

Some Simulations for Lab 9 STAT 457, Fall, 10

Problem #1—the 1-1-1 free throw Lady Griz

[1] Generate a simulation for a Lady Griz, 72% free throw shooter, who has a special free throw situation—a 1-and-1-and-1 situation. It is based as the 1-and-1 free throw situation, where she gets 1 free throw, and if made, gets another. For our situation, she gets up to 3 shots, as long as she keeps making free throws. Say she gets 10 of these shots per game, and the season is 25 games long. How many shots, on average, will she get per game during the season?

Analysis:

Below is a list of sample free throws shot by a 72% lady griz player.

first	0	1	1	0	0	1	1	1	1	1
second	1	1	0	1	0	0	1	1	0	0
third	0	1	0	1	1	1	1	0	0	1
points	0	3	1	0	0	1	3	2	1	1

After making this list of shots attempted, and totaling up the points, I tried to find a relationship (or formula) using the shots, which would accurately describe each point total made. After staring at the display and thinking for a long while, I came up with the following formula

$$points = first + first*second + first*second*third$$

So, I created a script shown below, which creates a one game total of points simulation.

```
# 1 and 1 and 1 simulation, 1 game
```

```
# initializing needed constants  
shot.percent <- .72
```

```
n <- 10
```

```

game.points <- c()

# now the shooting loop
for(i in 1:n) {
  first <- rbinom(1, size=1, prob=shot.percent)
  second <- rbinom(1, size=1, prob=shot.percent)
  third <- rbinom(1, size=1, prob=shot.percent)
  game.points[i] <- first + first*second + first*second*third
}
total1 <- sum(game.points)

```

Notice that we first created variables we would need in the script, the variables `shot.percent`, `n` (the number of shots taken per game) and `shot.points`. Since we create this vector, one element at a time in the *for* loop of R, we have to tell R before we enter the loop to save a location for a coming vector (indicated by the `c()`) which we will build up an element at a time in the *for* loop. Without this statement before we enter the *for* loop, R would give an error message saying that it doesn't know what `shot.points` is.

Next, we enter the *for* loop, creating all 3 shots, based upon a 72% free throw percentage. Then we store the total number of points in the variable `game.points`. Finally, we store the sum of all entries of the 10 element vector `game.points` into the variable `total1`, which contains our point total of free throw points made during one game. A sample run is shown below of this script.

```

> total1
[1] 14
> game.points
[1] 0 3 0 1 1 2 1 3 2 1
> |

```

We will take this script, most of which is mostly in a *for* loop which loops 10 times, and loop that loop within another *for* loop, which goes for 25 times. This *for* loop within another *for* loop script, called `season111.R`, will give us a complete season of points per game, where the 72% free throw shooter takes 10 shots per game.

```

# 1 and 1 and 1 simulation, 25 games
# initializing needed constants
shot.percent <- .72
n <- 10
game.points <- c()
season.points <-c()

games <- 25

```

```

# game for loop with shots per game
# for loop nested within it
total.season <- c()
for (j in 1:games) {
  for(i in 1:n) {
    first <- rbinom(1, size=1, prob=shot.percent)
    second <- rbinom(1, size=1, prob=shot.percent)
    third <- rbinom(1, size=1, prob=shot.percent)
    game.points[i] <- first + first*second + first*second*third
    totall <- sum(game.points)
  }
  season.points[j] <- totall
}
season.average <- mean(season.points)
print(season.average)

hist(season.points, main="25 game season 1 and 1 and 1 points made per game")
summary(season.points)
sd(season.points)

```

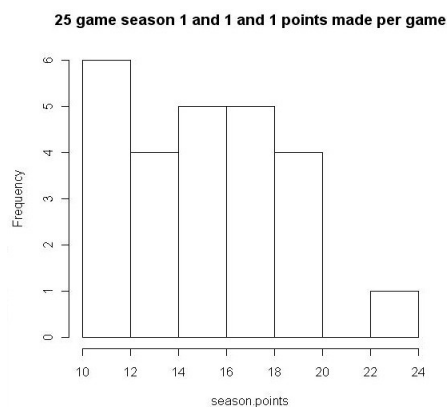
Notice that we saved new variables `games` and `season.points`, and that `season.points` is a vector which gets built up slowly in the 25 loop outside *for* loop. Again, we had to tell R that `season.points` was coming up in a *for* loop to follow. Our output is the average of the 25 games worth of points stored in `season.points`, as well as having an output of each game point totals. A sample output is shown below.

```

> print(season.average)
[1] 15.68
> season.points
[1] 13 16 14 24 13 10 19 14 12 16 17 20 18 12 12 16 18 12 15 12 17 17 15 20 20
> |

```

A histogram asked for in the output is shown below.



The summary information from the histogram output is also shown below.

```
> summary(season.points)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 10.00  13.00   16.00   15.68  18.00   24.00
> sd(season.points)
[1] 3.375401
< |
```

I do not claim that this series of scripts are the best ways, or the most efficient, or the most clever ways, to perform this simulation, only that it is one way to do it, along with good information from the simulation which would be typical to compute from a simulation like this.

Also, I do not expect you to care much about basketball players making free throws, only that this type of simulation might be tailored to fit some simulation you would like to do in your major area(s) of study.

=====

Problem #2—the single die problem

[2] How many times do you expect to roll a fair die until you get the same face you rolled previously (called “2 in a row”)?

This simulation would be a “showcase” opportunity to use the R command `sample()`. See the attached help menu printout on this command. We would use

```
sample(1:6, size=1)
```

to roll a single fair die once. The command tells R to randomly pick from the numbers 1, 2, 3, 4, 5, 6, all equally weighted, one time. The script to simulate the problem is called `die1.R`, listed below.

```
# roll 1 die until you get doubles

counter <-1
roll1 <- sample(1:6, size=1)
for (i in 1:20) {
  roll2 <- sample(1:6, size=1)
  if(roll2==roll1) break
  counter <- counter + 1
  roll1 <- roll2
}
print ("number of rolls to get doubles is")
counter
```

Below is a printout of a couple of simulations.

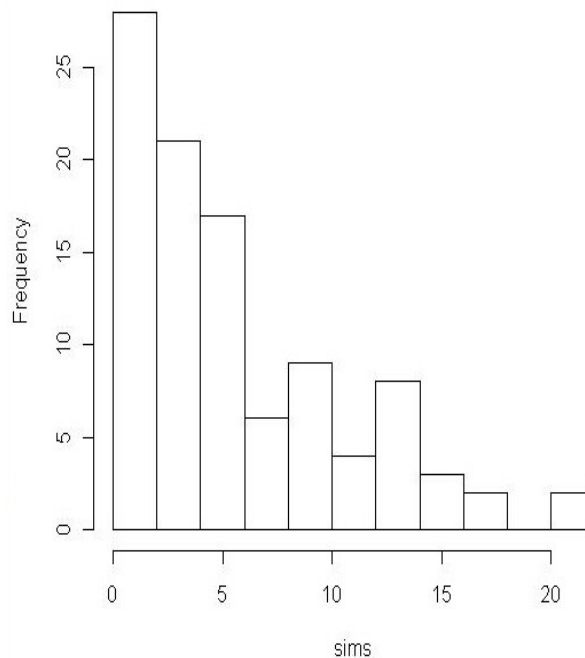
```
> print ("number of rolls to get doubles is")
[1] "number of rolls to get doubles is"
> counter
[1] 6
> # roll 1 die until you get doubles
>
> counter <-1
> roll1 <- sample(1:6, size=1)
> for (i in 1:20) {
+ roll2 <- sample(1:6, size=1)
+ if(roll2==roll1) break
+ counter <- counter + 1
+ roll1 <- roll2
+ }
> print ("number of rolls to get doubles is")
[1] "number of rolls to get doubles is"
> counter
[1] 2
```

If I wanted to make 100 simulations of the 2-in-a-row, I would just have to put the program shown above into a *for* loop 100 times, as done with the previous simulation. See below.

```
# roll 1 die to get doubles 100 times
sims <- c()
counter <- 1
simulate <- 100
for (j in 1:simulate) {
  roll1 <- sample(1:6, size=1)
  for (i in 1:20) {
    roll2 <- sample(1:6, size=1)
    if(roll2==roll1) break
    counter <- counter + 1
    roll1 <- roll2
  }
  sims[j] <- counter
  counter <-1
}
print("results per simulation =")
sims
summary(sims)
cat("std. dev. is", sd(sims), "\n")
hist(sims, main="number of rolls to duplicate roll")
```

Below is the output from 100 simulations, a histogram of them, and some summary statistics of the simulations.

number of rolls to duplicate roll



```
> print("results per simulation =")
[1] "results per simulation ="
> sims
 [1] 3 1 15 1 11 2 5 2 7 1 8 3 3 1 5 2 3 13 15 1 3 1 18 4 5
 [26] 9 21 8 3 6 13 5 11 7 7 11 4 3 9 5 1 13 4 5 2 2 5 9 6 15
 [51] 1 1 4 6 2 2 1 9 6 11 14 2 3 6 13 21 6 4 2 10 9 14 6 2 7
 [76] 3 2 14 6 3 4 17 5 4 4 1 1 2 1 4 9 13 9 5 9 2 4 1 3 2
> summary(sims)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.00  2.00   5.00   6.07  9.00   21.00
> cat("std. dev. is", sd(sims), "\n")
std. dev. is 4.837261
```

There is (at least)

another way to do this same simulation, without having to use a *for* loop. Remember that we had previously stated that it is a good idea to avoid using R *for* loops if at all possible, because these structures use up a lot of needless computer memory for big looping programs.

Picture the following situation. We create a vector 11 elements long, of random picks from the sample 1 through 6, as before. Call the vector `rolls`. Then we make a vector `offset1`, consisting of the 11 elements of `rolls`, and having a 0 as the last element. Next, we make another vector, `offset2`, which has a leading 0 followed by the elements of `rolls`. A typical run might look like the below.

rolls	1	4	3	3	6	5	2	3	1	1	4	
offset1	1	4	3	3	6	5	2	3	1	1	4	0
offset2	0	1	4	3	3	6	5	2	3	1	1	4

Now, create a vector called `same.roll`, which = `offset1 - offset2` . Notice that the only elements of `same.roll` which are 0 would be where the original `rolls` vector duplicated the previous roll!

Now, all we have to do is list which locations (called indices in R vectors) we have 0's in the `same.roll` vector, and average those location numbers, to have the average number of rolls it takes to get doubles (where, in our example, we only rolled the die 11 times).

R has a handy little function, called `which()` to do the job of listing the indices of `same.roll` with 0 entries. Information on `which()` is attached at the end of this paper.

So, we would use the command

```
location ← which(same.roll ==0)
```

and our vector `location` would = 4 10 . Now we would create vector `offset3` and `offset4` vectors from `location` vector, just like we did the offset vectors before, and have the following vector `result1 = offset3 - offset4` , as shown below.

location	4	10	
offset3	4	10	0
offset4	0	4	10
result1	4	6	-10

Now we would make the last entry of `result1 = 0`, having `result1` end up

result1	4	6	0
---------	---	---	---

Finally, our answer would be `sum(result1)/(the number of elements in result1 - 1)`. In R this would be the command:

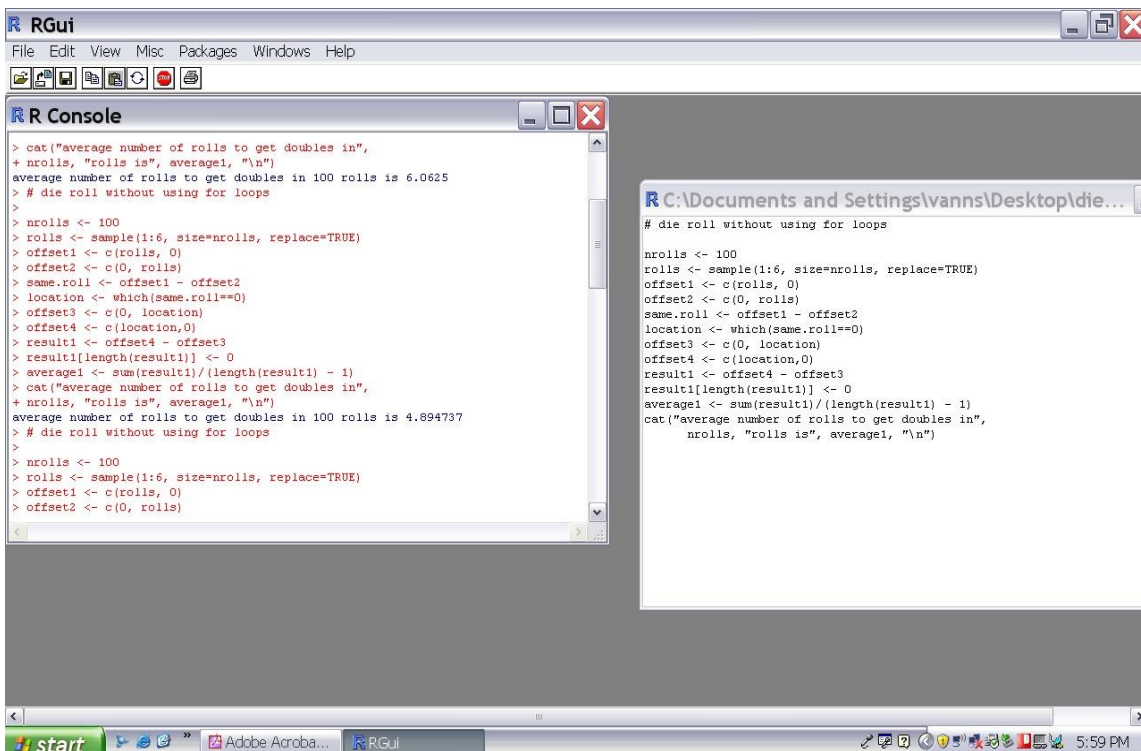
`sum(result1)/(length(result1) - 1)` , and would represent the average number of rolls in 11 tries which would result in a “doubles” being thrown.

Our script to do this, according to this plan is shown in `onedierevisited.R` , below. Note that we have 100 initial rolls of the die instead of just 11, as in our example above. Also note that the variable `average1` sums the entries of `result1`, then divides by the number of entries (i.e., `length(result1)`) less 1, so I don't include the last 0 entry of `result1` as a value for the average computed. `nrolls` is the number of rolls I use, which is 100 for my example.

```
# die roll without using for loops
```

```
nrolls <- 100
rolls <- sample(1:6, size=nrolls, replace=TRUE)
offset1 <- c(rolls, 0)
offset2 <- c(0, rolls)
same.roll <- offset1 - offset2
location <- which(same.roll==0)
offset3 <- c(0, location)
offset4 <- c(location,0)
result1 <- offset4 - offset3
result1[length(result1)] <- 0
average1 <- sum(result1)/(length(result1) - 1)
cat("average number of rolls to get doubles in",
    nrolls, "rolls is", average1, "\n")
```

A sample run or two is shown below.



Now, some of you might be scared that you could not come up with a seemingly elaborate vector indexing scheme like this. However, I just started out listing a die roll 11 times, randomly, then thought about how I could use shifting and subtraction to note when roll numbers were repeated, and after much thought and some “trial and error”, I came up with this way. If I can do it, anyone can do it. It does take patience and some thought, and, I recommend doing a small simulation by hand to “force” the result you want to achieve. Then find R commands which will do what you want.

=====

Problem #3—the 2 dice problem

[3] How many times do you expect to roll 2 dice until you get the same sum you previously rolled on the last roll?

We can approach this problem in the same way we found the value of repeating one die. In fact, we will use the same variable names where possible, to show you the parallel approach to this problem as to problem #2 (with the single die). See the script `twodiceroll.R` below.

```
# number of rolls of 2 dice
# to repeat sum

counter <- 1
roll1 <- sample(1:6, size=2, replace=TRUE)
for(i in 1:20) {
  roll2 <- sample(1:6, size=2, replace=TRUE)
  if(sum(roll1)==sum(roll2)) break
  counter <- counter + 1
  roll1 <- roll2
}
print("number of rolls to have same sum is")
counter
```

Below is the same script placed within a for loop which loops for 100 simulations, which we can graph, as we did for the 100 simulations done in the first simulation shown above.

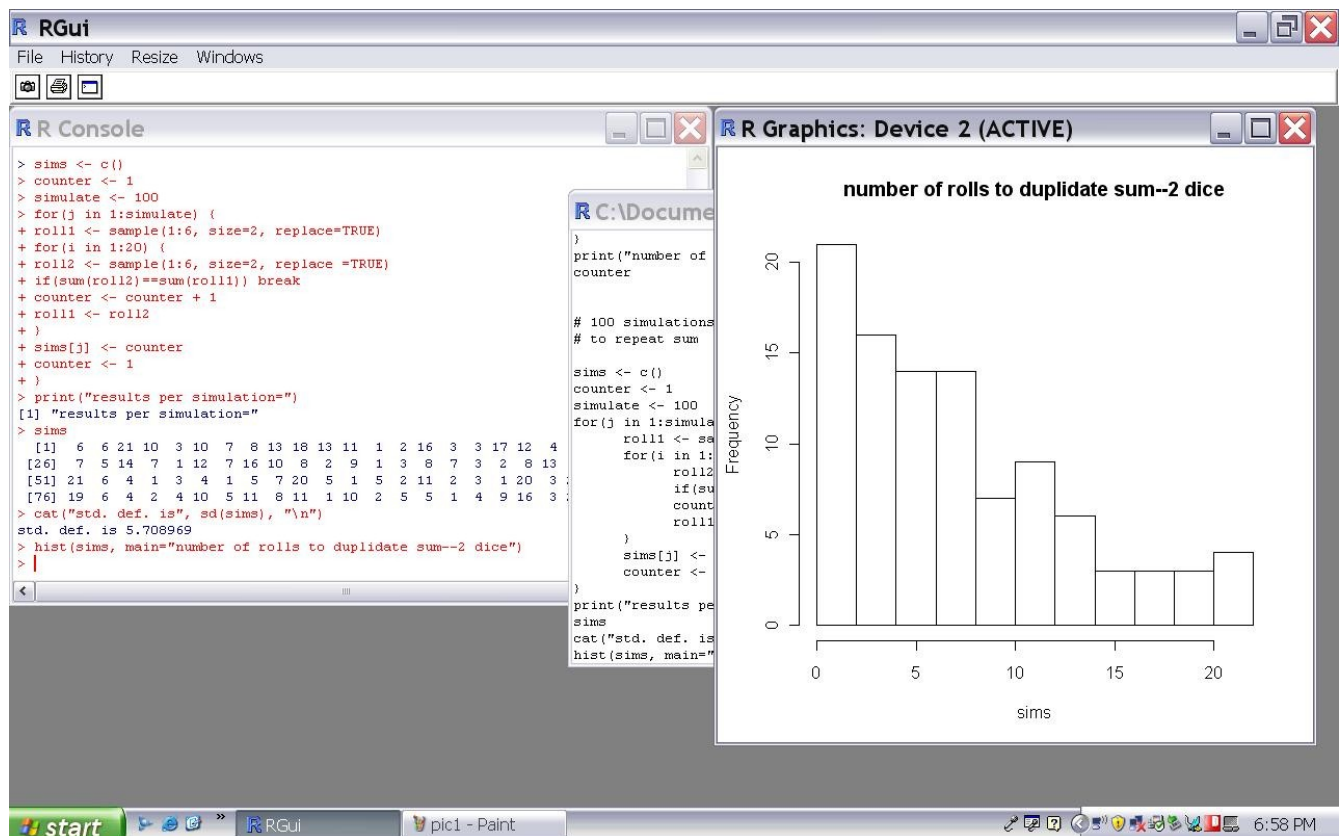
```
# 100 simulations of number of 2 dice
# to repeat sum
```

```

sims <- c()
counter <- 1
simulate <- 100
for(j in 1:simulate) {
  roll1 <- sample(1:6, size=2, replace=TRUE)
  for(i in 1:20) {
    roll2 <- sample(1:6, size=2, replace =TRUE)
    if(sum(roll2)==sum(roll1)) break
    counter <- counter + 1
    roll1 <- roll2
  }
  sims[j] <- counter
  counter <- 1
}
print("results per simulation=")
sims
cat("std. def. is", sd(sims), "\n")
hist(sims, main="number of rolls to duplicate sum--2 dice")

```

Below is some output from both scripts, the first one, and the 100 simulations of the first one.



```

RGui
File Edit Packages Windows Help
R Console
> sims <- c()
> counter <- 1
> simulate <- 100
> for(j in 1:simulate) {
+ roll1 <- sample(1:6, size=2, replace=TRUE)
+ for(i in 1:20) {
+ roll2 <- sample(1:6, size=2, replace =TRUE)
+ if(sum(roll2)==sum(roll1)) break
+ counter <- counter + 1
+ roll1 <- roll2
+ }
+ sims[j] <- counter
+ counter <- 1
+ }
> print("results per simulation=")
[1] "results per simulation="
> sims
 [1] 6 6 21 10 3 10 7 8 13 18 13 11 1 2 16 3 3 17 12 4 1 7 18 2 13
 [26] 7 5 14 7 1 12 7 16 10 8 2 9 1 3 8 7 3 2 8 13 8 11 12 5 1
 [51] 21 6 4 1 3 4 1 5 7 20 5 1 5 2 11 2 3 1 20 3 21 5 1 7 12
 [76] 19 6 4 2 4 10 5 11 8 11 1 10 2 5 5 1 4 9 16 3 21 1 3 14 5
> cat("std. def. is", sd(sims), "\n")
std. def. is 5.708969
> hist(sims, main="number of rolls to duplciate sum--2 dice")
>
R C:\Documents and Settings\vanns\Desktop\tw...
}
print("number of rolls to have same sum is")
counter

# 100 simulations of number of 2 dice
# to repeat sum

sims <- c()
counter <- 1
simulate <- 100
for(j in 1:simulate) {
  roll1 <- sample(1:6, size=2, replace=TRUE)
  for(i in 1:20) {
    roll2 <- sample(1:6, size=2, replace =TRUE)
    if(sum(roll2)==sum(roll1)) break
    counter <- counter + 1
    roll1 <- roll2
  }
  sims[j] <- counter
  counter <- 1
}
print("results per simulation=")
sims
cat("std. def. is", sd(sims), "\n")
hist(sims, main="number of rolls to duplciate sum--2 dice")

```

You may also avoid the *for* looping by setting up vectors similar to how I did it above for the single die roll. I'll let that generation be done by the more ambitious of you out there!!

=====

Problem #4—the 3 dice problem

[4] Roll 3 dice and find out how many rolls it takes to sum the same as the previous roll.

Make appropriate histograms of the result for 500 simulations. Do the results seem reasonable to you? Why or why not?

This problem is essentially the same program as the previous problem with 2 dice. The only change is to make the sample command `sample(1:6, size=3, replace=TRUE)` from the previous statement which has the `size=2` attribute. The program to do one simulation, as well as the program to do 500 simulations, are shown below.