Introduction to Probability and Statistics

2023/24

Stephen Connor

Table of contents

Overview

Welcome to IPS!

This web site is used to provide some of the course materials, and should be used alongside the module's Moodle page. All of the written assignment submission points can be found on Moodle, along with the quizzes for completion as you work through the computer labs.

You only *need* to use this site to access the computer lab material. However, you will also be able to access copies of the written assignments here in **html** format, in case you find that more accessible than the **pdf** files which will be available on Moodle.

> **Note**
>
> You can access the pdf version of any page of this site by clicking on the pdf icon in the left-hand menu. You can also choose to view the page in **dark mode**, if that's more comfortable.

Computer labs

The goal of these labs is to introduce you to, and build up your proficiency with, R and RStudio. You'll be using these throughout the course, both to learn the statistical concepts discussed in the lectures and also to analyze real data and come to informed conclusions. To straighten out which is which:

- R is the name of the programming language itself;
- RStudio is a convenient interface.

The R language is the standard statistical tool used by most statisticians at universities. One reason data scientists and statisticians like to use R is that all known statistical techniques are available in R. Whenever someone develops a new statistical technique, one of the first things they do is produce an R package so that the technique becomes available in R. The reason they do this for R rather than for one of the commercial alternatives is that R is open source and freely available to all, and of course that the previous methods on which the new method builds are already available in R.

Feeling comfortable using R is not only important for this module and any further statistics modules you may take at the Department of Mathematics of the University of York, it can also be an important factor for your future career (see the article "R skills attract the highest salaries". Even though R is specially designed for statistics, it is consistently in the list of the top ten most important programming languages compiled by the IEEE spectrum magazine.

As the labs progress, you are encouraged to explore beyond what the labs dictate; a willingness to experiment will make you a much better programmer.

Assessment

> **!** Important
>
> The five main labs (imaginatively named "Lab 1" to "Lab 5") **count for credit**: your best 4 out of 5 will marks will count for 20% of the module mark.

Each lab will have an accompanying **Moodle quiz**. As you work through each lab you will find places where you are asked to perform a calculation and then enter your mark in the appropriate quiz.

> **⚠** Warning
>
> The online quizzes will give you immediate feedback and allow you to try again if you get an answer wrong. However there will be a **20% deduction for each wrong**

> **attempt** at a part of a question.

The **Intro lab** does *not* count for credit, but you should attempt this in the first week of the semester to make sure that:

- you can successfully access R
- you know how to enter answers in the accompanying Moodle quiz.

Schedule

(Each link will only work once the relevant lab has been released.)

Lab	Hand-out date	Quiz due date (10am)
Intro Lab (not for assessment)	Tuesday 26 Sep (Week 1)	—
Lab 1	Thursday 5 Oct (Week 2)	Monday 9 Oct (Week 3)
Lab 2	Thursday 19 Oct (Week 4)	Monday 23 Nov (Week 5)
Lab 3	Thursday 9 Nov (Week 6)	Monday 13 Nov (Week 7)
Lab 4	Thursday 23 Nov (Week 8)	Monday 27 Nov (Week 9)
Lab 5	Thursday 7 Dec (Week 10)	Monday 11 Dec (Week 11)

Intro Lab: Meeting R and RStudio

In this introduction we begin with the fundamental building blocks of R and RStudio: the interface, reading in data, and basic commands.

The first step is to open RStudio.

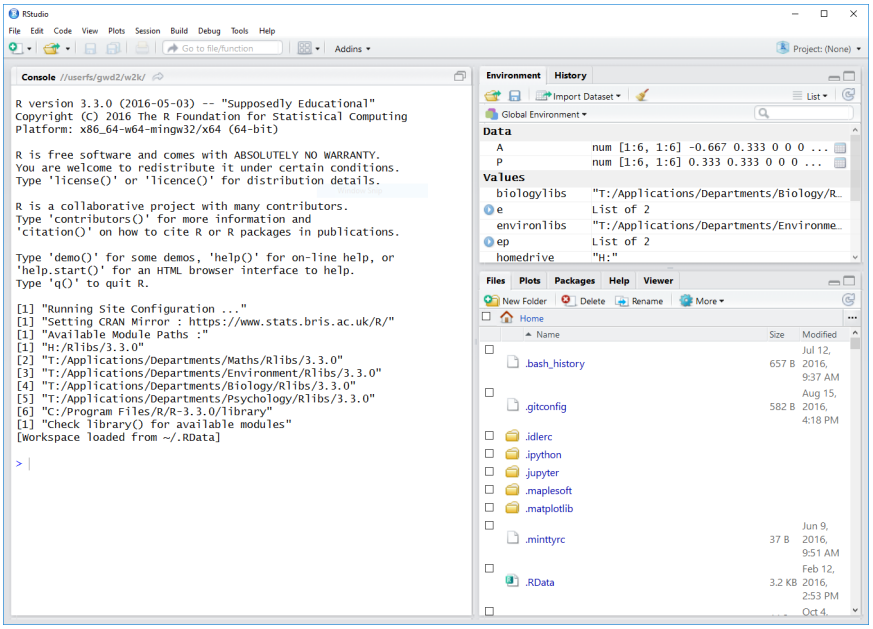
- If you are on a campus PC, RStudio is already installed and you can open it from the Windows Start menu. Just start typing 'RStudio' into the search box on the start menu and then click on RStudio when it shows up. (If you get a popup asking you whether you want to upgrade to a newer version of RStudio, simply click the "Ignore update" button.)
- If you would like to work on your own computer, you can download and install R from here and then download and install RStudio from here. Both are free and open-source and available for Windows, Mac and Linux.

Once you've opened RStudio, you should see a window similar to that depicted below.

A good way to work through these labs is is to have this file open on one half of your screen and RStudio on the other half. On a PC you can usually move a window to the left or right half of the screen by holding down the Windows key and pressing the left or right arrow key.

Note

You will see instructions to **Complete quiz questions** as you work thorugh this lab: remember that you should enter your answers in the **Quiz for Intro Lab** on Moodle.



The panel in the upper right of the RStudio window contains your *Environment* as well as a *History* of the commands that you've previously entered. The lower right panel has several tabs, including *Plots* where any plots that you generate will show up.

The panel on the left is where the action happens. It's called the *Console*. Every time you launch RStudio, it will have text at the top of the console giving lots of information that you can mostly ignore, including the version of R that you're running. Below that information is the *prompt*. As its name suggests, this prompt is really a request, a request for a command. Initially, interacting with R is all about typing commands and interpreting the output. These commands and their syntax have evolved over decades (literally) and now provide what many users feel is a fairly natural way to access data and organize, describe, and invoke statistical computations.

To get you started, enter the following command at the R prompt (i.e. right after > on the console). You can either type it in manually or copy and paste it from this document.

> **Tip**
>
> If you're using the html version of this document, then to copy the code you can simply hover your mouse over the box below: you should see a 'Copy to clipboard' symbol appear in the top right corner of the box – click on this, and then paste what you've copied into RStudio.

```
source("http://www.openintro.org/stat/data/arbuthnot.R")
```

This command instructs R to access the OpenIntro website and fetch some data: the Arbuthnot baptism counts for boys and girls. You should see that the environment area in the upper right hand corner of the RStudio window now lists a data set called `arbuthnot` that has 82 observations on 3 variables.

As you interact with R, you will create a series of objects. Sometimes you load them as we have done here, and sometimes you create them yourself as the by-product of a computation or some analysis you have performed.

Note that because it is accessing data on the web, the above command will work in a computer lab, in the library, or at home; just as long as you have access to the internet.

The data: Dr. Arbuthnot's baptism records

The Arbuthnot data set was compiled by Dr. John Arbuthnot, an 18^{th} century physician, writer, and mathematician. He was interested in the ratio of newborn boys to newborn girls, so he gathered the baptism records for children born in London for every year from 1629 to 1710. We can take a look at the data by typing its name into the console and hitting Enter.

```
arbuthnot
```

What you should see are four columns of numbers, each row representing a different year: the first entry in each row is simply the row number (an index we can use to access the data from individual years if we want), the second is the year, and the third and fourth are the numbers of boys and girls baptised that year, respectively. Use the scroll bar on the right side of the console window to examine the complete data set.

> **Tip**
>
> A nice feature of RStudio is that it comes with a built-in data viewer. Click on the name `arbuthnot` in the upper right window that lists the objects in your environment. This will bring up an alternative display of the Arbuthnot counts in the upper left panel of the RStudio window.

Moving back to the console, if we only want to see the first few lines of the data set, we can type

```
head(arbuthnot)
#>   year boys girls
#> 1 1629 5218  4683
#> 2 1630 4858  4457
#> 3 1631 4422  4102
#> 4 1632 4994  4590
#> 5 1633 5158  4839
#> 6 1634 5035  4820
```

Sometimes, as in this example, I'll show you the output of the commands when I run them on my computer, so that you can compare with what you get when you run the commands yourself: any line starting with #> corresponds to code output.

> **Tip**
>
> In the html version of this document, the word `head()` in the code block above is underlined (as is the command `source()` further up the page). Clicking on an R command which is underlined will take you to its online documentation, where you can read more about how to use it.

Note that the row numbers in the first column are not part of Arbuthnot's data. R adds them as part of its printout to help you make visual comparisons. You can think of them as the index that you see on the left side of a spreadsheet. In fact, the comparison to a spreadsheet will generally be helpful. R has stored Arbuthnot's data in a kind of spreadsheet or table called a **data frame**.

You can see the dimensions of this data frame by typing:

```
dim(arbuthnot)
#> [1] 82  3
```

This indicates that there are 82 rows and 3 columns (we'll get to what the `[1]` means in a bit), just as it says next to the object in your Environment tab. You can see the names of these columns (or variables) by typing:

```
names(arbuthnot)
#> [1] "year"  "boys"  "girls"
```

You should see that the data frame contains the columns `year`, `boys`, and `girls`. By this point, you might have noticed that many of the commands in R look a lot like functions; that is, invoking R commands means supplying a function with some number of arguments. The

`dim()` and `names()` commands, for example, each took a single argument, the name of a data frame.

Some exploration

Let's start to examine the data a little more closely. We can access the data in a single column of a data frame separately using a command like

```
arbuthnot$boys
```

This command will only show the number of boys baptised each year.

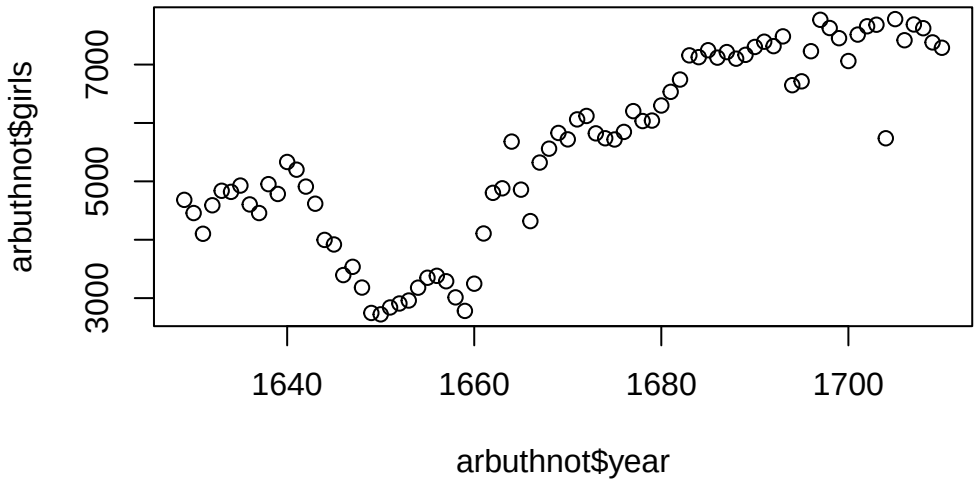
> Your turn
>
> What command would you use to extract just the counts of girls baptised each year? Try it!
> **Now answer quiz question 1.**

Notice that the way R has printed these data is different. When we looked at the complete data frame, we saw 82 rows, one on each line of the display. These data are no longer structured in a table with other variables, so they are displayed one right after another.

Objects that print out in this way are called **vectors**; they represent a set of numbers. R has added numbers in [brackets] along the left side of the printout to indicate locations within the vector. For example, `5218` follows `[1]`, indicating that `5218` is the first entry in the vector. And if `[43]` starts a line, then that would mean the first number on that line would represent the 43rd entry in the vector.

R has some powerful functions for making graphics. We can create a simple plot of the number of girls baptised per year with the command

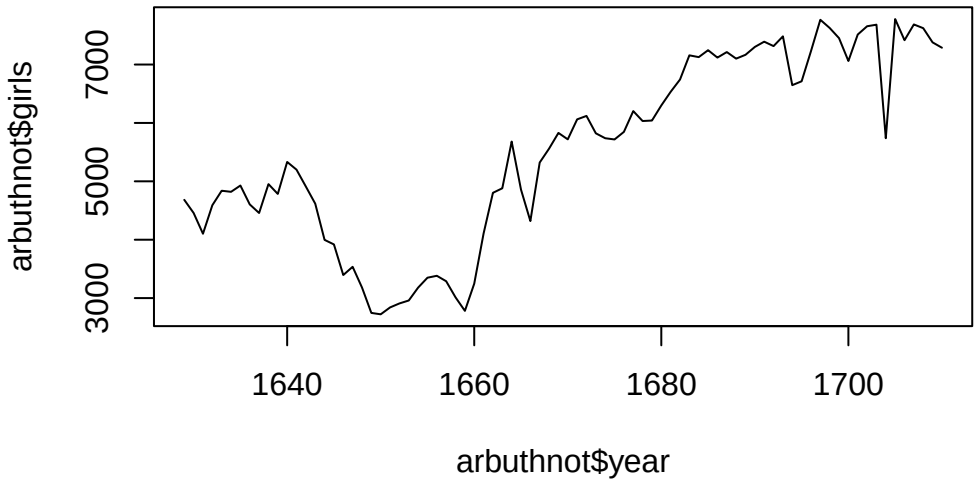
```
plot(x = arbuthnot$year, y = arbuthnot$girls)
```



By default, R creates a scatterplot with each (x,y) pair indicated by an open circle. The plot itself should appear under the *Plots* tab of the lower right panel of RStudio.

Notice that the command above again looks like a function, this time with two arguments separated by a comma. The first argument in the plot function specifies the variable for the x-axis and the second for the y-axis. If we wanted to connect the data points with lines, we could add a third argument, the letter `l` for **l**ine.

```
plot(x = arbuthnot$year, y = arbuthnot$girls, type = "l")
```



You might wonder how you are supposed to know that it was possible to add that third argument. Thankfully, R documents all of its functions extensively: you've already seen that clicking on any of the underlined commands in this page takes you to the relevant entry in

the documentation. Another way to read what a function does, and learn the arguments that are available to you, is to just type in a question mark followed by the name of the function that you're interested in. Try the following.

```
?plot
```

Can you figure out how to produce a plot that shows both the points and the lines connecting them?

Notice that the help file replaces the plot in the lower right panel. You can toggle between plots and help files using the tabs at the top of that panel.

> Your turn
>
> Is there an apparent trend in the number of girls baptised over the years?
> **Answer quiz question 2.**
> Can you also guess, just by looking at the graph, when the English civil war started?

Now, suppose we want to plot the total number of baptisms. To compute this, we could use the fact that R is really just a big calculator. We can type in mathematical expressions like

```
5218 + 4683
```

to see the total number of baptisms in 1629. We could repeat this once for each year, but there is a faster way. If we add the vector for baptisms for boys and girls, R will compute all sums simultaneously.

```
arbuthnot$boys + arbuthnot$girls
```

What you will see are 82 numbers (in that packed display, because we aren't looking at a data frame here), each one representing the sum we're after. Take a look at a few of them and verify that they are right.

We can now make a plot of the total number of baptisms per year with the command

```
plot(arbuthnot$year, arbuthnot$boys + arbuthnot$girls, type = "l")
```

This time, note that we left out the names of the first two arguments. We can do this because the help file shows that the default for `plot` is for the first argument to be the x-variable and the second argument to be the y-variable.

Next we calculate the proportion of the baptised children that are boys. We can do this for the year 1629 with the command

```
5218 / (5218 + 4683)
```

but this may also be computed for all years simultaneously:

```
arbuthnot$boys / (arbuthnot$boys + arbuthnot$girls)
```

Note that with R, as with your calculator, you need to be conscious of the order of operations. Here, we want to divide the number of boys by the total number of newborns, so we have to use parentheses. Without them, R will first do the division, then the addition, giving you something that is not a proportion.

Your turn

Now, make a plot of the proportion of boys over time. The command for making the plot will be similar to the plot command you used earlier, just with a different expression for the y argument.
Now answer quiz question 3.

Tip

If you use the up and down arrow keys, you can scroll through your previous commands, your so-called command *history*. You can also access it by clicking on the *History* tab in the upper right panel. This will save you a lot of typing in the future.

In addition to simple mathematical operators like subtraction and division, you can ask R to make comparisons like greater than, >, less than, <, and equality, == (note that it has to be a double equal sign, not a single equal sign). For example, we can ask if boys outnumber girls in each year with the expression

```
arbuthnot$boys > arbuthnot$girls
```

This command returns 82 values of either TRUE if that year had more boys than girls, or FALSE if that year did not (the answer may surprise you). This output shows a different kind of data than we have considered so far. In the `arbuthnot` data frame our values are numerical (the year, the number of boys and girls). Here, we've asked R to create *logical* data, data where the values are either TRUE or FALSE. In general, data analysis will involve many different kinds of data types, and one reason for using R is that it is able to represent and compute with many of them.

You can count the number of entries for which the condition is TRUE by just summing the entries in the vector

```
sum(arbuthnot$boys > arbuthnot$girls)
```

The reason this works is that R automatically converts TRUE to 1 and FALSE to 0 when asked to do a numerical calculation with these values.

Your turn

Above you have seen how to calculate the proportion of newborns that are boys. You have also learned how to count the number of entries in the data that satisfy a particular condition.
Now combine those two to answer quiz question 4.

A newer data set

In the previous few pages, you recreated some of the displays and preliminary analysis of Arbuthnot's baptism data. To practise your new skills, you will now repeat these steps, but for present day birth records in the United States. Load up the present day data with the following command.

```
source("http://www.openintro.org/stat/data/present.R")
```

The data are stored in a data frame called `present`.

Your turn

1. What years are included in this data set? What are the dimensions of the data frame and what are the variable or column names?
2. How do these counts compare to Arbuthnot's? Are they on a similar scale?
3. Does Arbuthnot's observation about boys being born in greater proportion than girls hold up in the U.S.?
4. Make a plot that displays the boy-to-girl ratio for every year in the data set. What do you see?
5. What was the largest total number of births in a single year in the U.S. during the

period covered by the dataset? You can refer to the help files or the R reference card to find helpful commands.

Now answer questions 5 and 6 in the quiz.

These data come from a report by the Centers for Disease Control. Check it out if you would like to read more about an analysis of sex ratios at birth in the United States.

To exit RStudio you can click the cross in the upper right corner of the whole window. You will be prompted to save your workspace. If you click *save*, RStudio will save the history of your commands and all the objects in your workspace so that the next time you launch RStudio, you will see `arbuthnot` and you will have access to the commands you typed in your previous session.

Lab 1: Script files and simulation

This lab has three goals:

1. to show you how to use R to do longer calculations using **R script files**;
2. to give you practice with using **variables** in R code;
3. to illustrate how we can use R to simulate **random samples**, and use these to empirically solve probability problems.

Especially the use of variables can be confusing, because, as the name "variable" indicates, the value of a variable can change over time.

I assume that you have already worked carefully through the previous lab so that you know how to open RStudio and execute some R commands. Again I would recommend that while working through this lab you keep this pdf file open on one half of your screen and RStudio on the other half. So now go ahead and open RStudio.

Working with an R script file

In the previous lab you worked directly in the console. For this lab you will be working in an **R script file**. An R script file is simply a text file that contains the commands that you want R to execute. The advantage of typing the R commands into the script file and executing them from there rather than typing them straight into the console is that in the script file you can lay out your calculations in an understandable way and you can revisit your calculations easily later to build on them or to share them with others.

The first step is to create a new R script file. To do that you click on the left-most icon on the toolbar at the top of the RStudio window, the one that looks like a piece of paper with a plus sign  . That opens a drop-down menu. The top entry is *R script* and is the one you want to

select. This will open an editor panel above your console with a new empty text file. That is where you will type in the R commands for this lab.

For a first example of using a script file, let's use R to simulate the experiment of drawing a ball at random from a bag containing 4 red, 6 green and 3 blue balls. (We'll look further into the idea of simulation later on in this lab; for now, just follow the instructions to get familiar with using a script file.)

- We can use the `rep()` function to create a vector with *repeated* entries. For example `rep("red", 4)`.
- We can use the `c()` function to *concatenate* several vectors.
- We can use the `sample()` function to choose a random element from a vector.

Let's combine these commands to create our bag; we will store this in a variable, that we choose to call `bag`, so that we can use it in what follows. We can also sample from the bag, and save the outcome in the variable `x`. Copy the following code into your script file:

```
#  Code to simulate the experiment of drawing balls at random
#  from a bag containing 4 red, 6 green and 3 blue balls.

#  First create the variable 'bag', which lists all ball colours:
bag <- c(rep("red", 4), rep("green", 6), rep("blue", 3))

#  Draw a ball at random from bag, and assign this to variable
 ↪  'x':
x <- sample(bag, size = 1)
```

Tip

Save the R script file frequently by clicking on the floppy disk icon on the toolbar. The first time you save the document you will be prompted to choose a **file name** and **location**:

- use an *informative* file name: don't just name it after yourself – you'll be creating lots of script files during this module, and in your future studies! A good name for this script might be `IPS_lab1.R`, or similar. (Note that R script files always have file extension `.R`.)
- if you are on a campus PC and save the document to your `H:` drive then you will be able to access it from any other campus PC or even from your home PC. For details see this IT Services page.

Now let's look at the code that you've just pasted into your script file. There are a few important things to notice here.

1. Notice the <- syntax for assigning a value to a variable. We will make a lot of use of that in the future. Many other programming languages use the syntax =.
2. Everything after a hash symbol # is ignored by R, so the hash symbol is used to start comments that explain your R code. **Commenting your code is a VERY good idea.** When you come back to look at your code again later you will be very glad that you left comments documenting what you were thinking when you originally wrote the code.
3. You probably also noticed the way I used extra spaces to align the code across the lines. Those spaces have no function, other than making the code more readable.

So far you have only put the code into your R script file – R has not yet evaluated the code. For that you should click somewhere in the first line of your code and then click the *Run* icon on the tool bar or, alternatively, hold down the *Ctrl* key and hit *Enter*. Either method will send that line of code to the R console and run it. (Notice that R skips the first few lines of comments, and only evaluates the line beginning bag.) It will also move the cursor to the next line, so that you can then execute the second line by again clicking *Run* or pressing *Ctrl-Enter*. Each time you send one of the commands to the console you should see a new variable appear in the *Environment* panel.

> **Tip**
>
> Instead of sending one line of code to the console at a time, you can also highlight multiple lines in the editor and hit *Run* just once.

Now let's suppose that we actually wanted to draw not one, but 100 balls from the bag (replacing the ball that we've withdrawn each time). We can just go back to our script and edit the final line (and its comment!) as follows:

```
#  Draw 100 balls at random from bag, and assign this to variable
 ↪  'x':
x <- sample(bag, size = 100, replace = TRUE)
```

Suppose that we want to calculate the frequencies with which we see each colour. Here's one possibility for calculating the proportion of red balls:

```
#  Calculate proportion of red balls in x:
red_prop <- sum(x == "red") / 100
```

Your turn

Add lines to your script file to calculate the proportions of blue and green balls in your vector x.

A more direct route is to use R's built-in function `table()`. This calculates counts of each distinct element in x; we can then divide by the number of draws to obtain the proportions.

```
#  Calculate counts of each colour in x:
x_counts <- table(x)
#  Now turn these into proportions:
x_props <- x_counts / length(x)
x_props
```

Note that I've used `length(x)` to calculate the number of elements in x: here we know that's 100, but writing it this way means that if I want to go back and change the number of samples, I don't have to remember to also change that number when calculating the proportions.

Note

You can download my R script file for all of the above here, and compare it to yours.

Your turn

Now add five additional yellow balls to the bag you used so far. Then record the outcome of 100,000 repetitions of the experiment of drawing a ball from that bag. Calculate the proportion of those 100,000 draws that gave a yellow ball.
Answer quiz question 1.

Simulation

We all have the intuitive idea that if we make many independent repetitions of a probability experiment, then the long run frequencies of events will be similar to their probabilities. This is indeed true, and we will investigate this formally in the lectures later when we prove the **Law of large numbers**. This means that one way to perform some of the more complicated probability calculations would be to just re-run the experiment many times to determine the frequencies of events.

Making many independent repetitions of a probability experiment is tedious. It takes a

long time to throw a die 100,000 times. So we will instead ask the computer to simulate the experiments, as we did above with the simple example of drawing balls from a bag.

In this document I am not only showing R commands that I want you to use, but I also show the output of those commands, preceded by #>, as well as the figures produced by plots. I nevertheless strongly recommend that you also evaluate the commands yourself and reproduce those outputs.

Simulating random samples

The first question we need to address is how to generate random numbers; this is a difficult problem, but one that has been extensively studied.

One way to generate random numbers would be to have an actual *physical device* in the computer that performs repeated measurements of some physical quantity whose distribution is well known. For example it is known that the arrival times of radioactive particles measured in a Geiger counter is exponentially distributed. (We'll meet the exponential distribution later in this course.)

An alternative and more convenient way to generate random numbers is to use a computer algorithm to produce a sequence of numbers that, while not truly random, is practically indistinguishable from a sequence of random numbers. They are not *truly* random numbers because if the same algorithm is run again with the same initial condition, it will produce the same sequence again. This initial condition is called the **seed** for the random number generator.

Most computer languages have good random number generators built in. This is of course particularly true for R. In fact, it has a whole range of different algorithms for generating random numbers. By default it uses the Mersenne-Twister algorithm.

There are functions in R to create samples from all of the common discrete and continuous probability distributions that we'll meet later on in this module, and it is also possible to specify your own distribution and sample from that. We will see examples of that later in this lab.

First we want to simulate a die. So we want to draw from the sample space $\{1, 2, 3, 4, 5, 6\}$ with equal probability. A quick way to generate the set of integers $\{m, m+1, m+2, \ldots, n-1, n\}$ in R is to use the command `m:n`. So with $m = 1$ and $n = 6$ we can obtain our sample space by typing

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

> **Note**
>
> We could also have used the very useful function `seq()` to do this job for us. Take a look at its documentation to see some examples of how it can be used.

Now that we have our sample space, we can use the `sample` function, as we saw above. The following produces a sample of size 30:

```
sample(1:6, 30, replace = TRUE)
#>  [1] 4 5 3 4 5 4 1 6 6 5 5 2 4 6 4 2 5 6 3 1 5 6 1 5 2 6 1 2 3
 ↪  2
```

Go ahead and put this command into a new R script file and send the command to the console repeatedly. A different random sample is produced each time.

> **Tip**
>
> A convenient way to send a chunk of code to the console repeatedly is to use the *Re-run previous code section* button, right next to the *Run* button.

Now try

```
set.seed(42)
sample(1:6, 30, replace = TRUE)
#>  [1] 1 5 1 1 2 4 2 2 1 4 1 5 6 4 2 2 3 1 1 3 4 5 5 5 4 2 4 3 2
 ↪  1
```

and notice that each time you reset the seed to 42 you get the same sequence of pseudo random numbers. Try changing the seed to a different number and see that that produces a different sample. If you want to repeat the same sample, you have to set the seed to the same value right before creating the sample, because each time you generate a random number the seed changes.

Whenever a lab introduces a new function, like `sample()` above, I recommend that you take a look at the help page for that function. To find the help for the function, you can

- type the function name into the console or the script file editor and then hit the F1 key;
- or click on the function, if it appears in R code in one of these labs and is underlined.

Doing the first of these will open the help page in the *Help* tab in the frame on the lower right of the RStudio window; the second will take you to the online documentation page. The help page first gives a brief description of the function, then sample usage, then explains the

arguments that the function can take, then provides more detailed explanations and finally, at the bottom, provides examples. I usually do not read all the details, but I have a look at the list of arguments and at some of the examples.

I strongly recommend that, in order to get a feel for the new function you just learned about, you start playing with it a bit by using it with different arguments. So for example you might try

```
sample(c("H","T"), 10, replace = TRUE)
#>  [1] "T" "T" "T" "H" "H" "T" "T" "T" "T" "T"
```

to create a sample of 10 coin flips. Or

```
sample(c("red","red", "red", "blue","blue"), 2, replace = FALSE)
#> [1] "red" "red"
```

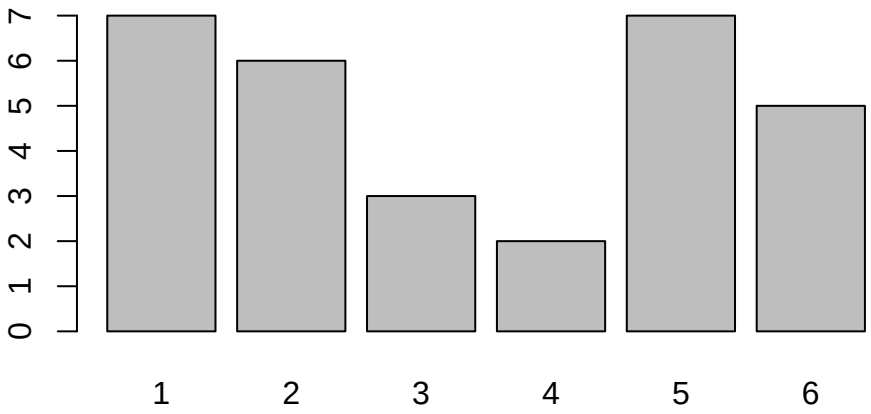
to draw two balls at random (*without* replacement) out of a bag containing three red and two blue balls. Experimentation is the best way to get friendly with the computer.

Your turn

Answer quiz question 2.

The following code sets the seed, sets the sample size to 30, creates a random sample, assigns it to the variable x, tables the frequency of each value, and then makes a barplot of the result.

```
set.seed(1)
n <- 30
x <- sample(1:6, n, replace = TRUE)
barplot(table(x))
```



> **Note**
>
> As always, you should be adding each line of code to your script file, so that you can easily re-run it later if necessary. Add your own comments to remind you what each chunk of code does!

Estimating probabilities from a random sample

Next let's estimate probabilities of various events by counting how frequently they occur in the sample.

Let's start by calculating the probability of the event that the die shows a number less or equal to 3. So our sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$, and our event of interest is $E = \{1, 2, 3\}$: we want to estimate $\mathbb{P}(E)$. We will use a trick that you met already in the first lab when you counted how many years had more newborn boys than girls. We create a vector of 0s and 1s in which a 1 in a particular place indicates that the event has taken place in that particular repetition of the experiment:

```
y <- as.numeric(x <= 3); y
#>  [1] 1 0 1 1 0 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1 1 0 1
 ↪  0
```

Then we calculate the proportion of repetitions for which the event has taken place by summing over all entries in the vector (hence counting the 1s) and then dividing by the size of the sample:

```
sum(y)/n
#> [1] 0.5333333
```

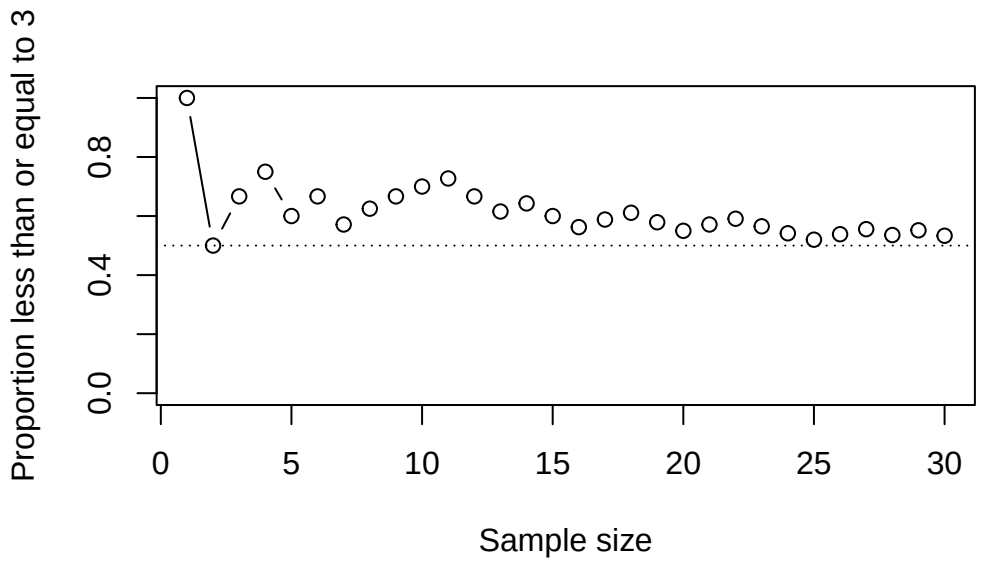
This gives the best approximation to the probability $\mathbb{P}(E)$ that we can obtain from this sample. It is close to but not exactly equal to the theoretical value of 0.5.

Your turn

Answer quiz question 3.

We can make a plot that shows how the approximation to the probability behaves as the sample size grows:

```
yn <- cumsum(y)/(1:n)
plot(yn, type = "b", ylim = c(0,1),
     xlab = "Sample size", ylab = "Proportion less than or equal
      ↪  to 3")
abline(h = 1/2, lty = "dotted")
```



This shows that while the values in the random sample keep fluctuating, the estimate of the probability settles down towards its true value as the sample size increases.

The first line of the code above produces a vector of values whose i^{th} entry is the proportion of 1s in the first i values in the vector y. It then assigns this vector of proportions to the variable yn. You do not have to understand the command in detail, unless you want to.

The second line produces the plot of the values, where we have asked R to show both the points and the straight lines joining them, and to limit the range of the y-axis to the interval (0,1). We've also added more informative labels to the axes.

Finally, the last line `abline(h = 1/2, lty = "dotted")` draws a dotted horizontal line at the height 0.5 to indicate the theoretical answer to $\mathbb{P}(E)$.

Now play around by producing similar plots for larger sample size.

We can similarly calculate the probability that the die shows a six with

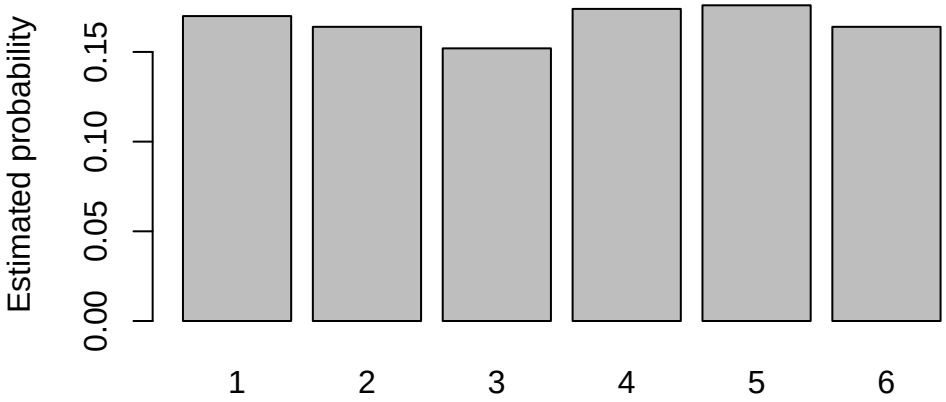
```
y <- as.numeric(x == 6); y
#>  [1] 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1 0
 ↪  0
sum(y)/n
#> [1] 0.1666667
```

The correct value of course is $1/6 \approx 0.167$. We see that the sample is really too small to give a reliable estimate of the probability of obtaining a six. So we redo this with a larger sample of size 1,000:

```
n <- 1000
set.seed(1)
x <- sample(1:6, n, replace=TRUE)
y <- as.numeric(x == 6)
sum(y)/n
#> [1] 0.164
```

The following code performs the calculation of the estimated probability for all values from 1 to 6 and plots them in a bar plot.

```
barplot(table(x)/n, ylab = "Estimated probability")
```



Better, but still not a very good approximation to the theoretical answer. This illustrates that one needs very large sample sizes to get reliable results. Repeat this with larger samples to see how the estimates improve.

> Your turn
>
> Set the seed to 12. Produce a sample of size 1,000,000 for the experiment of rolling a fair 6-sided die. What proportion of rolls give the outcome 6?
> **Answer quiz question 4.**

We can also use our sample to approximate the probability of more complicated events. For example, suppose that we wish to consider the event that the outcome of a fair die roll is a 2 or a 3. That is, we want to estimate $\mathbb{P}(\{2, 3\})$. We can do this by counting the numbers of 2s and 3s in our sample

```
sum(x == 2 | x ==3)
#> [1] 316
```

Note that we've used the symbol `|` to mean **or**. So `sum(x == 2 | x ==3)` counts how many entries in `x` are equal to 2 or equal to 3. Similarly, we can use the symbol `!=` to mean **not equal**, and the symbol `&` to mean **and**. So

```
sum(x > 1 & x < 4)
#> [1] 316
```

is another way of counting the number of 2s and 3s, while

```
sum(x != 5)
#> [1] 824
```

counts the number of outcomes in x that are not equal to 5.

> Your turn
>
> Use the same sample that you generated for Question 4 (the sample of size 1,000,000) to approximate the probability that a fair die roll gives an outcome that is an integer multiple of 3.
> **Now answer quiz question 5.**

Another probability problem

Simulation provides a lazy way of "solving" probability problems. Take for example the following problem.

> A shop receives a batch of 1,000 cheap lamps. The chance that any given lamp is defective is 0.1%. What is the probability that there are more than two defective lamps in the batch?

We can easily simulate a batch of 1,000 cheap lamps. Let us represent a defective lamp by 1 and a working lamp by 0.

```
set.seed(0)
lamps <- sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999,
 ↪  0.001))
lamps
#>    [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>   [38] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>   [75] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [112] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [149] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
```

```
#>  [186] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [223] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [260] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [297] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [334] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [371] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [408] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [445] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [482] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [519] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [556] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [593] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [630] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [667] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [704] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [741] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [778] 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [815] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [852] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
```

```
#>  [889] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [926] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0
#>  [963] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
 ↪  0 0 0 0 0 0 0 0 0
#> [1000] 0
```

We can then count how many defective lamps are in that batch.

```
sum(lamps)
#> [1] 2
```

There were 2 defective lamps in that sample. Now, without resetting the seed, we take another sample to represent another random batch of lamps and again count the defective lamps.

```
lamps <- sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999,
 ↪  0.001))
sum(lamps)
#> [1] 0
```

0 in this batch. Let's try another

```
sum(sample(c(0, 1), 1000, replace = TRUE, prob = c(0.999, 0.001)))
#> [1] 1
```

The `replicate()` function allows us to repeat this a chosen number of times and collect the results into a vector.

```
set.seed(0)
replicate(20, sum(sample(c(0,1), 1000, replace = TRUE, prob =
 ↪  c(0.999, 0.001))))
#>  [1] 2 0 1 1 0 0 0 1 0 0 0 0 1 2 0 1 2 1 0 2
```

So no batch with more than 2 defective lamps in the first 20 batches. Now we will simulate 100,000 batches and then count the number of batches with more than 2 defective lamps.

```
set.seed(0)
count_defective <- replicate(100000, sum(sample(c(0,1), 1000,
 ↪  replace = TRUE, prob = c(0.999, 0.001))))
sum(count_defective > 2)
#> [1] 8022
```

We can use this to estimate the probability of getting more than two defective lamps in a batch by dividing this by the total number of batches

```
sum(count_defective > 2) / 100000
#> [1] 0.08022
```

i What answer should we expect here?

If X is the number of defective lamps in a batch of size 1,000, then you may already know that X will follow a **binomial distribution** with parameters 1,000 and 0.001: $X \sim \text{Bin}(1000, 0.001)$. (Don't worry if this doesn't mean anything to you: we'll be learning about this properly later in the course!)
We can write

$$\mathbb{P}(X > 2) = 1 - \mathbb{P}(X = 0) - \mathbb{P}(X = 1) - \mathbb{P}(X = 2)\,.$$

So to calculate this we need to be able to evaluate the probability mass function of this binomially distributed random variable. Of course R has a function for this, called `dbinom()`. So we can calculate the probability that more than 2 batches have a defective lamp as

```
1 - dbinom(0, 1000, 0.001) - dbinom(1, 1000, 0.001) -
 ↪  dbinom(2, 1000, 0.001)
#> [1] 0.08020934
```

In fact, R also has a function `pbinom()` for calculating the *distribution function*. So we could also have calculated $\mathbb{P}(X > 2) = 1 - \mathbb{P}(X \leq 2)$ with

```
1 - pbinom(2, 1000, 0.001)
#> [1] 0.08020934
```

Of course in this example it was faster to solve the problem by using the binomial distribution instead of by simulation, but there are many real-world probability problems that

can not be solved analytically and for which simulation is the only viable approach.

Your turn

Set the seed to 0 and then simulate 10,000 batches of 2,000 lamps each to estimate the probability that there are exactly two defective lamps in a batch of 2,000 lamps.
Now answer quiz question 6.

Written assignments

Assignment sheets will appear on Moodle as **pdf** files. If you would prefer to view an **html** version then you can find links to these below. (Each link will only work once the relevant sheet has been released on Moodle.)

Assignment	Hand-out date	Due date	Solutions
Assignment 1	Tuesday 26 Sep (Week 1)	Thursday 5 Oct (Week 2)	Assignment 1
Assignment 2	Tuesday 10 Oct (Week 3)	Thursday 19 Oct (Week 4)	—
Assignment 3	Tuesday 24 Oct (Week 5)	Thursday 9 Nov (Week 6)	—
Assignment 4	Tuesday 14 Nov (Week 7)	Thursday 23 Nov (Week 8)	—
Assignment 5	Tuesday 28 Nov (Week 9)	Thursday 7 Dec (Week 10)	—