# Lab 4 – Discrete probability simulations

In this lab, we are going to simulate experiments and estimate probabilities using the relative frequency approach to probability. That is, we will simulate running an experiment *n* times, and each time we will count the number of “successes” *k*. This will allow us to estimate the probability of obtaining *k* successes.

**To submit: answers to all numbered questions. When the question asks you to write code or create graphs, submit the code and/or graphs in the Word document as part of your answer. Also submit a single .R file that contains all of your code.**

## Experiment 1: flipping a fair coin

Our first experiment will simulate flipping a fair coin once. We can do this in the console using the **sample** function.

Here is what R’s help file tells us about the **sample** function:

**Usage**

sample(x, size, replace = FALSE, prob = NULL)

**Arguments**

|  |  |
| --- | --- |
| x | Either a vector of one or more elements from which to choose, or a positive integer. See ‘Details.’ |
| n | a positive number, the number of items to choose from. See ‘Details.’ |
| size | a non-negative integer giving the number of items to choose. |
| replace | Should sampling be with replacement? |
| prob | A vector of probability weights for obtaining the elements of the vector being sampled. |

The first argument, x, is the sample we’re choosing from. We represent that sample by a collection of numbers. For the coin-flip example, our sample space is {heads, tails}. We can either represent those as the words “heads” and “tails”, or we can represent them with numbers – ie, “heads”=1, “tails”=2. We will try both versions.

The second argument, n, is the items we are selecting. Equivalently, n represents the number of times we are running the experiment. Here, we are flipping the coin once, so n=1.

To simulate the single coin-flip, type the following in the console:

> sample(1:2,1)

This tells R to select 1 item from a list of numbers that goes from 1 to 2.

output will be either

[1] 1

or

[1] 2

indicating a result of heads (1) or tails (2). The [1] that precedes both results indicates that you just made one selection from the list {1,2}, ie, you simulated just [1] coin flip.

1. Try running that command ten times to see the results. When you run it ten times, how many heads do you get? How many tails? Include these ten outputs.

> sample(1:2,1)

[1] 1

> sample(1:2,1)

[1] 2

> sample(1:2,1)

[1] 1

> sample(1:2,1)

[1] 2

> sample(1:2,1)

[1] 2

> sample(1:2,1)

[1] 2

> sample(1:2,1)

[1] 1

> sample(1:2,1)

[1] 1

> sample(1:2,1)

[1] 1

> sample(1:2,1)

[1] 1

The result is: 6 heads (1) and 4 tails (2)

We can also have our program return the actual result “Heads” or “Tails”. This time, instead of the list 1:2, we will use the list **c(“Heads”, “Tails”)**. Now enter the following in the console.

> sample(c("Heads", "Tails"),1)

Now your program will return either

[1] "Tails"

or

[1] "Heads"

# Writing a script in R

# 

# Our coin-flipping experiment was a single line that we could easily enter into the console. However, for more complicated programs, we will want to write scripts.

# Open a new script in R:

# 



# We will define a function called FlipOnce() to simulate a single coin flip and return either heads or tails. The two parentheses indicate that this function doesn’t take any arguments.

Type the following into the script window:

FlipOnce = function()

{ HeadOrTail<-sample(c(“Heads”, “Tails”), 1)

return(HeadOrTail)

}

Note the syntax: curly braces { } instruct R how to parse your commands. We assign the result of the coin flip to the variable **HeadOrTail**, which R will return to us.

Now we can run our script. First, save your script as Lab4.R in an appropriate folder. Be sure to check off the “Source on Save” option; this tells R where to look when you call your function.

Now call your command from the console:

> FlipOnce()

As before, your program will return

[1] "Tails"

or

[1] "Heads"

Now we will write a script to flip a coin *n* times and return the results. There are several ways we can do this. One way is to call the **FlipOnce()** function *n* times.

In the same script window, define a new function **CoinResults(n)**. This function will call the **FlipOnce()** function in a loop. Type the following:

CoinResults=function(n)

{coinList<-c(1:n)

{for (i in 1:n)

coinList[i]=FlipOnce()

}

return(coinList)}  
  
Let’s break down this code:

The first line of code defines the function, and allows the user to specify the number *n* of coin flips.

The second line initializes the list of results.

The third through fifth lines simulate *n* coin flips using your **FlipOnce()** function. Note the syntax of the   
**for** loop.

Finally, the last line returns the list of results.

You can insert comments using the **#** sign.

CoinResults=function(n)

{coinList<-c(1:n)

{for (i in 1:n) #create a list of results

coinList[i]=FlipOnce()

}

return(coinList)}

Now save your script again, and call **CoinResults(10)** in the console:

> CoinResults(10)

[1] "Tails" "Tails" "Heads" "Tails" "Heads" "Heads" "Tails"

[8] "Heads" "Heads" "Heads"

This time, the numbers in brackets give the index of the beginning of the list. For instance, the second row begins with the 8th coin flip.

Try running your script with different values of **n**.

For computing probabilities, we are really only interested in the number of coins that came up heads and tails, not the actual list of results. We will create a third script that counts the number of heads out of **n** and returns the probability.

ProbHeads=function(n)

{coinList<-CoinResults(n)

numHeads<-sum(coinList=="Heads")

return(numHeads/n)}  
  
The third line sums up the number of “Heads”. Note the syntax, particularly the double equals sign used in the comparison operator. If you returned the line **CoinList==“Heads”** on its own, and ran the function for n=100, you would get something like this:

[1] TRUE FALSE FALSE TRUE TRUE FALSE TRUE TRUE FALSE

[10] FALSE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE

[19] TRUE TRUE FALSE TRUE FALSE FALSE TRUE FALSE TRUE

[28] FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE TRUE

[37] TRUE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE

[46] TRUE FALSE TRUE FALSE FALSE FALSE TRUE TRUE TRUE

[55] FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE

[64] TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE TRUE

[73] TRUE TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE

[82] FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE

[91] TRUE TRUE TRUE TRUE FALSE FALSE TRUE TRUE FALSE

[100] FALSE

Note that if an entry on your list is “Heads”, the corresponding entry of **CoinList==“Heads”** is TRUE. Otherwise, it’s false. The third line of your **ProbHeads(n)** function counts the number of TRUE entries.

If you run your **ProbHeads** function ten times for for **n**=100, you will get results like this:

> ProbHeads(100)

[1] 0.49

> ProbHeads(100)

[1] 0.57

> ProbHeads(100)

[1] 0.49

> ProbHeads(100)

[1] 0.51

> ProbHeads(100)

[1] 0.54

> ProbHeads(100)

[1] 0.51

> ProbHeads(100)

[1] 0.5

> ProbHeads(100)

[1] 0.45

> ProbHeads(100)

[1] 0.45

> ProbHeads(100)

[1] 0.51

These are around 50%, which is consistent with the true probability of obtaining heads on a fair coin. But some of the results are off by a fair bit – the lowest probability is 45% and the highest is 57%. How do you expect those numbers to change when you run **ProbHeads(1000)**, **ProbHeads(10000)**, or **ProbHeads(10000)**?

We will address that question in a bit. First, though, we will write a simpler function that takes better advantage of R’s built-in commands. For instance, we saw that the **sample** function allows us to sample multiple times, instead of just once. This will spare us the trouble of writing a loop.

For instance, type this line in the console:

> sample(c("Heads", "Tails"), 10, repl=T)

Your results will look something like this:

[1] "Tails" "Heads" "Heads" "Tails" "Heads" "Tails" "Tails"

[8] "Heads" "Heads" "Tails"

1. Now rewrite your **CoinResults(n)** function in the script window without using loops. Use your new **CoinResults**(**n**) function in your **ProbHeads(n)** function. Copy both your **ProbHeads(n)** code and its output for n=1000 into your Word document.

CoinResults = function(n) {

return(sample(c("heads", "tails"), n, repl=T))

}

ProbHeads = function(n)

{

coinList = CoinResults(n)

numHeads = sum(coinList == "heads")

return(numHeads/n)

}

> ProbHeads(1000)

[1] 0.492

Note that R only lets you return one result per function. However, if you want to return multiple results, you can get around this restriction by having it return the list c(result1, result2).

1. Write a function called **MaxAndMinHeads(n,m)** that calls your **ProbHeads(n) function m** times, and returns the maximum and minimum probability of obtaining heads. Run your program for the following values of **n** and **m**, and submit your results in the following table (you can copy/paste this table into your Word document and fill it in manually). What do you notice as **m**  and **n** increase?

The bigger the sample size and sample trials, the more balanced our probability is looking (that is, we expect to have 50% chance of heads or tails). We observe, that our propability approaches it the bigger the numbers are.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n \ m | 10 | 100 | 1000 | 10000 |
| 10 | 0.8  0.2 | 0.9  0.2 | 1  0 | 1  0 |
| 100 | 0.56  0.33 | 0.60  0.34 | 0.69  0.33 | 0.68  0.32 |
| 1000 | 0.505  0.472 | 0.541  0.449 | 0.549 0.447 | 0.557 0.444 |
| 10000 | 0.5018 0.4910 | 0.5122 0.4871 | 0.5151 0.4807 | 0.5196 0.4828 |

## Experiment 2: rolling a fair die

In our simulations, we are often interested in returning a table of results of the following form:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Result |  |  |  |  |  |  |
| Frequency of result |  |  |  |  |  |  |

For instance, in the coin flip example for n=100:

|  |  |  |
| --- | --- | --- |
| Result | Heads | Tails |
| Frequency of result | 47 | 53 |

R lets us do this very easily, by just keeping a list of results and letting us convert that list into a table. (We did this in Lab 1, with a dataset.) For example, suppose we roll a 6-sided die 10 times and get the following results: 3,6,3,1,2,6,4,2,1,5. We can store these results in a list and convert them into a table, as follows:

> dieList<-c(3,6,3,1,2,6,4,2,1,5)

> dieTable<-table(dieList)

> dieTable

dieList

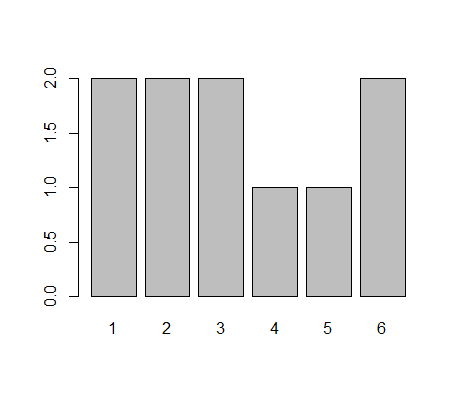
1 2 3 4 5 6

2 2 2 1 1 2

This tells us that we have rolled two 1’s, two 2’s, two 3’s, one 4, one 5, and two sixes.

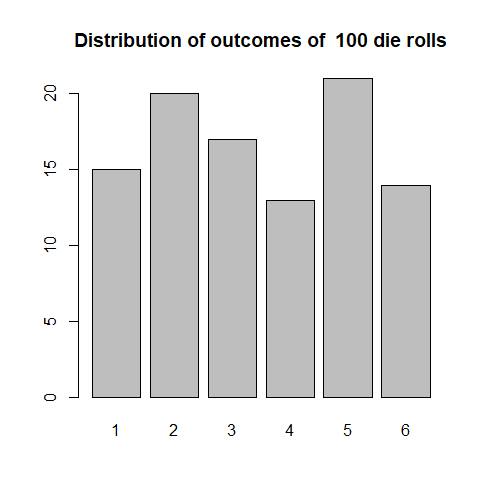
We can then create a bar graph based on this data:

> barplot(dieTable))

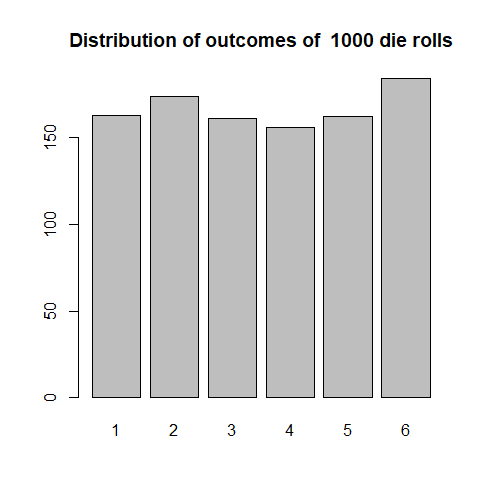


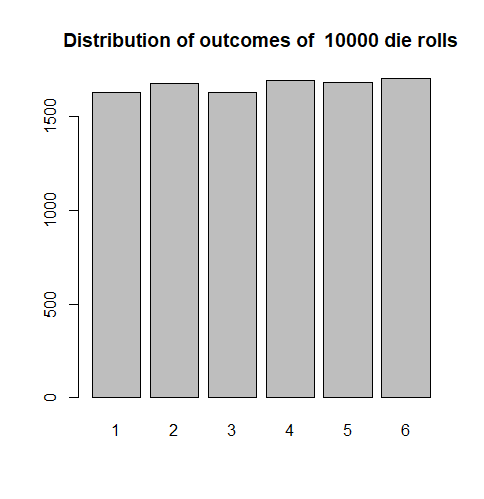
As always, you can customize your bar graph with labels.

1. Write a function called **RollDie(n)** that simulates rolling a die **n** times, and returns a bar plot like the one above that gives the distribution of outcomes. The title of your bar plot should be “Distribution of outcomes of **n** die rolls”, where **n** is the actual value of **n** you entered. (See the **paste** function for help concatenating strings). Run your function for n=100, 1000, and 10000. Submit all three graphs along with descriptions of their shapes. Are the results as expected?



For n=100 we can see somewhat equal roll distribution among the 6 values. Giving a very small sample size we expect to have unequal distribution.



Now, as we increased the sample size we can observe that the distribution equalized among all values yet giving a slight skew to values 2 and 6. Running a bigger sample size test expect to equalize even more.  


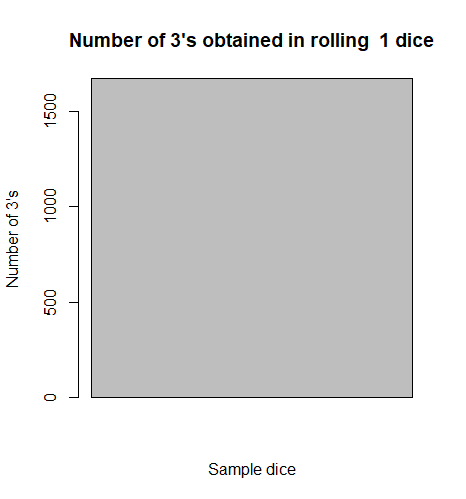
As expected, for a big sample size we get almost perfectly equal distribution among all 6 values of the die. Results are within our expectations.

1. Write a function called **RollSomeDice(n,m)** that simulates rolling **m** dice **n** times, and each time counts the number of 3’s obtained in the **m** dice. Your function should output a table that gives the distribution of the number of 3’s, as well as a bar plot. The title of your bar plot should be “Number of 3’s obtained in rolling m dice”, where m is the actual value of m you entered. Choose your axis labels appropriately.

> RollSomeDice(10000,1)

dieRollList

1669

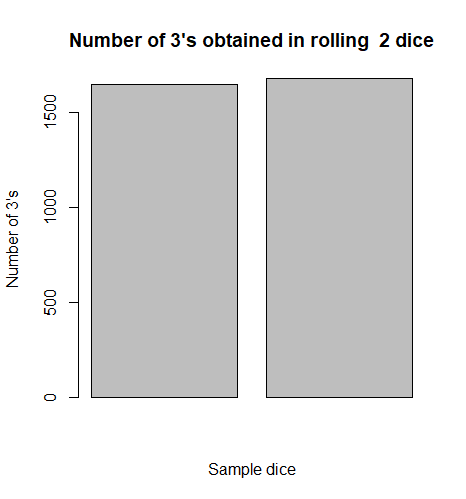


First graph is very simple as it just shows how many 3’s we got from rolling 1 die 10000 times

> RollSomeDice(10000,2)

dieRollList

1647 1679

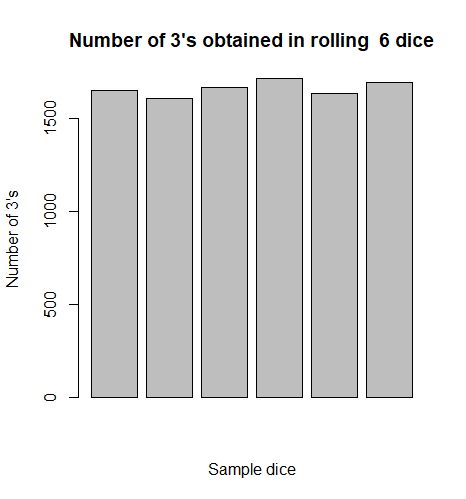


Second graph allows us to see that from 2 different samples 3’s distribution is fairy equal

> RollSomeDice(10000,6)

dieRollList

1606 1630 1650 1665 1690 1711

1 1 1 1 1 1   


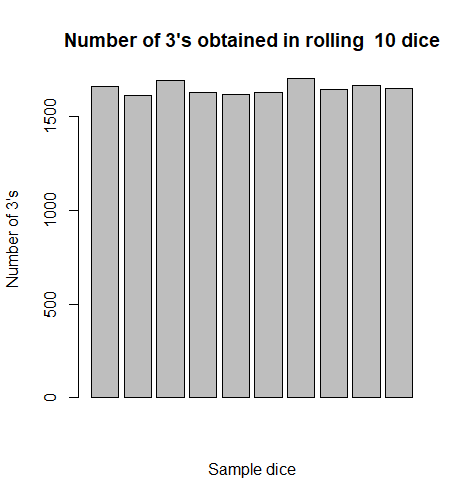
The distribution is equal among all 6 dice.

> RollSomeDice(10000,10)

dieRollList

1614 1619 1627 1628 1646 1652 1658 1666 1691 1702

1 1 1 1 1 1 1 1 1 1



The distribution is still very equal among all 10 dice.

> RollSomeDice(10000,100)

dieRollList

1595 1598 1603 1606 1607 1613 1629 1631 1632 1633

1 1 1 1 1 1 3 1 1 1

1635 1636 1637 1638 1640 1641 1642 1644 1646 1647

1 1 1 1 1 2 2 1 2 1

1648 1649 1650 1651 1653 1656 1657 1658 1660 1661

1 1 2 2 3 2 1 1 1 1

1662 1663 1665 1667 1668 1669 1670 1671 1673 1674

3 1 1 1 1 1 1 1 1 1

1675 1676 1679 1680 1681 1683 1685 1687 1689 1691

2 1 3 1 1 1 1 1 3 2

1692 1693 1695 1696 1697 1699 1700 1701 1702 1705

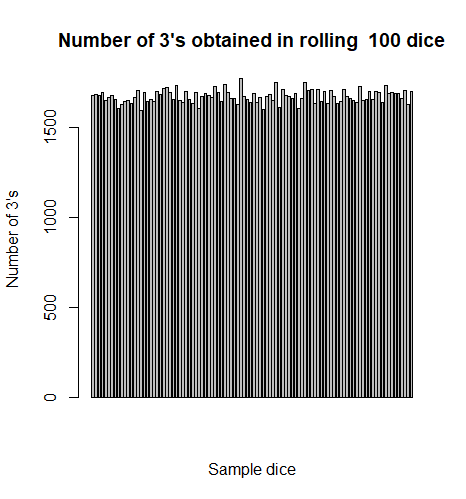
3 1 1 1 2 2 2 1 1 2

1708 1709 1711 1713 1717 1720 1728 1733 1741 1748

2 1 1 2 1 1 2 2 1 1

1751 1771

1 1



Our last result shows that even with a high sample size and number of runs, number of 3’s fluctuates but stays in within average margin (that is, we can’t observe any outlying results within a big sample size).

## Experiment 3: drawing cards

We modelled coin flips and die rolls as sampling with replacement. That is, we simulated the die-roll experiment by imagining a box that contained the numbers 1, 2, 3, 4, 5, and 6, drawing a number, and then putting it back and drawing again.

Some experiments are best modelled as sampling *without* replacement. Drawing cards is a common example of such an experiment. We may want to know the probability of getting 3 red cards if we draw 5 cards without replacement from a 52-card deck.

As before, we use the **sample** function, but this time we set **replace** to **FALSE**.

1. Write a function **DrawCardsWithReplacement(n,m)** that simulates drawing **m** cards from a 52-card deck **with** replacement, **n** times. For each draw, record the number of red cards. (Think about how you want to model the card colour.) Your program should return an appropriately-labelled bar graph (**m** should be in the title of the graph) that gives the frequency of the number of red cards.  
     
   Run your program for **n=10000** and **m=1, 5, 10, 30, 50.** Save those graphs for now; you will be using them in the next question.
2. Now write a function **DrawCardsWithoutReplacement(n,m)** that simulates drawing **m** cards from a 52-card deck **without** replacement, **n** times. For each draw, record the number of red cards. Your program should return an appropriately-labelled bar graph that gives the frequency of the number of red cards.  
     
   Run your program for **n=10000** and **m=1, 5, 10, 30, 50.** Save those graphs.

Now paste your graphs into the following table, which you can copy/paste into your Word document. Shrink them if necessary to fit them all on one page.

|  |  |  |
| --- | --- | --- |
| m | With replacement | Without replacement |
| 1 |  |  |
| 5 |  |  |
| 10 |  |  |
| 30 |  |  |
| 50 |  |  |

For each m, compare the two types of graphs. What do you notice as **m** increases?What accounts for the differences?

**For m=1:** the graphs are identical, since we have equal chances of pulling red card despite the replacement property.

**For m=5:** the graphs are also almost identical as for the m=1

**For m=10:** Even though the graph look identical, we can observe that with replacements we have more outliers -that is, there were more samples with higher or lower number of red cards.

**For m=30:** The graphs have similar shape, but the biggest difference is the variance – it is impossible to draw more than 26 and less than 4 red cards without replacement, whereas it is the case. With replacement number of red cards vary from 5 to 25, whereas without replacement we have values between 8 and 21.

**For m=50:** the deal-breaking graph that shows, that without replacements we are guaranteed to draw 24,25 or 26 red cards, whereas with replacement we can draw anywhere between 13 to 37 cards.

All the graphs make sense and fall within theoretical predictions.