weinberg departures, but if we were doing additional analyses, we could use Hardy-Weinberg departures to filter them from our data. It’s common practice to do so, but with a Bonferroni correction and using data from within populations to do the filtering.

Follow-up activities

In the **data** folder, we are including data files that you can explore to gain more experience. These include global data for other chromosomes (`H938_chr*.geno`) and the same data but limited to European populations

(H938_Euro_chr*.geno). Here are a few suggested follow-up activities. It may be wise to split the activities across class members and reconvene after carrying them out.

Follow-up activity: Look at a chromosome from the European-restricted data - is the global deficiency in heterozygosity as strong as it was on the global scale? Before you begin, what would you expect to see?

Follow-up activity Using the European data, do you find any regions of the genome that are outliers for F on chromosome 2? Using genome browsers and/or literature searches, can you find what is the likely locus under selection for that region?

Follow-up activity: Using the global data or the European data, analyze other chromosomes – do you find other loci that show high F values?

References

Li, Jun Z, Devin M Absher, Hua Tang, Audrey M Southwick, Amanda M Casto, Sohini Ramachandran, Howard M Cann, et al. 2008. "Worldwide Human Relationships Inferred from Genome-Wide Patterns of Variation." *Science* 319 (5866): 1100–1104.

Pickrell, Joseph K, Graham Coop, John Novembre, Sridhar Kudaravalli, Jun Z Li, Devin Absher, Balaji S Srinivasan, et al. 2009. "Signals of Recent Positive Selection in a Worldwide Sample of Human Populations." *Genome Research* 19 (5): 826–37.