MSAN 593 - Homework 1

Andre Guimaraes Duarte

July 17, 2016

Question 1

1

First, let's create and populate a few vectors that will be useful for this question. These vectors relate to the courses that I am taking this summer at USF.

```
courseNum <- c(501, 502, 504, 593)
courseName <- c("Computation for Analytics",
                "Review of Linear Algebra",
                "Review of Probability and Statistics",
                "Exploratory Data Analysis")
courseProf <- c("Terence Parr","David Uminski", "Jeff Hamrick", "Paul Intrevado")
enrolled <- c(T, T, F, T)
anticipatedGrade <- c("A+", "A-", NA, "A")
anticipatedHours <- c(15, 25, NA, 25)
```

From these entries, we can see that I am **not** enrolled in MSAN504: Review of Probability and Statistics with Jeff Hamrick.

Further information about these vectors (their `type` and `class`) are shown in the table below.

Table 1: Summary of the `type` and `class` for each vector.

vector	type	class
courseNum	double	numeric
courseName	character	character
courseProf	character	character
enrolled	logical	logical
anticipatedGrade	character	character
anticipatedHours	double	numeric

We can see from table 1 that, for example, `courseNum` has `type` `double` and its `class` is `numeric`.

2

Now, let's create a data frame called `bootcampDataFrame` by combining all the above vectors. When creating this data frame, it is important to remember to set the flag `stringsAsFactors` to `FALSE` (or `F`), so that strings are not converted into factors. We also call the function `str` on our data frame to give us a quick overview of its contents. `str` stand for *structure*. Note: the output of `str` can often be long, and run off the printable page, hence the inclusion of the flag `strict.width` to `"w"` –or `"wrap"`– so the results wrap around the default (or specified) `width` of the printable space.

```
bootcampDataFrame <- data.frame(courseNum = courseNum,
                                courseName = courseName,
```

```
                                courseProf = courseProf,
                                enrolled = enrolled,
                                anticipatedGrade = anticipatedGrade,
                                anticipatedHours = anticipatedHours,
                                stringsAsFactors = F)
str(bootcampDataFrame, strict.width = "w")
```

```
## 'data.frame':    4 obs. of  6 variables:
## $ courseNum : num 501 502 504 593
## $ courseName : chr "Computation for Analytics" "Review of Linear Algebra"
##    "Review of Probability and Statistics" "Exploratory Data Analysis"
## $ courseProf : chr "Terence Parr" "David Uminski" "Jeff Hamrick" "Paul
##    Intrevado"
## $ enrolled : logi TRUE TRUE FALSE TRUE
## $ anticipatedGrade: chr "A+" "A-" NA "A"
## $ anticipatedHours: num 15 25 NA 25
```

The table below summarizes the `type` and `class` for the data frame.

Table 2: Summary of the `type` and `class` for the data frame variables.

vector	type	class
courseNum	double	numeric
courseName	character	character
courseProf	character	character
enrolled	logical	logical
anticipatedGrade	character	character
anticipatedHours	double	numeric

We can see from table 2 that, here, all the variables retain their original types and classes.

3

Let's combine the vectors from the beggining of this exercise in a single list called `bootcampDataList`. As previously, it is also good practice to show the structure of the created list.

```
bootcampDataList <- list(courseNum,
                         courseName,
                         courseProf,
                         enrolled,
                         anticipatedGrade,
                         anticipatedHours)
str(bootcampDataList, strict.width = "w")
```

```
## List of 6
## $ : num [1:4] 501 502 504 593
## $ : chr [1:4] "Computation for Analytics" "Review of Linear Algebra"
##    "Review of Probability and Statistics" "Exploratory Data Analysis"
## $ : chr [1:4] "Terence Parr" "David Uminski" "Jeff Hamrick" "Paul
##    Intrevado"
```

```
## $ : logi [1:4] TRUE TRUE FALSE TRUE
## $ : chr [1:4] "A+" "A-" NA "A"
## $ : num [1:4] 15 25 NA 25
```

Now each vector is an element of this list. One notable difference to the data frame that we can see from this output is that the vectors are not named. We can name each of them by its original vector's name, so it's easier to understand what each element represents.

```
names(bootcampDataList) <- c("courseNum",
                             "courseName",
                             "courseProf",
                             "enrolled",
                             "anticipatedGrade",
                             "anticipatedHours")
str(bootcampDataList, strict.width = "w")
```

```
## List of 6
## $ courseNum : num [1:4] 501 502 504 593
## $ courseName : chr [1:4] "Computation for Analytics" "Review of Linear
##    Algebra" "Review of Probability and Statistics" "Exploratory Data
##    Analysis"
## $ courseProf : chr [1:4] "Terence Parr" "David Uminski" "Jeff Hamrick"
##    "Paul Intrevado"
## $ enrolled : logi [1:4] TRUE TRUE FALSE TRUE
## $ anticipatedGrade: chr [1:4] "A+" "A-" NA "A"
## $ anticipatedHours: num [1:4] 15 25 NA 25
```

As we can see, now the data list has proper names for our vectors.

The table below summarizes the `type` and `class` for the data list

Table 3: Summary of the `type` and `class` for the data list elements.

vector	type	class
courseNum	double	numeric
courseName	character	character
courseProf	character	character
enrolled	logical	logical
anticipatedGrade	character	character
anticipatedHours	double	numeric

We can see from table 3 that the variables all retain their original types and classes.

4

- In order to get the course numbers in `courseNum` except the fourth one, we can use the following code:

```
courseNum[c(1,2,3)]
```

```
## [1] 501 502 504
```

There are many other ways to get this result, but this one is simple and concise.

- The total number of hours I anticipate spending on coursework each week is found using a simple `sum`. **Note**: it is important to remember to include the flag `na.rm = T` in order to not consider `NA` values (or else the return value would be `NA`):

```
sum(anticipatedHours, na.rm = T) # per week
```

```
## [1] 65
```

```
sum(anticipatedHours, na.rm = T) * 5 # over all of boot camp
```

```
## [1] 325
```

I anticipate spending 65 hours on coursework per week, and 325 hours throughout all of boot camp. That's quite a bit!

- In order to get only the third row and the first two columns of `bootcampDataFrame`, we use the following code:

```
bootcampDataFrame[3, c(1,2)]
```

```
##   courseNum                          courseName
## 3       504 Review of Probability and Statistics
```

We can see it returns `courseNum` and `courseName` of the third course (`Review of Probability and Statistics`)

- The first value in the second element of `bootcampDataList` can be obtained by running the following code:

```
bootcampDataList[[2]][1]
```

```
## [1] "Computation for Analytics"
```

It is, in fact, `courseName` of the first course (`Computation for Analytics`). Note that we had to use double square brackets `[[2]]` to access the second element in the list.

5

Since we haven't done it before, let's convert the `anticipatedGrade` variable in `bootcampDataFrame` into an ordinal factor (not forgetting to set the flag `ordered = T`):

```
bootcampDataFrame$anticipatedGrade <- factor(anticipatedGrade,
                                             levels = c("F", "C-", "C", "C+", "B-", "B",
                                             "B+", "A-", "A", "A+"), ordered = T)
```

Note: there is no problem in defining more `levels` than there are grades in our data frame. It is actually better, since we can afterwards add new anticipated grades in the data frame, and we won't have to update the `factor levels`. It is a precaution that can come in handy.

We can check that it has in fact been altered by printing the `structure` of `anticipatedGrade` by using the function `str`:

```
str(bootcampDataFrame$anticipatedGrade)
```

```
##  Ord.factor w/ 10 levels "F"<"C-"<"C"<"C+"<..: 10 8 NA 9
```

Now, `anticipatedGrade` has 10 levels, ranging from F to A+.

- The maximum letter grade I anticipate receiving in boot camp is easily retrieved by using the `max` function (since the `levels` are ordered), as shown below:

```
max(bootcampDataFrame$anticipatedGrade, na.rm = T)
```

```
## [1] A+
## Levels: F < C- < C < C+ < B- < B < B+ < A- < A < A+
```

The name and course number of this class is found and printed via the code:

```
paste("MSAN",
      bootcampDataFrame$courseNum[which(bootcampDataFrame$anticipatedGrade ==
                                          max(bootcampDataFrame$anticipatedGrade,
                                              na.rm = T))],
      ":",
      bootcampDataFrame$courseName[which(bootcampDataFrame$anticipatedGrade ==
                                           max(bootcampDataFrame$anticipatedGrade,
                                               na.rm = T))])
```

```
## [1] "MSAN 501 : Computation for Analytics"
```

Therefore, the course in which I anticipate getting an A+ is MSAN 501 : Computation for Analytics. Hopefully I survive these five weeks and get these anticipated grades in all courses!

Question 2

1

First, we read the file `titanic.csv` in the current working directory, and store the data in the data frame `titanicData` (and remembering to set the flag `stringsAsFactor` to `FALSE` so that character strings are not considered individual factors). By calling `str` on this data frame, we can get a summary of its contents.

```
titanicData <- read.csv("titanic.csv", stringsAsFactors = F)
str(titanicData, strict.width = "w")
```

```
## 'data.frame':    891 obs. of  12 variables:
## $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
## $ Survived : int 0 1 1 1 0 0 0 0 1 1 ...
## $ Pclass : int 3 1 3 1 3 3 1 3 3 2 ...
## $ Name : chr "Braund, Mr. Owen Harris" "Cumings, Mrs. John Bradley
##    (Florence Briggs Thayer)" "Heikkinen, Miss. Laina" "Futrelle, Mrs.
##    Jacques Heath (Lily May Peel)" ...
## $ Sex : chr "male" "female" "female" "female" ...
## $ Age : num 22 38 26 35 35 NA 54 2 27 14 ...
## $ SibSp : int 1 1 0 1 0 0 0 3 0 1 ...
## $ Parch : int 0 0 0 0 0 0 0 1 2 0 ...
## $ Ticket : chr "A/5 21171" "PC 17599" "STON/O2. 3101282" "113803" ...
## $ Fare : num 7.25 71.28 7.92 53.1 8.05 ...
## $ Cabin : chr "" "C85" "" "C123" ...
## $ Embarked : chr "S" "C" "S" "S" ...
```

2

The first line of the output from `str` shows that there are 891 rows in this data frame. This result is also easily obtained by running the command:

```
nrow(titanicData)
```

```
## [1] 891
```

3

The first line of the output from `str` shows that there are 12 columns in this data frame. This result is also easily obtained by running the command:

```
ncol(titanicData)
```

```
## [1] 12
```

4

To view how many `NA` entries each variable has, we can print the `summary` of the data frame.

```
summary(titanicData)
```

```
##   PassengerId       Survived          Pclass          Name
##  Min.   :  1.0   Min.   :0.0000   Min.   :1.000   Length:891
##  1st Qu.:223.5   1st Qu.:0.0000   1st Qu.:2.000   Class :character
##  Median :446.0   Median :0.0000   Median :3.000   Mode  :character
##  Mean   :446.0   Mean   :0.3838   Mean   :2.309
##  3rd Qu.:668.5   3rd Qu.:1.0000   3rd Qu.:3.000
##  Max.   :891.0   Max.   :1.0000   Max.   :3.000
##
##      Sex                 Age            SibSp           Parch
##  Length:891         Min.   : 0.42   Min.   :0.000   Min.   :0.0000
##  Class :character   1st Qu.:20.12   1st Qu.:0.000   1st Qu.:0.0000
##  Mode  :character   Median :28.00   Median :0.000   Median :0.0000
##                     Mean   :29.70   Mean   :0.523   Mean   :0.3816
##                     3rd Qu.:38.00   3rd Qu.:1.000   3rd Qu.:0.0000
##                     Max.   :80.00   Max.   :8.000   Max.   :6.0000
##                     NA's   :177
##     Ticket               Fare           Cabin             Embarked
##  Length:891         Min.   :  0.00   Length:891         Length:891
##  Class :character   1st Qu.:  7.91   Class :character   Class :character
##  Mode  :character   Median : 14.45   Mode  :character   Mode  :character
##                     Mean   : 32.20
##                     3rd Qu.: 31.00
##                     Max.   :512.33
##
```

We can see that the variable `Age` is the only one with `NA` entries, and has 177 of them.

5

From the output of `summary`, we can see that the variable `Survived` was imported as a `integer`. It would make more sense if it were `logical`, since the passenger either survived or did not. We can make this modification by executing the following code:

```
titanicData$Survived <- as.logical(titanicData$Survived)
summary(titanicData$Survived)
```

```
##    Mode   FALSE    TRUE    NA's
## logical     549     342       0
```

We can see that there are 0 `NA` entries for this variable after this modification.

In addition, other variables should be converted into factors.

- The variable `Embarked`, which denotes the port of embarkation of each passanger, can be converted to a factor, with levels "S", "C", and "Q" denoting the cities "Southampton", "Queenstown", and "Cherbourg". Since the Titanic departed from Southampton, then stopped at Cherbourg then Queenstown, tt can be an interesting idea to order these factors in this fashion.
- The variable `SibSp`, which denotes the number of siblings/spouses aboard the Titanic, can be converted into an ordered factor.
- The variable `Parch` denotes the number of parents/children aboard the ship, and can be converted into an ordered factor.

- The variable `Pclass`, which is the cabin class, can also be converted into an ordered factor.
- The variable `Sex`, the sex of the passenger, can be converted into a factor with levels `male` and `female` (which are the only two values, so we don't need to specifiy the `levels` argument). Note: sexes are **not** ordered! Males and females are **equal**!

```
titanicData$Embarked <- factor(titanicData$Embarked,
                               levels = c("S", "C", "Q"),
                               ordered = T)
titanicData$SibSp <- factor(titanicData$SibSp,
                            levels = unique(titanicData$SibSp)[
                                     order(unique(titanicData$SibSp))],
                            ordered = T)
titanicData$Parch <- factor(titanicData$Parch,
                            levels = unique(titanicData$Parch)[
                                     order(unique(titanicData$Parch))],
                            ordered = T)
titanicData$Pclass <- factor(titanicData$Pclass,
                             levels = unique(titanicData$Pclass)[
                                      order(unique(titanicData$Pclass))],
                             ordered = T)
titanicData$Sex <- factor(titanicData$Sex)
str(titanicData, strict.width = "w")
```

```
## 'data.frame':    891 obs. of  12 variables:
## $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
## $ Survived : logi FALSE TRUE TRUE TRUE FALSE FALSE ...
## $ Pclass : Ord.factor w/ 3 levels "1"<"2"<"3": 3 1 3 1 3 3 1 3 3 2 ...
## $ Name : chr "Braund, Mr. Owen Harris" "Cumings, Mrs. John Bradley
##    (Florence Briggs Thayer)" "Heikkinen, Miss. Laina" "Futrelle, Mrs.
##    Jacques Heath (Lily May Peel)" ...
## $ Sex : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2 2 1 1 ...
## $ Age : num 22 38 26 35 35 NA 54 2 27 14 ...
## $ SibSp : Ord.factor w/ 7 levels "0"<"1"<"2"<"3"<..: 2 2 1 2 1 1 1 4 1 2
##    ...
## $ Parch : Ord.factor w/ 7 levels "0"<"1"<"2"<"3"<..: 1 1 1 1 1 1 1 2 3 1
##    ...
## $ Ticket : chr "A/5 21171" "PC 17599" "STON/O2. 3101282" "113803" ...
## $ Fare : num 7.25 71.28 7.92 53.1 8.05 ...
## $ Cabin : chr "" "C85" "" "C123" ...
## $ Embarked : Ord.factor w/ 3 levels "S"<"C"<"Q": 1 2 1 1 1 3 1 1 1 2 ...
```

We can see from the `structure` that the classes have been properly updated. Our data frame is better structured now.

6

- The mean age of survivors is 29.7 years.
- The mean age of those who did not survive is 30.63 years.
- To plot side-by-side histograms of the ages of survivors and non-survivors, we use the following code, and generate the subsequent image.

```
par(mfrow = c(1,2))
hist(titanicData$Age[titanicData$Survived == T],
     xlab = "Age (years)", main = "Histogram of the ages\n of surviors")
hist(titanicData$Age[titanicData$Survived == F],
     xlab = "Age (years)", main = "Histogram of the ages\n of non-surviors")
```

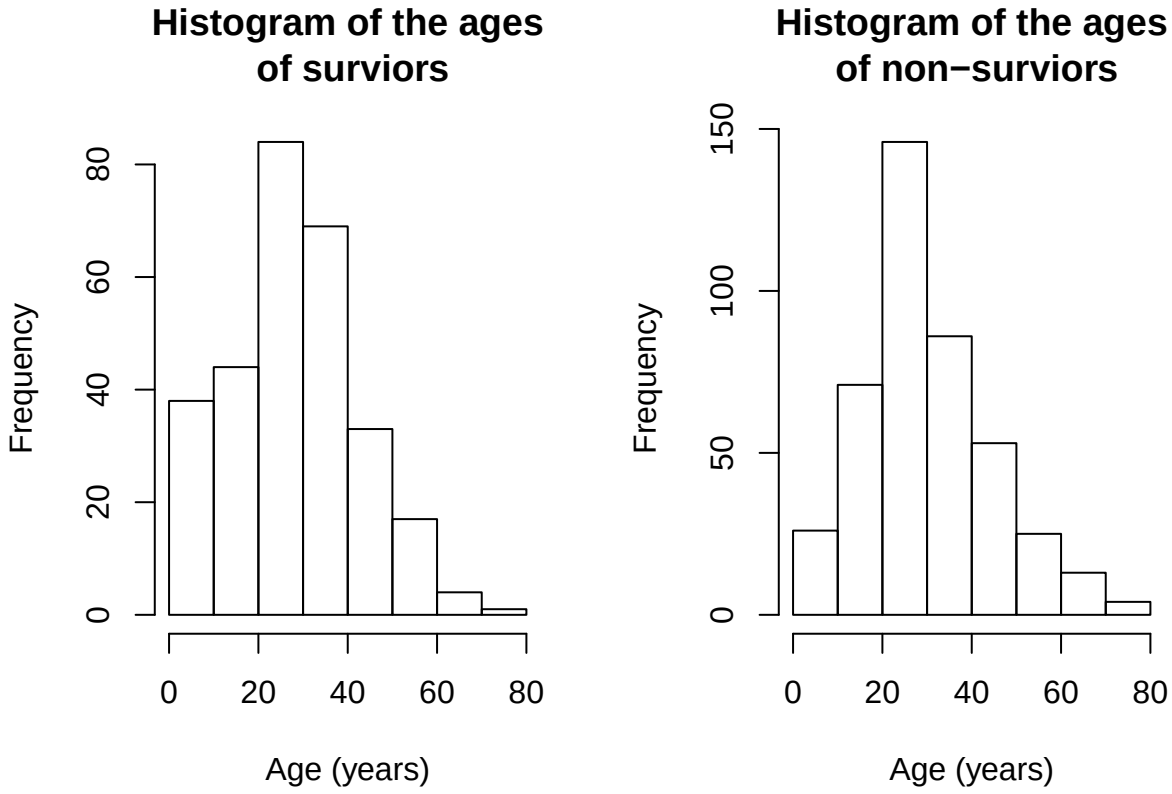

Figure 1: Histograms of the ages of survivors and non-survivors

The histograms in figure 1 show the distribution of ages of the people who survived and those who did not survive the sinking of the ship. It can be seen that the histogram for the survivors is slightly more dense on the left (toward younger ages) than the histogram for the non-survivors: there are more young people proportionally to the total among the survivors than the non-survivors.

7

The first 10 values of the `Cabin` variable are shown below.

```
titanicData$Cabin[1:10]
```

```
##  [1] ""     "C85"  ""     "C123" ""     ""     "E46"  ""     ""     ""
```

We can see that many are blank. Let's write a script that replaces all blanks in the **entire** data frame `titanicData` with `NA`s.

```
# for every variable
for(i in 1:ncol(titanicData)){
  # for every value of the variable
  for(j in 1:nrow(titanicData)){
    # if the value is not NA and is a blank
    if(!is.na(titanicData[[i]][j]) & titanicData[[i]][j] == ""){
      # replace the value with NA
      titanicData[[i]][j] <- NA
    }
  }
}
```

We can see that now the variable `Cabin` has 687 `NA`s, making it the one with the most `NA`s:

```
sum(is.na(titanicData$Cabin))
```

```
## [1] 687
```

8

Around 19.87% of the observations for `Age` are `NA`s, which is almost one in five. We can replace all of them with the mean age with the following line of code:

```
titanicData$Age[is.na(titanicData$Age)] <- mean(titanicData$Age, na.rm = T)
summary(titanicData$Age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.42   22.00   29.70   29.70   35.00   80.00
```

By comparison the output of `summary` to the one shown in sub-question 4, we can see that the minimum, maximum, and mean have not changed (and nor should they have, since we only introduced values equal to the mean!). However, the median is now the same as the mean (29.7), while the first and third quantiles have been brought nearer to this value.

This technique is called *imputation*, and more specifically *mean* imputation. While this method does not introduce too much bias in a distribution that is normally distributed, it can have serious drawbacks if the original distribution is skewed. Indeed, let's say that a variable is skewed towards higher values. By implementing mean imputation, the histogram of the values for the variable will shift towards the mean, which is not realistic considering the original distribution. Other imputation methods (median, Hotdeck, ...) can lead to more likely results.

Question 3

1

- In order to generate a random sample of `n = 100` random variable $\sim \mathcal{U}\{-1, 1\}$, we use the following code. Note: it is good practice to save variables (in this case `a` and `b`), so that, if you need to change their values at one point, you only need to do it in one place, and the changes are automatically reflected in the rest of the script.

```
a <- -1
b <- 1
u100 <- runif(100, a, b)
(u100_mean <- mean(u100))
```

```
## [1] -0.02152274
```

```
(u100_var <- var(u100))
```

```
## [1] 0.3028889
```

We can see that the mean of this sample is -0.0215, and the variance is 0.3029.

- We can repeat the previous code for sample sizes of 1,000, 10,000, 100,000, and 1,000,000. In order to do so, we can use a `for` loop. We store our samples in a vector `runifSamples`, the means in `runifMeans`, and the variances in `runifVars`. Note: `runifSamples` is a `list`, since it will contain separate vectors. To access one of its elements, we will need to use double square brackets (for example: `[[1]]` to access the first element in the list).

```
# initialize the vectors
runifSamples <- list()
runifMeans <- vector()
runifVars <- vector()

# add the previous results
runifSamples[[1]] <- u100
runifMeans[1] <- u100_mean
runifVars[1] <- u100_var

# populate the vectors
i <- 2 # 1 is already taken by u100!
for(n in c(1000, 10000, 100000, 1000000)){
  runifSamples[[i]] <- runif(n, a, b)
  runifMeans[i] <- mean(runifSamples[[i]])
  runifVars[i] <- var(runifSamples[[i]])
  i <- i + 1 # don't forget to increment i
}
```

Table 4: summary of means and variances for random samples of size `n` from $\mathcal{U}\{-1, 1\}$.

n	mean	variance
100	-0.0215227	0.3028889
1,000	-0.0142615	0.3362088
10,000	-0.0044045	0.330597
100,000	$-1.1026604 \times 10^{-4}$	0.3337522
1,000,000	$-6.4175929 \times 10^{-4}$	0.3334016

We can see from table 4 that, as `n` gets larger and larger, the mean and variance both tend to their respective theoretical values of $\frac{a+b}{2} = 0$ and $\frac{(b-a)^2}{12} = 0.3333333$.

- Now, let's create a data frame called `unifDataFrame` with seven variables: `sampleSize`, `theoreticalMean`, `sampleMean`, `deltaMean`, `theoreticalVariance`, `sampleVariance`, `deltaVariance`, where `deltaMean` and `deltaVariance` are the differences between the sample and theoretical mean and variances respectively for each sample size.

First, let's create a vector with our desired sample sizes, and call it `sampleSize`:

```
sampleSize <- vector()
for(i in 1:length(runifSamples)){
  sampleSize[i] <- length(runifSamples[[i]])
}
```

Now, let's create vectors `theoreticalMean` and `theoreticalVariance`. They have the same length as `sampleSize`, and contain repetitions of 0 and 0.3333333 respectively. We can use the built-in function `rep` to achieve this:

```
theoreticalMean <- rep((a+b)/2, length(sampleSize))
theoreticalVariance <- rep(((b-a)**2)/12, length(sampleSize))
```

Now, let's create the vectors `sampleMean` and `sampleVariance`:

```
sampleMean <- runifMeans
sampleVariance <- runifVars
```

Let's now create our last two vectors, `deltaMean` and `deltaVariance`:

```
deltaMean <- abs(sampleMean - theoreticalMean)
deltaVariance <- abs(sampleVariance - theoreticalVariance)
```

Finally, we can combine all these vectors into our data frame:

```
unifDataFrame <- data.frame(sampleSize = sampleSize,
                            theoreticalMean = theoreticalMean,
                            sampleMean = sampleMean,
                            deltaMean = deltaMean,
                            theoreticalVariance = theoreticalVariance,
                            sampleVariance = sampleVariance,
                            deltaVariance = deltaVariance)
str(unifDataFrame, strict.width = "w")
```

```
## 'data.frame':    5 obs. of  7 variables:
## $ sampleSize : int 100 1000 10000 100000 1000000
## $ theoreticalMean : num 0 0 0 0 0
## $ sampleMean : num -0.021523 -0.014261 -0.004404 -0.00011 -0.000642
## $ deltaMean : num 0.021523 0.014261 0.004404 0.00011 0.000642
## $ theoreticalVariance: num 0.333 0.333 0.333 0.333 0.333
## $ sampleVariance : num 0.303 0.336 0.331 0.334 0.333
## $ deltaVariance : num 3.04e-02 2.88e-03 2.74e-03 4.19e-04 6.83e-05
```

Calling `str` on our data frame allows to quickly visualize the contents. More specifically, we can see that both `deltaMean` and `deltaVariance` become increasingly small as `n` grows large.

- An easy way to evaluate the effect of the sample size on the precision of the mean is to plot `deltaMean` as a function of `sampleSize`. Since we are increasing the sample size `n` by factors of 10, it makes sense to use a logarithmic scale for the x axis.

```
plot(deltaMean ~ sampleSize, log = "x",
     main = "Impact of the sample size on the delta mean",
     ylab = "Delta mean", xlab = "Sample size")
```

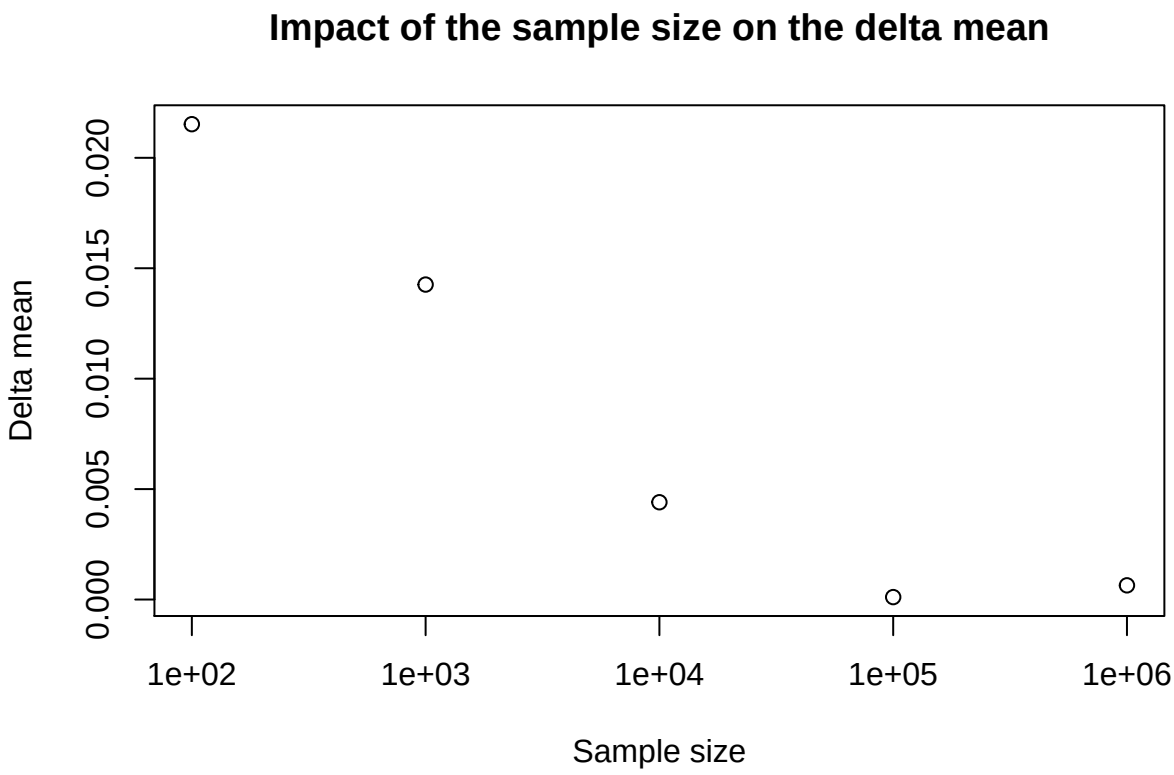

Figure 2: Effect of sample size on the difference between theoretical mean and sample mean for a uniform distribution between -1 and 1

- As previously, a similar plot can be constructed for `deltaVariance`.

```r
plot(deltaVariance ~ sampleSize, log = "x",
     main = "Impact of the sample size on the delta variance",
     ylab = "Delta variance", xlab = "Sample size")
```

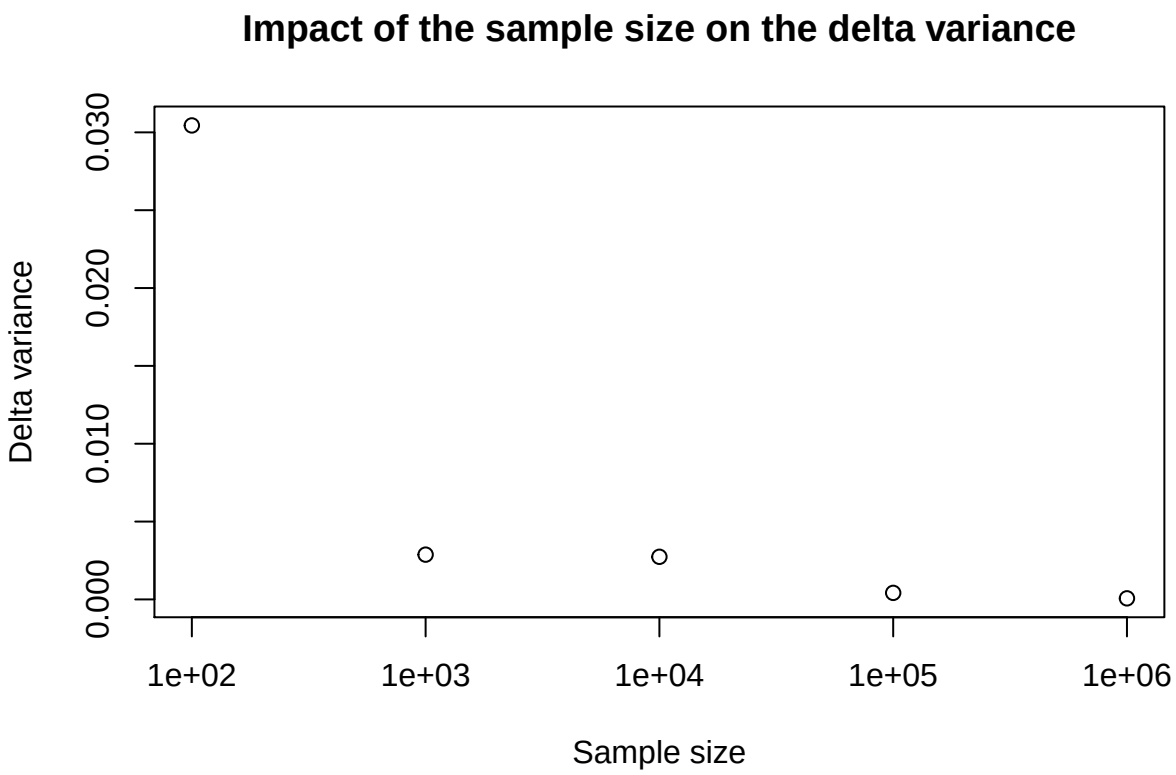

Figure 3: Effect of sample size on the difference between theoretical variance and sample variance for a uniform distribution between -1 and 1

The two plots, shown in figures 2 and 3, show that, as the sample size increases, the difference between the theoretical mean (respectively variance) and the sample mean (respectively variance) tends to zero. The larger the sample size, the more accurate the sample mean and variance are.

2

Let's create a vector `myRunifVec` containng 10,000,000 random variables $\sim \mathcal{U}\{0, 1\}$.

```r
a <- 0
b <- 1
myRunifVec <- runif(10000000, a, b)
```

Now, we randomly sample 100,000 values from this vector a draw the corresponding histogram. The flag `freq` is set to `FALSE` in order to plot the histogram with densities instead of frequencies.

```r
hist(sample(myRunifVec, 100000), freq = F,
     main = "Distirbution of 100,000 uniformely distributed random variables",
     xlab = expression(u%~%U(0,1)), ylab = "Density")
```

The histogram in figure 4 corresponds to the distribution of a random variable that follows a uniform distribution from 0 to 1. When randomly sampling from a $\mathcal{U}\{a, b\}$ distribution, the sample is also $\sim \mathcal{U}\{a, b\}$.

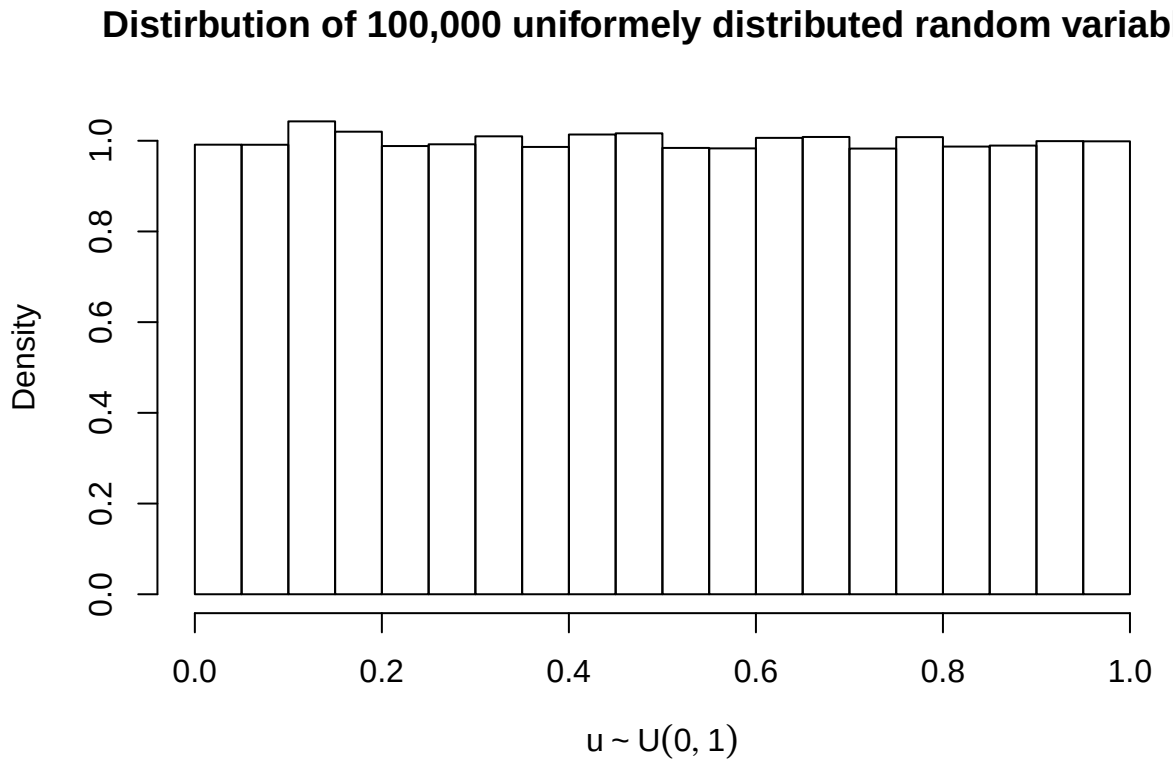

Figure 4: Histogram of 100,000 values randomly sampled from a uniform distirbution from 0 to 1

3

Now, let's create a data frame `myRunifDataFrame` containing three columns:

- the first two, `col1` and `col2`, contain two separate samples of 10,000,000 random variables samples from a $\sim \mathcal{U}\{0, 1\}$ distribution.
- the third, `runifSum`, contains the sum of `col1` and `col2`. We also call `str` on the data frame to make sure everything was created correctly.

```
col1 <- runif(10000000, 0, 1)
col2 <- runif(10000000, 0, 1)
runifSum <- col1 + col2
myRunifDataFrame <- data.frame(col1 = col1,
                               col2 = col2,
                               runifSum = runifSum)
str(myRunifDataFrame, strict.width = "w")
```

```
## 'data.frame':    10000000 obs. of  3 variables:
## $ col1 : num 0.639 0.755 0.185 0.306 0.72 ...
## $ col2 : num 0.0617 0.6758 0.4536 0.732 0.7879 ...
## $ runifSum: num 0.7 1.431 0.638 1.038 1.508 ...
```

Now, let's create a histogram from the values in `runifSum`, which we call from `myRunifDataFrame` using the `$` symbol.

```r
hist(myRunifDataFrame$runifSum, freq = F,
     main = "Convolution histogram for the sum of two
     uniform random variables", xlab = expression(u[1] + u[2]),
     ylab = "Density", ylim = c(0, 1))
```

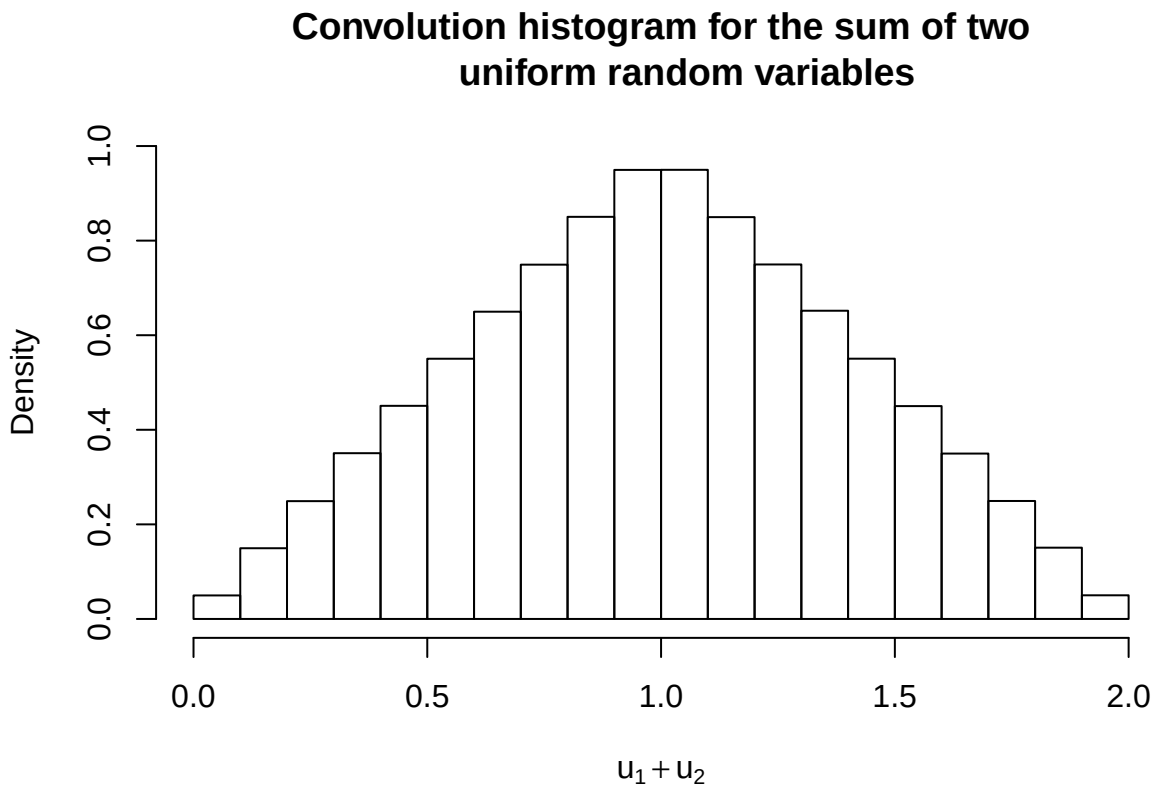

Figure 5: Convolution histogram for the sum of two uniformely distributed random variables ~U(0, 1)

We can immediately see, in figure 5, that the shape of this histogram is not similar to what we would expect from a uniform distribution. The act of adding several independent random variables is called a *convolution*. In this case, with random variables from a uniform distribution, it is called the *Irwin-Hall distribution*. The triangular shape of this distribution is fairly easy to understand. We are adding random variables that are uniformly distributed from 0 to 1. Therefore, the sum of two of these random variables will be between `0` and `2`, but with a higher probability of it being toward the center (in this case, `1`). In fact, for the sum to be around `0` or `2`, both random variables will have to be close to `0` or `1` respectively. On the other hand, many variations are possible for the sum to be around `1`. Therefore, we get more values around `1` than near the extremities.

4

Finally, let's repeat the previous exercise, but instead of sampling from a uniform distribution, we will be sampling from an exponential distribution with parameter $\lambda = 1$.

```r
col11 <- rexp(10000000, 1)
col21 <- rexp(10000000, 1)
rexpSum <- col11 + col21
myRexpDataFrame <- data.frame(col11 = col11,
                              col21 = col21,
```

```
                                rexpSum = rexpSum)
str(myRexpDataFrame, strict.width = "w")
```

```
## 'data.frame':    10000000 obs. of  3 variables:
## $ col11 : num 0.2058 1.56 0.1576 0.5776 0.0924 ...
## $ col21 : num 4.6248 0.0773 1.3193 0.5991 0.1178 ...
## $ rexpSum: num 4.83 1.64 1.48 1.18 0.21 ...
```

Let's now plot the histogram the convolution (`rexpSum`):

```
hist(myRexpDataFrame$rexpSum, freq = F,
     main = "Convolution histogram for the sum of two
     random exponential variables with lambda = 1",
     xlab = expression(Exp[1] + Exp[2]), ylab = "Density",
     ylim = c(0, 0.4))
```

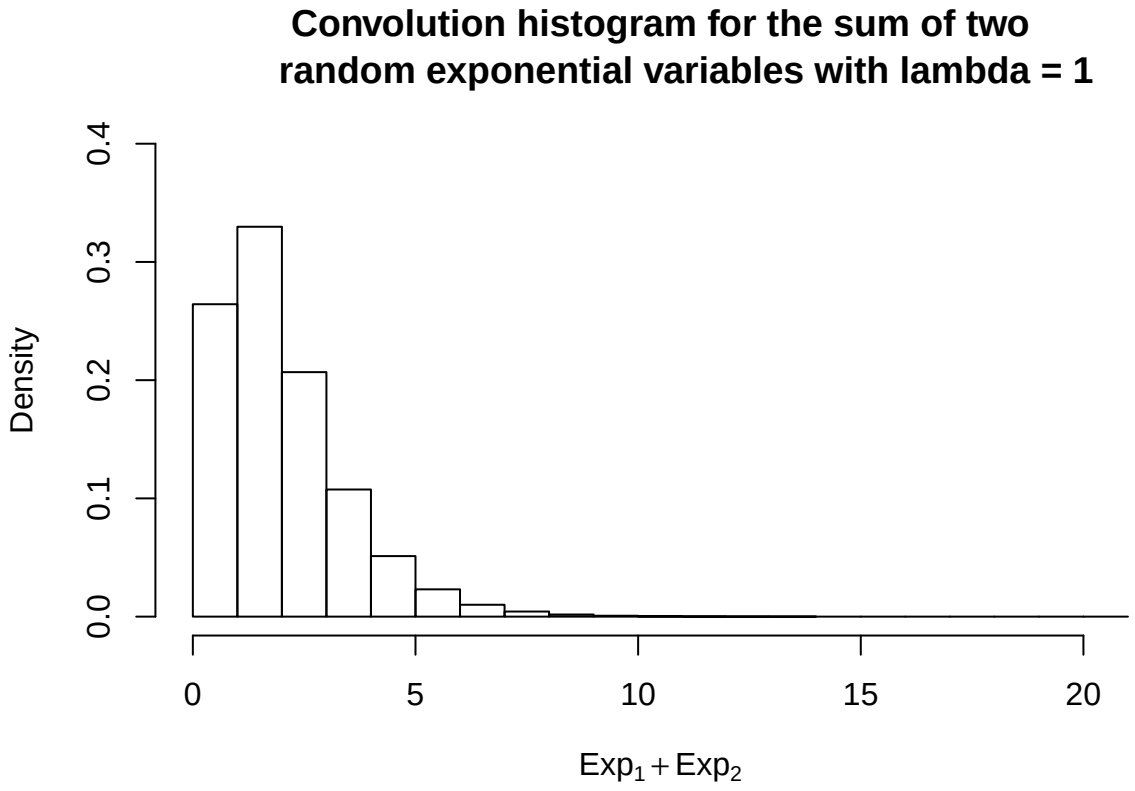

Figure 6: Convolution histogram for the sum of two uniformely distributed random variables ~Exp(lambda)

The convolution of two independent exponentially distributed random variables results in a Gamma distribution. In the present case, where the two exponentially distributed random variables have the same rate parameter $\lambda = 1$, the resulting convolution distribution, as seen in figure 6, is equivalent to a Gamma distribution with parameter $\alpha = 2$, which is similar to the histogram above. In fact, we can easily plot the Gamma distribution with $\alpha = 2$ on top of the histogram:

```
hist(myRexpDataFrame$rexpSum, freq = F,
     main = "Convolution histogram for the sum of two
```

```
     random exponential variables with lambda = 1",
     xlab = expression(Exp[1] + Exp[2]), ylab = "Density",
     ylim = c(0, 0.4))
x <- seq(0,20,0.1)
y <- dgamma(x, shape = 2)
lines(y~x, col = "blue", legend(5, 0.3, "Gamma distribution (alpha=2)",
                                col = "blue", lty = 1, cex = 0.9))
```

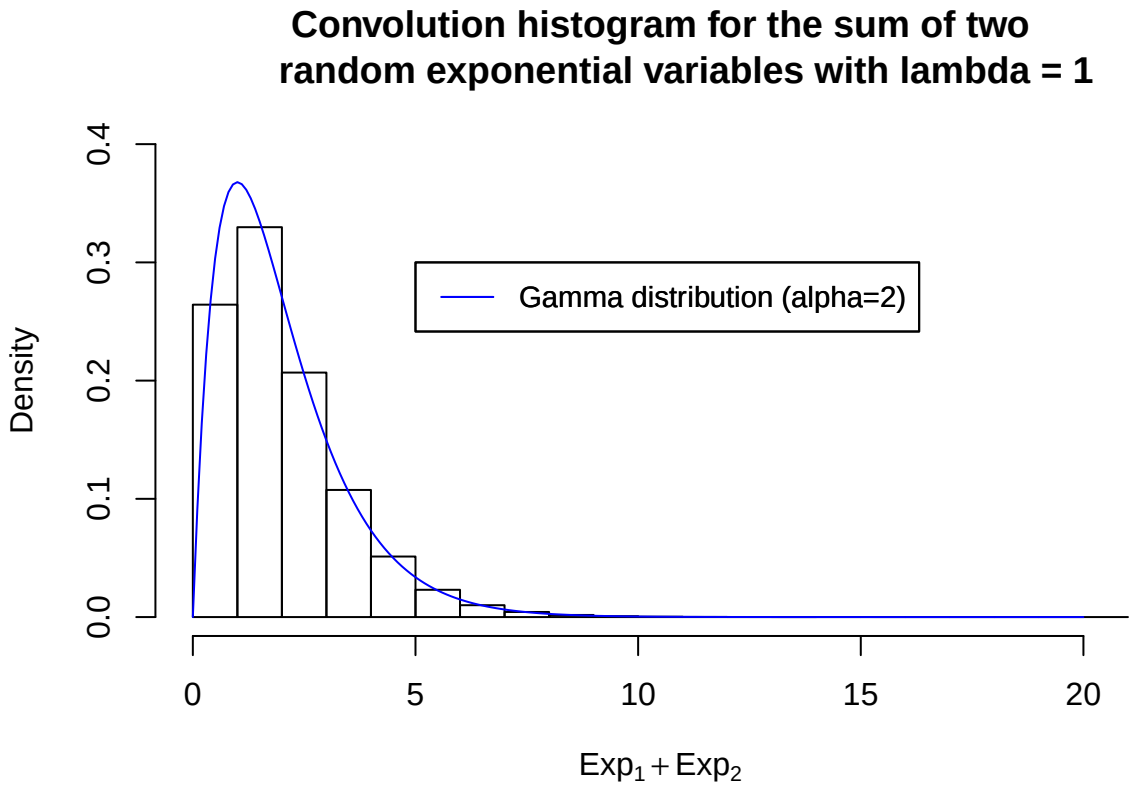

Figure 7: Convolution histogram for the sum of two uniformely distributed random variables ~Exp(lambda)

As we can see in figure 7, the histogram closely resembles the theoretical distribution of a Gamma random variable with $\alpha = 2$.